□□

Rechnen mit Sage

Calcul mathématique avec

SAGE

Die Autoren des Originals:
Alexandre Casamayou Nathan Cohen
Guillaume Connan Thierry Dumont Laurent Fousse
François Maltey Matthias Meulien Marc Mezzarobba
Clément Pernet Nicolas M. Thiéry Paul Zimmermann

Aus dem Französischen übertragen,
an Sages aktuelle Version 7 angepasst, soweit möglich,
sonst mit einem Hinweis versehen
und in LaTeX gesetzt von Helmut Büch, Gifhorn

Inhaltsverzeichnis

Vorwort zur französischen Ausgabe

Dieses Buch ist für alle diejenigen gedacht, die ein System für mathematische Berechnungen, insbesondere das Programm Sage, effizient nutzen möchten. Diese Systeme bieten eine Fülle von Funktionen und herauszufinden, wie ein gegebenes Problem gelöst werden kann, ist nicht immer einfach. Ein Referenzhandbuch liefert zu jeder Funktion des Systems eine analytische und detaillierte Beschreibung, doch muss man den Namen der Funktion wissen, die man sucht! Der hier eingenommene Standpunkt ist ergänzend, wir geben mit Betonung der zugrunde liegenden Mathematik eine nach Themen geordnete Übersicht über die Klassen der zu lösenden Probleme und die entsprechenden Algorithmen.

Der erste Teil, der mehr an Sage ausgerichtet ist, zeigt die Handhabung des Systems. Dieser Teil sollte allen Studenten wissenschatlich-technischer Fächer (BTS, IUT, Vorbereitungskurse, Bachelor)[1] zugänglich sein und in gewissem Umfang auch Schülern der gymnasialen Oberstufe. Die anderen Teile wenden sich an Studenten in höheren Semestern[2]. Anders als in einem Referenzhandbuch werden die mathematischen Begriffe klar formuliert, bevor sie mit Sage umgesetzt werden. Damit ist dieses Buch auch ein Buch über Mathematik.

Zur Illustration dieser Vorgehensweise fällt die Wahl natürlich auf Sage, denn das ist freie Software, die jedermann nach Belieben benutzen, verändern und verteilen darf. So wird ein Schüler, der auf dem Gymnasium Sage kennen gelernt hat, es auch auf seinem beruflichen Wege anwenden, als Bachelor, Master oder Doktorand, auf der Ingenieurschule, im Unternehmen usw. Sage ist im Vergleich zu konkurrierender Software ein noch junges Programm und trotz seiner vielseitigen Fähigkeiten enthält es noch zahlreiche Fehler. Doch dank seiner sehr aktiven Commumity entwickelt sich Sage sehr rasch. Jeder Anwender von Sage kann auf `trac.sagemath.org` oder über die Liste `sage-support` einen Fehler melden - und vielleicht sogar dessen Behebung.

Bei der Erstellung dieses Buches hat uns die Version 5.9 vorgelegen. Trotzdem sollten die Beispiele mit jeder weiteren Version funktionieren, hingegen könnten bestimmte Aussagen nicht mehr stimmen, beispielsweise weil Sage zur Auswertung numerischer Integrale Maxima verwendet.

Als ich im Dezember 2009 Alexandre Casamayou, Guillaume Connan, Thierry Dumont, Laurent Fousse, François Maltey, Matthias Meulien, Marc Mezzarobba, Clément Pernet und Nicolas M. Thiéry vorgeschlagen habe, über Sage ein Buch zu schreiben, haben alle trotz erheblicher Arbeitsbelastung prompt geantwortet, wie auch Nathan Cohen, der sich uns bei diesem Abenteur angeschlossen hat. Ich lege Wert darauf, Ihnen allen zu danken, besonders dafür, dass sie den engen Zeitplan eingehalten haben, den ich vorgegeben hatte, und ganz besonders Nathan Cohen, Marc Mezzarobba und Nicolas Thiéry für ihr großes Engagemant auf der Zielgeraden.

Alle Autoren danken folgenden Personen, die die Vorabversionen dieses Buches gelesen haben: Gaëtan Bisson, Françoise Jung, Hugh Thomas, Anne Vaugon, Sébastien Desreux, Pierrick Gaudry. Maxime Huet, Jean Thiéry, Muriel Shan Sei Fan, Timothy Walsh, Daniel Duparc, Kévin Rowanet und Kamel Naroun (eine besondere Erwähnung dieser beiden, die Druckfehler gefunden haben, die 17 Korrekturlesungen wiederstanden haben); wie auch Emmanuel Thomé

[1] Brevet de Technicien Supérieur, Institute Universitaire des Technologie

[2] étudiants au niveau agrégation

für deine wertvolle Hilfe bei der Realisierung dieses Buches, Sylvain Chevillard, Gaëtan Bisson und Jérémie Detrey für ihre besonnenen typographischen Ratschläge und für die Fehler, die sie gefunden haben. Der Entwurf des Umschags wurde nach einer originellen Idee von Albane Saintenoy von Corinne Thiéry realisiert.

Beim Schreiben dieses Buches haben wir viel über Sage gelernt. Wir haben dabei natürlich auch Fehler gefunden, von denen einige schon korrigiert worden sind. Wir hoffen, dass dieses Buch auch anderen von Nutzen sein wird, Schülern, Studenten, Lehrern, Ingenieuren, Forschern oder Amateuren! Die Arbeit enthält sicherlich noch zahlreiche Unvollkommenheiten, umgekehrt erwarten wir deshalb vom Leser, dass er uns alle Fehler anzeigt und uns auch Kritik und Anregungungen für eine spätere Auflage zukommen lässt: danke, dass Sie dafür die Seite `sagebook.gforge.inria.fr` benutzen.

Nancy, Frankreich
Mai 2013 Paul Zimmermann

Teil I.

Die Handhabung von Sage

1. Erste Schritte

Dieses einführende Kapitel präsentiert die *Denkungsart* des Mathematikprogramms Sage. Die anderen Kapitel dieses ersten Teils entwickeln die Grundbegriffe von Sage: Ausführung numerischer oder symbolischer Rechnungen in der Analysis, Operationen auf Vektoren und Matrizen, Schreiben von Programmen, Verarbeitung von Datenlisten, Entwerfen von Grafiken usw. Die folgenden Teile dieses Buches vertiefen einige Zweige der Mathematik, bei denen die Informatik eine hohe Leistungsfähigkeit unter Beweis stellt.

1.1. Das Programm Sage

1.1.1. Ein Werkzeug für Mathematiker

Sage ist eine Software, die mathematische Algorithmen aus vielen Gebieten implementiert. In erster Linie operiert das System auf verschidenen Zahlenarten: den ganzen Zahlen oder den rationalen, mit variabler Präzision auf den numerischen Näherungen der reellen und der komplexen Zahlen oder auch auf den Elementen der endlichen Körper. Sehr schnell kann sich der Anwender Sages als wissenschatlichem und evtl. grafischem Taschenrechner bedienen.

Doch das Rechnen beschränkt sich nicht auf Zahlen. Die Studenten lernen beispielsweise, Systeme inhomogener linearer Gleichungen zu lösen, Ausdrücke, die manchmal Variable enthalten, auszumultiplizieren, zu vereinfachen und in Faktoren zu zerlegen. Sage ist auch und nicht zuletzt ein System zum symbolischen Rechnen, in der Lage, algebraische Rechnungen auszuführen („Rechnen mit Buchstaben“) - und für anspruchsvollere Aufgaben auch auf Polynomringen oder Körpern von Quotienten von Polynomen.

In der Analysis[1] kennt Sage die gebräuchlichen Funktionen wie Quadratwurzel, Potenz, Logarithmus und trigonometrische Funktionen und die damit gebildeten Ausdrücke. Sage leitet ab, integriert und berechnet Grenzwerte, vereinfacht Summen, entwickelt Reihen, löst verschiedene Typen von Differentialgleichungen.

Sage führt die in der linearen Algebra auf Vektoren, Matrizen und Vektorräumen üblichen Operationen aus, Sage ermöglicht auch den Umgang mit Wahrscheinlichkeiten, mit Statistik und mit Fragen der Kombinatorik.

So sind die von Sage behandelten Gebiete der Mathematik recht vielfältig, von der Gruppentheorie bis zur numerischen Analysis. Sage kann die erhaltenen Ergebnisse auch als ebene oder räumlichelder und als Animationen darstellen.

[1] In Frankreich ist der Begriff Analysis etwas weiter gefasst als in Deutschland, wo nur Differntial- und Integralrechnung dazu zählen. In Frankreich gehören auch alle Themen dazu, die Leonhard Euler im ersten Band seiner „Introductio in analysin infinitorum“ behandelt hat.

Ziele und Entwicklung von Sage

William Stein, ein Mathematiker aus den USA, Forscher und Hochschullehrer, beginnt 2005 mit der Entwicklung von Sage mit dem Ziel, eine Mathematik-Software zu schreiben, deren ganzer Code zugänglich und lesbar sein sollte. Das Programm ist zunächst auf das Arbeitsgebiet seines Autors zugeschnitten, die Zahlentheorie. Nach und nach formiert sich eine internationale Community von mehreren hundert Entwicklern, die meisten Forscher oder Lehrende. Mit der Zeit erweitert sich die Funktionalität der Beiträge, um auch andere Gebiete der Mathematik abzudecken, was aus Sage den Software-Generalisten gemacht hat, der es heute ist.

Nicht nur, dass Sage kostenlos herunter geladen und verwendet werden kann, Sage ist auch frei. Seine Autoren verzichten auf jede Beschränkung bei der Verteilung, der Installation, ja selbst der Modifikation, vorausgestzt, dass die modifizierten Versionen ebenfalls frei sind. Wäre Sage ein Buch, könnte das in allen Bibliotheken ohne weiteres verliehen oder fotokopiert werden. Diese Erlaubnis harmoniert mit der Verbreitung von Wissen, wie sie durch Forschung und Lehre angestrebt wird.

Die Entwicklung von Sage geht relativ schnell vor sich, denn es bevorzugt die Einbindung bereits existierender freier Software. Sage selbst ist in Python geschrieben, einer weithin verwendeten und leicht zu erlernenden Programmiersprache. Dieser Code ruft andere bereits existierende Mathematikprogramme auf, die mit Sage geliefert werden.

Wie fast alle Mathematikprogramme arbeitet auch Sage mit Befehlen, die in einer Programmiersprache geschrieben sind. Man kann dennoch nicht von einer eigenen Sprache Sage reden: es ist Python, das diese Rolle übernimmt, und zwar mit etlichen syntaktischen Abkürzungen zur Erleichterung der mathematischen Notation. Die vom Anwender eingegebenen Befehle werden durch den Interpreter Python ausgewertet. Für komplexe oder auch nur sich wiederholende Rechnungen ist es möglich, veritable Programme zu schreiben statt der zeilenweisen Eingabe von Befehlen, und sie sogar, warum nicht, zur Einbindung in Sage vorschlagen.

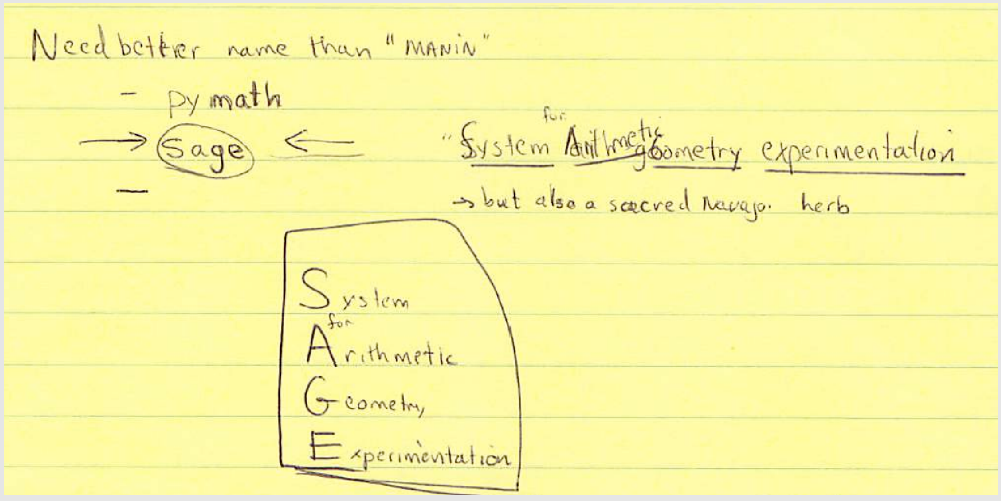


Abb. 1.1 - Das erste Auftreten des Namens Sage in einem Heft von W. Stein. Ursprünglich war Sage auch ein Akronym, sobald sich das System aber auf die Gesamtheit der Mathematik erweitert hat, ist allein der Name Sage geblieben, der im Englischen Salbei[a] bedeutet.

[a] Ein stilisiertes Bild dieser Heilpflanze ziert denn auch den Buchdeckel des französischen Originals. Natürlich kann man *sage* auch mit *weise* übersetzen.

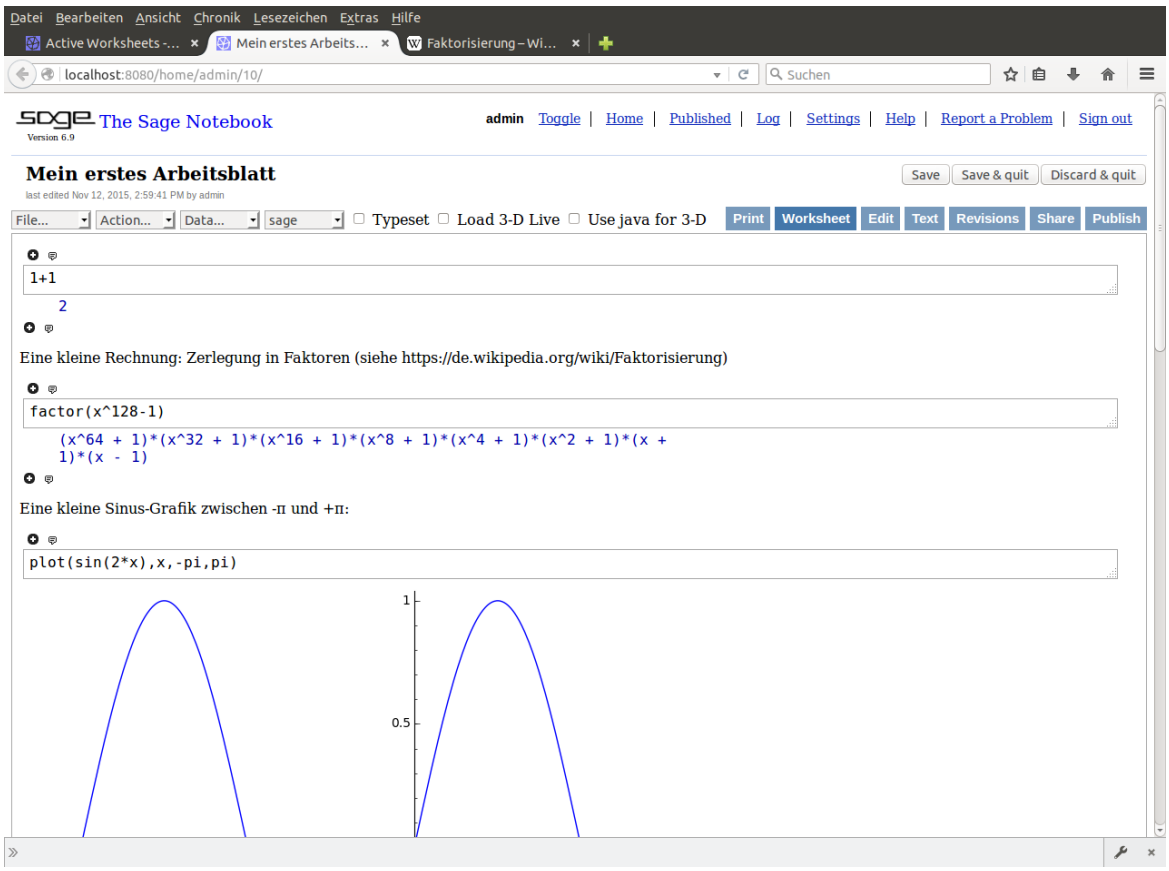


Abb. 1.2 - Ein Arbeitsblatt im Sage-Notebook

Die Verwendung einer und derselben Software zur Behandlung unterschiedlicher Aspekte der Mathematik befreit den Mathematiker, welchen Niveaus auch immer, von der Notwendigkeit, Daten zwischen verschiedenen Werkzeugen hin und her zu schieben und vom Erlernen der Syntax verschiedener Programmiersprachen. Sage will auf seinen verschiedenen Einsatzgebieten homogen sein.

1.1.2. Zugang zu Sage

Die im Buch beschriebene Möglichkeit, ein Sage-Notebook auf einem öffentlichen Server im Internet einfach durch Aufruf der Seite `http.//sagenb.org` zu nutzen, existiert seit dem 17. April 2015 nicht mehr. Stattdessen wird auf der Seite `https://cloud.sagemath.com/` ebenfalls kostenlos die viel umfangreichere *Sage Cloud* angeboten. Deren Benutzung wird hier nicht beschrieben, wir empfehlen stattdessen die Installation von Sage auf Ihrem Rechner, die im folgenden für Linux-Rechner beschrieben ist. Für andere Betriebssysteme ist das Verfahren ähnlich[2], Anleitungen findet man auf der Sage-Seite.

[2]Sage ist für die meisten Betriebssysem verfügbar, Linux, Windows, Mac OS X und Solaris.

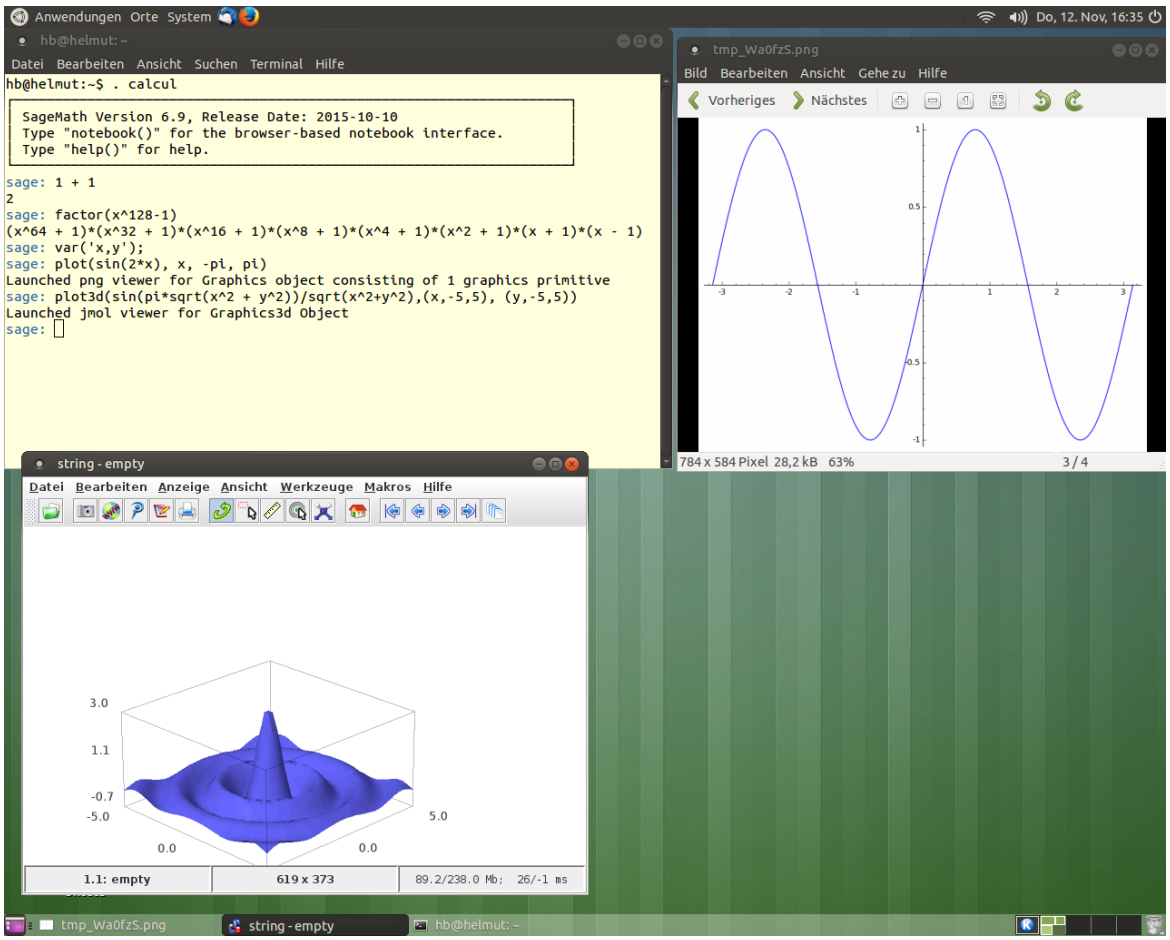


Abb. 1.3 - Sage von der Konsole

1. Rufen Sie die Seite `https://sagemath.org` auf und klicken Sie den Download-Button an.

2. Wählen Sie einen Lieferanten aus, z.B. die Freie Universität Berlin.

3. Entscheiden Sie sich für eine Rechnerarchitektur, z.B. 64 bit.

4. Laden Sie die für Ihr Betriebssystem passende Sage-Version herunter.

5. Entpacken Sie die heruntergeladene Datei in ein Verzeichnis, z.B. nach `~/bin`.

6. Richten Sie für die Dateien, die Sie im weiteren Verlauf erstellen werden, ein Arbeitsverzeichnis ein, z.B. mit `mkdir ~/SageWorkSpace`.

7. Erstellen Sie in Ihrem persönlichen Verzeichnis eine Textdatei, z.B. mit dem Namen `sage`, die zwei Zeilen enthält:

   ```
   cd ~/SageWorkSpace
   ~/bin/SageMath/sage
   ```

   Diese Zeile bewirken erstens den Wechsel in Ihr Arbeitsverzeichnis und zweitens von dort aus den Start von Sage.

8. Starten Sie nun Sage mit dem Befehl `. sage`, wobei Sie bitte auf den Punkt und das folgende Leerzeichen achten. Sage wird sich nun einrichten und durch den Prompt

   ```
   sage:
   ```

   auf der Sage-Konsole seine Bereitschaft anzeigen, Ihre Anweisungen entgegenzunehmen.

Abb. 1.4 - Die Hilfe zu den Notebooks

1.1.3. Quellen

Probieren Sie die Hinweise auf `help()` und `notebook()` bitte in aller Ruhe aus. Möglicherweise gefällt Ihnen die Arbeit mit Sage-Arbeitsblättern (worksheets) besser als mit der Sage-Konsole, doch werden wir im Buch die Notation für die Sage-Konsole verwenden. Eine Einweisung in den Umgang mit Sage-Arbeitblättern finden Sie, wenn Sie `notebook()` aufgerufen haben und dann den Menupunkt `Help` anklicken. .

Ob Sie die Sage-Konsole lieber mit weißer Schrift auf schwarzem Grund oder mit schwarzer Schrift auf hellem Grund vor sich sehen, ist Geschmackssache[3] und einstellbar.

Auf der Seite `sagemath.org` finden unter dem Button `Help/Documentation` etliche Links, wo Sie Informationen zu Sage in verschieden Sprachen - darunter auch in der deutschen - und in verschiedenen Formaten finden können.

1.2. Sage als Taschenrechner

1.2.1. Erste Rechnungen

In diesem Buch stellen wir die Rechnungen mit Sage dar in der Form

```
sage: 1+1
2
```

Die Zeichenkette `sage:` am Beginn der ersten Zeile ist die *Eingabemarke* oder der *Prompt* des Systems. Der Prompt (der in den Arbeitsblättern nicht erscheint), zeigt an, dass Sage einen Befehl des Anwenders erwartet. Dahinter ist ein Befehl einzugeben, der nach Drücken der Taste `<Enter>` ausgewertet wird. Die folgenden Zeilen enthalten die Antwort des Systems, das ist im allgemeinen das Ergebnis einer Rechnung. Manche Instruktionen erstrecken sich über mehrere Zeilen (siehe Kapitel 3). Die zusätzlichen Befehlszeilen werden dann durch den Zwischenprompt `....:` eingeleitet. Bei der Eingabe eines mehrzeiligen Befehls wird jede Zeile mit `<Enter>` abgeschlossen. Außerdem muss auf die richtige Einrückung geachtet werden (Verschiebung der Zeile mit Leerzeichen in Bezug auf die Zeile darüber). Dabei muss der Zwischenprompt am Zeilenanfang stehen bleiben.

In den Arbeitsblättern wird der Befehl direkt in die markierte Rechenzelle eingegeben. Die Eingabe wird nach Anklicken von `Evaluate` unter der Zelle oder nach Eingabe der Tastenkombination `<Shift><Enter>` ausgewertet. Die Tastenkombination `<Alt><Enter>` erzeugt eine neue Zelle unter der aktuellen und bewirkt außerdem deren Auswertung Durch Anklicken des Symbols + oberhalb oder unterhalb der aktuellen Zelle kann ebenfalls eine neue Zelle geöffnet werden.

Sage interpretiert einfache Formeln wie ein wissenschaftlicher Taschenrechner. Die Operaionen $+$, $*$ usw. haben die übliche Priorität und Klammern die gewohnte Funktion.

[3] „Über Geschmack lässt sich bekanntlich nicht streiten - schon deshalb nicht, weil es nur einen einzigen richtigen gibt, nämlich meinen.“

Arithmetische Operationen	
die vier Grundrechenarten	`a+b, a-b, a*b, a/b`
Potenz	`a^b` oder `a**b`
Quadratwurzel	`sqrt(a)`
n-te Wurzel	`a^(1/n)`

Operationen auf ganzen Zahlen	
Ganzzahldivision	`a // b`
Rest, modulo	`a % b`
Fakultät n!	`factorial(n)`
Binomialkoeffizient $\binom{n}{k}$	`binomial(n,k)`

Gebrauchliche Funktionen auf $\mathbb{R}$ und $\mathbb{C}$	
ganzzahliger Anteil	`floor(a)`
Absolutwert, Modul	`abs(a)`
elementare Funktionen	`sin, cos, ... (siehe Tabelle 2.2)`

Tab. 1.1 - Einige gebräuchliche Funktionen

```
sage: (1 + 2*(3 + 5))*2
34
```

Hier steht das Zeichen $*$ für die Multiplikation und darf nicht weggelassen werden, auch nicht in Ausdrücken wie $2x$. Die Potenz wird mit `^` oder mit $**$ notiert:

```
sage: 2^3
8
sage: 2**3
8
```

und die Division mit /

```
sage: 20/6
10/3
```

Wir sehen, dass eine exakte Rechnung erfolgt. Das Resultat der Division ist nach der Vereinfachung die rationale Zahl 10/3 und nicht ein Näherungswert wie z.B. 3.33333. Für die Größe einer ganzen oder einer rationalen Zahl gibt es keine Grenze[4]:

```
sage: 2^10
1024
sage: 2^100
1267650600228229401496703205376
sage: 2^1000
10715086071862673209484250490600018105614048117055336074437503\
88837035105112493612249319837881569585812759467291755314682518\
71452856923140435984577574698574803934567774824230985421074605\
06062371141877954182153046474983581941267398767559165543946077\
70629145711964776865421676604298316526243868372056680693761
```

[4] Außer der, die mit dem Speicher des Rechners gesetzt ist.

Um eine numerische Näherung zu erhalten, genügt es, eine der Zahlen mit Dezimalpunkt zu schreiben (man kann statt 20.0 auch 20. oder 20.000 sagen).

```
sage: 20.0/14
1.42857142857143
```

Außerdem verarbeitet die Funktion `numerical_approx` das Ergebenis einer exakten Rechnung und macht daraus eine numerische Näherung.

```
sage: numerical_approx(20/14)
1.42857142857143
```

Es ist möglich, Näherungen mit beliebig großer Genauigkeit zu berechnen Vergrößern wir sie beispielsweise auf 60 Ziffern, um die Periode der Dezimalentwicklung einer rationalen Zahl zu erkennen:

```
sage: numerical_approx(20/14, digits=60)
1.428571428571428571428571428571428571428571428571428571428571
```

Wir kehren zu den Unterschieden zwischen exakter und numerischer Rechnung im Kasten auf Seite 12 zurück.

Die Operatoren `//` und `%` ergeben den Quotienten und den Rest der Division zweier ganzer Zahlen.

```
sage: 20 // 6
3
sage: 20 % 6
2
```

Es gibt noch viele andere Funktionen auf ganzen Zahlen, von denen wir hier nur die Fakultät oder den Binomialkoeffizienten erwähnen wollen (siehe Tabelle 1.1).

```
sage: factorial(100)
 933262154439441526816992388562667004907159682643816\21
468592963895217599993229915608941463976156518286253\69
792082722375825118521091686400000000000000000000000
```

Hier nun noch eine Möglichkeit der Zerlegung einer ganzen Zahl in Primfaktoren. In Kapitel 5 greifen wir dieses Problem wieder auf und dann noch einmal in Kapitel 6.

```
sage: factor(2^(2^5)+1)
641 * 6700417
```

Fermat hatte vermutet, dass alle Zahlen der Form $2^{2^n}+1$ Primzahlen wären. Das obige Beispiel ist das kleinste, das Fermats Vermutung widerlegt.

1.2.2. Elementare Funktionen und Konstanten

Man findet die gebräuchlichen Funktionen und Konstanten, die auch mit komplexen Zahlen verwendet werden können, in den Tabellen 1.1 und 1.2. Auch da sind die Rechnungen exakt:

Wichtige Werte	
Wahrheitswerte „wahr“ und „falsch“	`True, False`
imaginäre Einheit	`I` oder `i`
unendlich ∞	`infinity` oder `oo`

Gebräuchliche Konstanten	
Kreiszahl π	`pi`
Eulersche Zahl $e = \exp(1)$	`e`
Euler-Mascheroni-Konstante γ	`euler_gamma`
Goldener Schnitt $\varphi = (1+\sqrt{5})/2$	`golden_ratio`
Catalansche Konstante	`catalan`

Tabelle 1.2 - Vordefinierte Konstanten

```
sage: sin(pi)
0
sage: tan(pi/3)
sqrt(3)
sage: arctan(1)
1/4*pi
A sage: exp(2*I*pi)
1
```

bzw. endet mit Rückgabe einer Formel statt eines Zahlenwertes:

```
sage: arccos(sin(pi/3))
arccos(1/2*sqrt(3))
sage: sqrt(2)
sqrt(2)
sage: exp(I*pi/6)
e^(1/6*I*pi)
```

Die Formeln, die man so erhält, sind nicht immer das, was man erwartet hat. Tatsächlich erfolgen nur wenige Vereinfachungen automatisch. Erscheint ein Resultat zu komplex, kann man versuchen, zur Vereinfachung explizit eine Funktion aufzurufen:

```
sage: simplify(arccos(sin(pi/e)))
 1/6*pi
sage: simplify(exp(i*pi/6))
1/2*sqrt(3) +,1/2*I
```

In Abschnitt 2.1 werden wir sehen, wie man die Vereinfachung von Ausdrücken noch besser steuern kann. Natürlich kann man auch numerische Näherungen der Ergebnisse berechnen (am häufigsten mit einer Genauigkeit, die man vorschreibt):

```
sage: numerical_approx(6*arccos(sin(pi/3)), digits=60)
3.14159265358979323846264338327950288419716939937510582097494
```

```
sage: numerical_approx(sqrt(2), digits=60)
1.41421356237309504880168872420969807856967187537694807317668
```

Symbolisches Rechnen und numerische Methoden

Ein Computer-Algebra-System (CAS) zum symbolischen Rechnen ist eine Software, deren Zweck die Bearbeitung, die Vereinfachung und die Berechnung mathematischer Formeln einzig durch exakte Umformungen ist. Der Terminus *symbolisch* steht hier als Gegensatz zu *numerisch*. Er besagt, dass die Rechnungen auf symbolischen Ausdrücken stattfinden und zwar algebraisch. Im Englischen sagt man *computer algebra* oder *symbolic computation*.
Die Taschenrechner verarbeiten ganze Zahlen mit weniger als einem Dutzend Stellen exakt, größere Zahlen werden gerundet, was zu Fehlern führt. So wertet ein Taschenrechner den folgenden Ausdruck falsch aus und liefert 0 anstatt 1:

$$(1 + 10^{50})10^{50}$$

Solche Fehler sind schwer zu finden, wenn sie durch keine theoretische Untersuchung vorhergesehen - und auch nicht leicht vorhersehbar - bei einer Zwischenrechnung auftreten. Die Systeme zum symbolischen Rechnen hingegen verschieben diese Grenzen und führen bei den ganzen Zahlen keine Rundungen durch, damit die Rechnungen in ihrem ganzen Verlauf exakt bleiben: sie beantworten die obige Rechnung mit 1.
Die Verfahren zur numerischen Analysis nähern das Integral $\int_0^\pi \cos t dt$ mit einer vorgegebenen Genauigkeit an (Trapezregel, Simpson, Gauß) um zum Beispiel ein numerisches Ergebnis zu erhalten, das mehr oder weniger dicht an 0 liegt (beispielsweise 10^{-10}), ohne sagen zu können, dass das Resultat exakt die ganze Zahl 0 ist - oder auch, dass es fast 0 ist aber nicht genau 0.
Ein CAS formt das Integral $\int_0^\pi \cos t dt$ durch eine Manipulation der mathematischen Symbole in die Formel $\sin \pi - \sin 0$ um, die dann exakt zu $0 - 0$ ausgewertet wird. Diese Methode beweist deshalb $\int_0^\pi \cos t dt = 0$.
Dennoch haben auch die nur algebraischen Umformumgen ihre Grenzen. Die meisten der von einem CAS verarbeiteten Ausdrücke sind Quotienten von Polynomen und ein Ausdruck a/a wird automatisch zu 1 vereinfacht. Solche algebraische Rechnung eignet sich nicht zum Lösen von Gleichungen; in diesem Rahmen hat die Gleichung $ax = a$ die Lösung $x = a/a$, was zu $x = 1$ vereinfacht wird ohne zu bemerken, dass für $a = 0$ jede Zahl x Lösung der Gleichung ist.

1.2.3. Online-Hilfe und automatische Vervollständigung

Interaktiv kann man auf das Referenz-Handbuch zugreifen und jede Funktion, jede Konstante oder jeden Befehl nachschlagen, indem man dahinter ein Fragezeichen schreibt

```
sage: sin?
```

Die Seite der Dokumentation (auf englisch) enthält die Beschreibung der Funktion und Anwendungsbeispiele.

Die Tabulator-Taste <TAB> nach einem Wortanfang zeigt alle Befehle, die so anfangen: arc gefolgt von <TAB> zeigt die Namen sämtlicher Arkus- und Areafunktionen:

```
sage: arc
arc      arccosh  arccoth  arccsch  arcsech  arcsinh  arctan2
arccos   arccot   arccsc   arcsec   arcsin   arctan   arctanh
```

1.2.4. Python-Variablen

Sobald man das Resultat einer Rechnung speichern möchte, kann man es einer *Variablen zuweisen:*

```
sage: y = 1 + 2
```

um es später weiterzuverwenden:

```
sage: y
3
sage: (2 + y) * y
15
```

Zu beachten ist, dass das Resultat nach einer Zuweisung nicht automatisch ausgegeben wird. Oft werden wir auch die folgende Akürzung verwenden:

```
sage: y = 1 + 2; y
3
```

Das Semikolon ; trennt mehrere Anweisungen auf einer einzigen Zeile. Die Berechnung des Ergebnisses erfolgt vor der Zuweisung. Daher kann man dieselbe Variable weiterverwenden:

```
sage: y = 3 * y + 1; y
10
sage: y = 3 * y + 1; y
31
sage: y = 3 * y + 1; y
94
```

Schließlich speichert Sage die Ergebnisse der letzten drei Rechnungen in den Sondervariablen _, __ und ___:

```
sage: 1 + 1
2
sage: _ + 1
 3
sage: __
2
```

Die Variablen, die wir gerade bearbeitet haben, sind Variablen aus der Programmierung in Python, und wir werden das im Unterabschnitt 3.1.3 noch einmal genauer behandeln. Hier wollen wir nur vermerken, dass es empehlenswert ist, weder Funktionen noch in Sage vordefinierte Konstanten zu überschreiben. Das würde den internen Ablauf von Sage zwar nicht beeinträchtigen, doch die nachfolgenden Ergebnisse Ihrer Rechnungen oder Iher Programme könnten unsicher werden.:

```
sage: pi = -I/2
sage: exp(2*i*pi)
 e
```

Zur Wiederherstellung des Ausgangswertes kann ein Befehl verwendet werden wie: So sind die von Sage behandelten Gebiete der Mathematik recht vielfältig, von der Gruppentheorie bis zur numerischen Analysis. Sage kann die erhaltenen Ergebnisse auch als ebene oder räumlichelder und als Animationen darstellen.

```
sage: from sage.all import pi
```

1.2.5. Symbolische Variablen

Bis jetzt haben wir nur konstante Ausdrücke wie $\sin(\sqrt{2})$ verarbeitet. Sage ermöglicht auch und vor allem mit Ausdrücken zu rechnen, die Variablen enthalten wie $x + y = z$ oder auch $\sin(x) + \cos(x)$. Die *symbolischen Variablen* des „Mathematikers“, x, y, z, erscheinen in Sage in verschiedenen Ausdrücken, Variablen des „Programmierers“, denen wir schon im vorigen Abschnitt begegnet sind. Sage unterscheidet sich insbesondere in diesem Punkt von anderen CAS wie Maple oder Maxima.

Die symbolischen Variablen müssen vor ihrer Verwendung explizit deklariert werden[5]:

```
sage: z = SR.var('z')
sage: 2*z + 3
2*z + 3
```
[6]

In diesem Beispiel „konstruiert“ der Befehl `var('z')` eine symbolische Variable mit dem Namen z. Diese symbolische Variable ist ein Sage-Objekt wie andere auch: sie wird nicht anders behandelt als komplexere Ausdrücke wie $\sin(x)+1$. Dann wird diese symbolische Variable der Variablen `z` „des Programmiers“ zugeordnet, was ermöglicht, sich ihrer zur Bildung beliebig komplexer Ausdrücke zu bedienen.

Wir hätten z auch einer anderen Variablen als `z` zuordnen können:

```
sage: y = SR.var('z')
sage: 2*y + 3
2*z + 3
```

Die systematische Bindung der symbolischen Variablen z an die Variable `z` ist daher keine bloße Konvention, sondern sie wird zur Vermeidung von Verwirrung empfohlen.

Andererseits hat die Variable `z` mit der symbolischen Variablen z nichts zu tun:

```
sage: c = 2*y + 3
sage: z = 1
sage: 2*y + 3
2*z + 3
sage: c
2*z + 3
```

[5] Tatsächlich ist die symbolische Variable x in Sage vordefiniert - das ist aber auch die einzige.
[6] `SR` steht für `Symbolic Ring`, siehe dazu auch Kapitel 5.

Wie weist man einer symbolischen Variablen nun aber einen Wert zu, der in einem Ausdruck erscheint? Man *substituiert*, wie in

```
sage: x = SR.var('x')
sage: expr = sin(x); expr
sin(x)
sage: expr(x=1)
sin(1)
```

Ausführlich wird die Substitution in symbolischen Ausdrücken in Kapitel 2 behandelt.

Übung 1. Erläutern Sie Schritt für Schritt, was bei dieser Anweisungsfolge vor sich geht:

```
u = SR.var('u')
 u = u+1
u = u+1
u
```

Da es ein wenig arbeitsaufwendig ist, auf diese Weise eine große Zahl von symbolischen Variablen zu generieren, gibt es eine Kurzform `var('x')`, die zu `x = SR.var('x')` äquivalent ist. Beispielsweise kann man schreiben

```
sage: var('a, b, c, x, y')
(a, b, c, x, y)
sage: a*x + b*y + c
a*x + b*y + c
```

Wird die explizite Variablendeklaration als zu schwerfällig eingeschätzt, kann das Verhalten von Maxima oder Maple emuliert werden. Im Moment (Sage 7.6) steht diese Funktionalität nur im Notebook zur Verfügung. Dort führt nach dem Aufruf von

```
sage: automatic_names(True)
```

jeder Gebrauch einer neuen Variablen implizit zur Erzeugung einer symbolischen Variablen gleichen Namens und ihre Zuweisung.

```
sage: 2*bla + 3
2*bla + 3
sage: bla
bla
```

1.2.6. Erste Grafiken

Der Befehl `plot` ermöglicht auf ganz einfache Weise das Zeichnen des Graphen einer reellen Funktion in einem gegebenen Intervall. Der Befehl `plot3d` dient zum Zeichnen in drei Dimensionen, wie den Graphen einer reellen Funktion von zwei Veränderlichen. Hier die Befehle, mit denen die Grafiken in Abb. 1.3 (S. 6) erzeugt wurden:

```
sage: plot(sin(2*x), x, -pi, pi)
sage: plot3d(sin(pi*sqrt(x^2+y^2))/sqrt(x^v2+y^2), (x,-5,5), (y,-5,5))
```

Die grafischen Möglichkeiten von Sage zeigen sich auch an anderen Beispielen. Ausführliche Erklärungen folgen im 4. Kapitel.

2. Analysis und Algebra

Dieses Kapitel stellt anhand einfacher Beispiele die Grundfunktionen vor, die in Analysis und Algebra von Nutzen sind. Schüler und Studenten werden hier Material finden, um *Bleistift und Papier* durch *Tastatur und Bildschirm* zu ersetzen, und damit für das Verstehen von Mathematik vor der gleichen intellektuellen Herausforderung stehen.

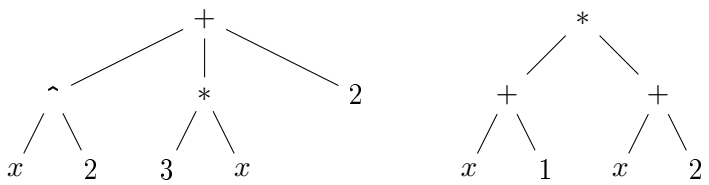
Diese Auflistung der gebräuchlichsten Befehle für das Rechnen mit Sage will auch für Oberstufenschüler zugänglich erscheinen. Mit einem Stern (∗) gekennzeichnet, bilden sie Ergänzungen für Studenten im ersten Studienjahr. Wegen weiterer Einzelheiten verweisen wir auf die anderen Kapitel.

2.1. Symbolische Ausdrücke und Vereinfachungen

2.1.1. Symbolische Ausdrücke

Sage ermöglicht alle Arten von Rechnungen zur Analysis mit *symbolischen Ausdrücken* und Zahlen auszuführen, mit symbolischen Variablen, den vier Grundrechenarten und den üblichen Funktionen wie `sqrt, exp, log, sin, cos` etc. Ein symbolischer Ausdruck kann durch einen Baum dargestellt werden, wie in Abb. 2.1. Es ist wichtig zu verstehen, dass ein symbolischer Ausdruck eine *Formel* ist und kein Wert oder eine mathematische Funktion. So erkennt Sage nicht, dass die folgenden Ausdrücke gleich sind[1]:

```
sage: bool(arctan(1+abs(x)) == pi/2 - arctan(1/(1+abs(x))))
 False
```



`x^2 + 3*x + 2` `(x + 1)*(x + 2)`

Abb. 2.1 - Zwei symbolische Ausdrücke, die dasselbe mathematische Objekt darstellen.

[1] Der Test auf Gleichheit == ist insofern kein einfacher syntaktischer Vergleich von Formeln: beispielsweise *werden* in Sage 7.6 die Ausdrücke `arctan(sqrt(2))` und `pi/2 - arctan(1/sqrt(2))` als gleich betrachtet. Wenn man zwei Ausdrücke mit `bool(x==y)` vergleicht, versucht Sage zu beweisen, dass ihre Differenz null ist und gibt `True` zurück, wenn das gelingt.

Dank der in diesem Kapitel vorgestellten Befehle kann der Anwender Ausdrücke umformen und sie in die gewünschte Form bringen.

Die wohl häufigste Operation ist das *Auswerten* eines Ausdrucks, indem einem oder mehreren der darin vorkommenden Parameter ein Wert zugewiesen wird. Das bewirkt die - vielleicht nicht immer ausdrücklich angegebene - Methode `subs`.

```
sage: a, x = var('a, x'); y = cos(x+a) * (x+1); y
(x + 1)*cos(a + x)
sage: y.subs(a=-x); y.subs(x=pi/2, a=pi/3); y.subs(x=0.5, a=2.3)
x + 1
-1/4*sqrt(3)*(pi + 2)
-1.41333351100299
sage: y(a=-x); y(x=pi/2, a=pi/3); y(x=0.5, a=2.3)
x + 1
-1/4*sqrt(3)*(pi + 2)
-1.41333351100299
```

Wie auch in der üblichen Schreibweise $x \mapsto f(x)$ muss der Name der ersetzten Variablen angegeben werden. Die Substitution mehrerer Variablen geschieht auf analoge Weise, indem alle Ersetzungen nacheinander hingeschrieben werden, wie die beiden nächsten Beispiele zeigen:

```
sage: x, y, z = var('x, y, z') ; q = x*y + y*z + z*x
sage: bool(q(x=y, y=z, z=x) == q), bool(q(z=y)(y=x) == 3*x^2)
(True, True)
```

Um einen Unterausdruck zu ersetzen, der komplizierter ist als eine Variable, greift man zur Funktion `substitute`.

```
sage: y, z = var('y, z'); f = x^3 + y^2 + z
sage: f.substitute(x^3 == y^2, z==1)
2*y^2 + 1
```

2.1.2. Umformung von Ausdrücken

Die einfachsten Ausdrücke „mit Variablen“ sind die Polynome und Quotienten aus zwei Polynomen mit einer oder mehreren Variablen. Die Funktionen, die erlauben, sie auf verschiedene Weise umzuschreiben oder in eine Normalform zu bringen, sind in Tabelle 2.1 aufgeführt. Zum Beispiel dient die Funktion `expand` dazu, Polynome auszumultiplizieren.

```
sage: x, y = SR.var('x,y')
sage: p = (x+y)*(x+1)^2
sage: p2 = p.expand(); p2
x^3 + x^2*y + 2*x^2 + 2*x*y + x + y
```

während die Methode `collect` die folgenden Terme nach fallenden Potenzen einer gegebenen Variablen umordnet:

```
sage: p2.collect(x)
x^3 + x^2*(y + 2) + x*(2*y + 1) + y
```

Diese Funktionen werden auch auf Ausdrücke angewendet, die Polynome sind, aber keine mit symbolischen Variablen sondern mit Unterausdrücken, die komplizierter sind als `sin(x)`:

```
sage: ((x+y+sin(x))^2.expand().collect(sin(x))
2*(x + y)*sin(x) + x^2 + 2*x*y + y^2 + sin(x)^2
```

Symbolische Funktionen

Sage erlaubt auch die Definition von *symbolischen Funktionen* zur Verarbeitung eines Ausdrucks.

```
sage: f(x) = (2*x + 1)^3; f(-3)
-125
sage: f.expand()
x |--> 8*x^3 + 12*x^2 + 6*x + 1
```

Eine symbolische Funktion ist nichts anderes als ein Ausdruck, den man wie einen Befehl aufrufen kann und bei dem die Reihenfolge der Variablen festgelegt ist. Um einen symbolischen Ausdruck in eine symbolische Funktion zu verwandeln, benutzt man entweder die schon erwähnte Syntax oder die Methode `function`:

```
sage: y = var('y'); u = sin(x) + x*cos(y)
sage: v = u.function(x, y); v
(x, y) |--> x*cos(y) + sin(x)
sage: w(x, y) = u; w
(x, y) |--> x*cos(y) + sin(x)
```

Die symbolischen Funktionen dienen zur Modellierung mathematischer Funktionen. Sie spielen nicht die gleiche Rolle wie die Funktionen (oder *Prozeduren*) in Python. Das sind Konstruktionen der Programmierung, die wir im 3. Kapitel beschreiben. Der Unterschied zwischen beiden entspricht dem Unterschied zwischen symbolischen Variablen und den in Unterabschnitt 1.2.4 vorgestellten Python-Variablen.

Praktisch verwendet man eine symbolische Funktion genauso wie einen Ausdruck, was bei Python-Funktionen nicht der Fall ist, beispielsweise verfügen letztere nicht über die Methode `expand`.

Polynom $p = zx^2 + x^2 - (x^2 + y^2)(ax - 2by) + zy^2 + y^2$	
`p.expand().collect(x)`	$-ax^3 - axy^2 + 2byy^3 + (2by + z + 1)x^2 + y^2z + y^2$
`p.collect(x).collect(y)`	$2bx^2y + 2by^3 - (ax - z - 1)x^2 - (ax - z - 1)y^2$
`p.expand()`	$-ax^3 - axy^2 + 2bx^2y + 2by^3 + x^2z + y^2z + x^2 + y^2$
`p.factor()`	$-(x^2 + y^2)(ax - 2by - 7 - 1)$
`p.factor_list`	$[(x^2 + y^2, 1), (ax - 2by - z - 1, 1), (-1, 1)]$

Quotient von Polynomen $r = \dfrac{x^3 + x^2y + 3x^2 + 3xy + 2x + 2y}{x^3 + 2x^2 + xy + 2y}$	
`r.simplify_rational()`	$\dfrac{(x+1)y + x^2 + x}{x^2 + y}$
`r.factor()`	$\dfrac{(x+1)(x+y)}{x^2 + y}$
`r.factor().expand()`	$\dfrac{x^2}{x^2 + y} + \dfrac{xy}{x^2 + y} + \dfrac{x}{x^2 + y} + \dfrac{y}{x^2 + y}$

Quotient von Polynomen $r = \dfrac{(x-1)x}{x^2 - 7} + \dfrac{y^2}{x^2 - 7} + \dfrac{b}{a} + \dfrac{c}{a} + \dfrac{1}{x+1}$	
`r.combine()`	$\dfrac{(x-1)x + y^2}{x^2 - 7} + \dfrac{b+c}{a} + \dfrac{1}{x+1}$

Quotient von Polynomen $r = \dfrac{1}{(x^3 + 1)y^2}$	
`r.partial_fraction(x)`	$-\dfrac{(x-2)}{3(x^2 - x + 1)y^2} + \dfrac{1}{3(x+1)y^3}$

Tab. 2.1 - Polynome und Quotienten von Polynomen[2]

Was Quotienten von Polynomen angeht, d.h. gebrochen rationale Ausdrücke, so erlaubt die Funktion `combine` die Umgruppierung von Termen mit gleichem Nenner, während die Funktion `partial_fraction` die Zerlegung in einfache Elemente in $\mathbb{Q}$ bewirkt (Eine genauere Beschreibung dieses Körpers, in dem die Zerlegung in einfache Elemente zu erfolgen hat, wird man in Abschnitt 7.4 finden.)

Die gebräuchlichsten Darstellungen sind für das Polynom die ausmultiplizierte Form und für den Quotienten zweier Polynome die gekürzte Form P/Q mit ausmultiplizierten P und Q. Sobald zwei Polynome oder zwei Quotienten von Polynomen in diesen Formen geschrieben sind, genügt für die Entscheidung, ob sie gleich sind, der Vergleich ihrer Koeffizienten: man nennt diese Formen die *Normalformen.*

2.1.3. Gebräuchliche mathematische Funktionen

In Sage liegen schon die meisten mathematischen Funktionen vor. insbesondere die trigonometrischen Funktionen, die Logarithmus- und die Exponentialfunktion: In Tab. 2.2 sind sie zusammengestellt. Die Vereinfachung dieser Funktionen ist wesentlich. Um einen Aussdruck oder eine symbolische Funktion zu vereinfachen, verfügen wir über die Funktion `simplify`:

Gebräuchliche mathematische Funktionen	
Exponentialfunktion und Logarithmus	`exp, log`
Logarithmus zur Basis a	`log(x,a)`
trigonometrische Funktionen	`sin, cos, tan, cot, sec, csc`
Arcusfunktionen	`arcsin, arccos, arctan, arccot`
hyperbolische Funktionen	`sinh, cosh, tanh, coth`
Areafunktionen	`arcsinh, arccosh, arctanh, arccoth`
ganzzahliger Anteil usw.	`floor, ceil, trunc, round`
Quadratwurzel und n–te Wurzel	`sqrt, nth_root`

Umformung trigonometricher Ausdrücke	
Vereinfachung	`simplify_trig`
Linearisierung	`reduce_trig`
Anti_Linearisierung	`expand_trig`

Tab. 2.2 - Gebräuchliche Funktionen und Vereinfachungen

```
sage: (x^x/x).simplify()
x^(x-1)
```

Jedoch muss man für feinere Vereinfachungen den Typ der erwarteten Vereinfachung angeben:

```
sage: f = (e^x-1) / (1+e^(x/2)); f.simplify_exp()
e^(1/2*x) - 1
```

Entsprechend nimmt man den Befehl `simplify_trig` zur Vereinfachung trigonometrischer Ausdrücke.

[2] Die Darstellung der Ausdrücke auf der rechten Seite erhält man in einem Sage-Arbeitsblatt nach Aktivierung von `Typeset`.

```
sage: f = cos(x)^6 + sin(x)^6 + 3*sin(x)^2*cos(x)^2
sage: f.simplify_trig()
1
```

Zur Linearisierung (bzw. Anti-Linearisierung) eines trigonometrischen Ausdrucks verwendet man `reduce_trig` (bzw. `expand_trig`):

```
sage: f = cos(x)^6; f.reduce_trig()
1/32*cos(6*x) + 3/16*cos(4*x) + 15/32*cos(2*x) + 5/16
sage: f = sin(6*x): f.expand_trig()
6*cos(x)^5*sin(x) - 20*cos(x)^3*sin(x)^3 + 6*cos(x)*sin(x)^5
```

Man kann Ausdrücke mit Fakultäten ebenfalls vereinfachen:

```
sage: n = var('n'); f = factorial(n+1)/factoral(n)
n+1
```

Was die Funktion `simplify_rational` angeht, so versucht sie, einen Quotienten von Polynomen dadurch zu vereinfachen, dass sie seine Glieder ausmultipliziert. Für die Vereinfachung von Quadratwurzeln, logarithmischen oder exponentiellen Ausdrücken steht die Funktion `canonicalize_radical` zur Verfügung.

```
sage: f = sqrt(abs(x)^2); f.canonicalize_radical()
x
sage: y = var('y')
sage: f = log(x*y); f.canonicalize_radical()
log(x) + log(y)
```

Der Befehl `simplify_full` wendet die Funktionen `simplify_factorial`, `simplify_ trig` und `simplify_rational` an (in dieser Reihenfolge).

Alles, was zur klassischen Palette einer Kurvendiskussion gehört (Berechnung von Ableitungen, Asymptoten, Extrema, Nullstellen und Skizze des Graphen), kann mit einem CAS leicht realisiert werden. Sages wichtigste Operationen die man auf Funktionen anwenden kann, werden in Abschnitt 2.3 vorgestellt.

2.1.4. Vorgaben für eine symbolische Variable

Bei der Rechnung mit symbolischen Variablen wird im allgemeinen angenommen, dass sie jeden Wert einer reellen oder komplexen Zahl annehmen können. Das führt zu Problemen, wenn ein Parameter eine auf ihren Definitionsbereich beschränkte Größe darstellt (beispielsweise positiv reell).

Ein typischer Fall ist der Ausdruck $\sqrt{x^2}$. Zur Vereinfachung solcher Ausdrücke bietet die Funktion `assume`, mit der die Eigenschaften einer Variablen päzisiert werden können, eine gute Lösung. Möchte man diese Vorgabe wieder aufheben, kommt der Befehl `forget` zum Einsatz:

```
sage: assume(x>0); bool(sqrt(x^2) == x)
True
sage: forget(x>0); bool(sqrt(x^2) == x)
False
```

```
sage: n = var('n'); assume(n, 'integer'); sage: sin(n*p).simplify()
0
```

2.1.5. Mögliche Gefahren

Sei c ein ganz klein wenig komplizierter Ausdruck:

```
sage: a = var('a')
sage: c = (1+a)^2 - (a^2+2*ab+1)
```

und versuchen wir, die in x gegebene Gleichung $cx = 0$ nach x aufzulösen:

Das Problem der Vereinfachung

Die Beispiele in Unterabschnitt 2.5.1 illustrieren die Wichtigkeit der Normalformen und spziell des *Tests auf 0*, ohne den jede Rechnung, die eine Division enthält, zum Glücksspiel wird.
Bestimmte Familien von Ausdrücken, die aussehen wie Polynome, lassen eine Prozedur zu, die entscheidet, ob der Nenner 0 ist. Das heißt, dass für diese Klassen von Ausdrücken ein Programm entscheiden kann, ob ein gegebener Ausdruck null ist oder nicht. In vielen Fällen wird diese Entscheidung durch Reduktion auf die Normalform getroffen: der Ausdruck ist null dann und nur dann, wenn wenn seine Normalform 0 ist.
Unglücklicherweise können nicht alle Klassen von Ausdrücken eine Normalform bilden, und für einige Klassen kann man zeigen, dass es keine allgemeine Methode gibt, die in endlicher Zeit feststellt, ob ein Ausdruck null ist. Ein Beispiel einer solchen Klasse ist ein Ausdruck, der aus rationalen Zahlen, π, $\ln 2$ und einer Variablen zusammengesetzt ist mit wiederholter Addition, Subraktion, Multiplikation, Exponentiation und der Sinusfunktion. Eine wiederholte Anwendung von `numerical_approx` mit erhöhter Genauigkeit erlaubt häufig eine Vermutung, ob ein spezieller Ausdruck null ist oder nicht; doch es ist bewiesen, dass es nicht möglich ist, ein Programm zu schreiben, das als Argument einen Ausdruck dieser Klasse erhält und das wahre Ergebnis zurückgibt, wenn das Argment null ist und andernfalls ein falsches.
Bei diesen Klassen stellt sich also das Problem der Vereinfachung mit größter Schärfe. Ohne Normalform können die Systeme nur eine bestimmte Zahl von Umformungen angeben, mit denen der Anwender jonglieren muss, um zu einem Ergebnis zu kommen. Um hier klarer zu sehen, muss man die Unterklassen der Ausdrücke identifizieren, die Normalformen haben und wissen, welche Funktionen aufzurufen sind, um letztere zu berechnen. Sages Verhalten gegenüber diesen Schwierigkeiten wird in Kapitel 5 ausführlicher behandelt.

```
sage: eq = c*x == 0
x == 0
```

Ein unvorsichtiger Anwender wird versucht sein, diese Gleichung zu vereinfachen, bevor er sie löst:

```
sage: eq2 = eq/c; eq2
x == 0
```

```
sage: solve(eq2, x)
[x == 0]
```

Glücklicherweise macht Sage diesen Fehler nicht:

```
sage: solve(eq, x)
[x == x]
```

Hier konnte Sage die Aufgabe korrekt lösen, denn der Koeffizient c ist ein polynomialer Ausdruck. Daher kann c leicht auf 0 getestet werden. Es reicht hin, c auszumultiplizieren:

```
sage: expand(c)
0
```

und davon Gebrauch zu machen, dass zwei Polynome mit gleicher ausmultiplizierter Form gleich sind, anders gesagt, dass die ausmultiplierte Form eine Normalform ist.

Im Gegenzug begeht Sage bei einem kaum komplizierteren Ausdruck einen Fehler:

```
sage: c = cos(a)^2 + sin(a)^2 - 1
sage: eq = c*x == 0
sage: solve(eq, x)
[x == 0]
```

und das, obwohl die Vereinfachung und der Test auf 0 korrekt sind:

```
sage: c.simplify_trig()
0
sage: c.is_zero
True
```

2.2. Gleichungen

Wir wenden uns jetzt den Gleichungen zu und ihrer Lösung; die wichtigsten Funktionen dazu sind in Tab. 2.3 zusammengestellt.

Numerische Gleichungen	
symbolische Lösung	`solve`
Lösunmg (mit Mehrfachwurzeln)	`roots`
numerische Lösung	`find_root`
Vektorgleichungen und Funktionsgleichungen	
Lösung linearer Gleichungen	`right_solve, left_solve`
Lösung von Differentialgleichungen	`desolve`
Lösung von Rekursionen	`rsolve`

Tab. 2.3 - Lösung von Gleichungen

2.2.1. Explizite Lösung

Wie betrachten folgende Gleichung mit der Unbekannten z und dem Parameter φ:

$$z^2 - \frac{2}{\cos\varphi}z + \frac{5}{\cos^2\varphi} - 4 = 0, \quad \text{mit } \varphi \in \left]-\frac{\pi}{2}, \frac{\pi}{2}\right[.$$

Wir schreiben

```
sage: z, phi = var('z, phi')
sage: eq = z**2 - 2/cos(phi)*z + 5/cos(phi)**2 - 4 == 0; eq
z^2 - 2*z/cos(phi) + 5/cos(phi)^2 - 4 == 0
```

Mit der Methode `lhs` (bzw. mit der Methode `rhs`) kann man die linke Seite der Gleichung (bzw. die rechte) isolieren:

```
sage: eq.lhs()
z**2 - 2/cos(phi)*z + 5/cos(phi)**2 - 4
sage: eq.rhs()
0
```

was jedoch nicht erforderlich ist, um sie mit `solve` zu lösen. In einem Sage-Arbeitsblatt sieht das so aus:

```
solve(eq,z)
```

$$\left[z = -\frac{2\sqrt{\cos(\varphi)^2 - 1} - 1}{\cos(\varphi)}, z = \frac{2\sqrt{\cos(\varphi)^2 - 1} + 1}{\cos(\varphi)}\right]$$

Sei nun die Gleichung $y^7 = y$ zu lösen.

```
sage: y = var('y'); solve(y^7==y, y)
[y == 1/2*I*sqrt(3) + 1/2, y == 1/2*I*sqrt(3) - 1/2, y == -1, y ==
-1/2*I*sqrt(3) - 1/2, y == -1/2*I*sqrt(3) + 1/2, y == 1, y == 0]
```

Die Lösungen können auch in Gestalt eines Objekts vom Typ Diktionär[3] zurückgegeben werden (siehe Unterabschnitt 3.3.9).

```
sage: solve(x^2-1, x, solution_dict=True)
[{x: -1}, {x: 1}]
```

Der Befehl `solve` erlaubt ebenfalls das Lösen von Gleichungssystemen:

```
sage: solve({x+y == 3, 2*x+2*y == 6], x, y)
[[x == -r1 + 3, y == r1]]
```

Bei unbestimmten Systemen wie diesem übernehmen Hilfsvariablen, die mit `r1, r2` usw. bezeichnet werden, die Parametrisierung der Lösungsmenge. Kommen als Parameter nur ganze Zahlen in Betracht, werden sie mit `z1, z2` usw. angegeben ((hierunter ist es `z38`, was je nach Sage-Version auch anders sein kann).

```
sage: solve([cos(x)*sin(x) == 1/2, x+y == 0], x, y)
[[x == 1/4*pi + pi*z38, y == -1/4*pi - pi*z38]]
```

[3] engl. *dictionary*

Schließlich kann man mit der Funktion `solve` sogar Ungleichungen lösen:

```
sage: solve(x^2+x-1 > 0, x)
[[x < -1/2*sqrt(5) - 1/2], [x > 1/2*sqrt(5) - 1/2]]
```

Es kommt vor, dass von `solve` die Lösungen eines Gleichungssystems als Fließkommazahlen zurückgegeben werden. Sei beispielsweise das folgende Gleichungssystem im $\mathbb{C}^2$ zu lösen:

$$\begin{cases} x^2yz &= 18, \\ xy^3z &= 24, \\ xyz^3 &= 3. \end{cases}$$

```
sage: x, y, z = var('x, y, z')
sage: solve([x^2*y*z == 18, x*y^3*z == 24,\
....: x*y*z^4 == 3], x, y, z)
[[x = (-2.76736473308 - 1.71347969911*I), y = (-0.570103503963 +
    2.00370597877*I), z = (-0.801684337646 - 0.14986077496*I)],...]
```

Sage gibt hier 17 angenäherte komplexe Lösungstripel zurück. Wegen symbolischer Lösungen befrage man das 9. Kapitel.

Um die numerische Lösung einer Gleichung zu erhalten, verwendet man die Funktion `find_root`, die als Argument eine Funktion einer Veränderlichen oder eine symbolische Gleichung sowie die Grenzen des Intervalls aufnimmt, in welchem die Lösung gesucht werden soll.

```
sage: expr = sin(x) + sin(2*x) + sin(3*x)
sage: solve(expr, x)
[sin(3*x) == -sin(2*x) - sin(x)]
```

Zu dieser Gleichung findet Sage keine symbolische Lösung. Nun gibt es zwei Möglichkeiten: entweder greift man zu einer numerischen Lösung

```
sage: find_root(expr,0.1,pi)
2.0943951023931957
```

oder man formt den Ausdruck vorher um:

```
sage: f = f = expr.simplify_trig(); f
2*(cos(x)^2 + cos(x))*sin(x)
sage: solve(f, x)
[x == 0, x == 2/3*pi, x == 1/2*pi]
```

Schließlich kann man mit der Funktion `roots` die exakten Lösungen samt deren Vielfachheit bekommen. Man kann überdies den Ring angeben, in welchem man die Lösung zu erhalten wünscht; wählt man `RR` für $\mathbb{R}$ oder `CC` für $\mathbb{C}$, erhält man die Ergebnisse als Fließkommazahlen. Das untenstehende Lösungsverfahren ist für die betrachtete Gleichung spezifisch, im Gegensatz zu `find_root`, das eine generische Methode verwendet.

Betrachten wir die Gleichung dritten Grades $x^3 + 2x + 1 = 0$. Diese Gleichung hat eine negative Diskriminante und besitzt somit eine reelle Wurzel und zwei komplexe, die man mit der Funktion `roots` erhalten kann (dargestellt im Sage-Arbeitsblatt):

```
(x^3+2*x+1).roots(x)
```

$$\left[\left(-\frac{1}{2}\left(\frac{1}{18}\sqrt{59}\sqrt{3}-\frac{1}{2}\right)^{\frac{1}{3}}\left(i\sqrt{3}+1\right)-\frac{i\sqrt{3}-1}{3\left(\frac{1}{18}\sqrt{59}\sqrt{3}-\frac{1}{2}\right)^{\frac{1}{3}}},1\right),\right.$$
$$\left(-\frac{1}{2}\left(\frac{1}{18}\sqrt{59}\sqrt{3}-\frac{1}{2}\right)^{\frac{1}{3}}\left(-i\sqrt{3}+1\right)-\frac{-i\sqrt{3}-1}{3\left(\frac{1}{18}\sqrt{59}\sqrt{3}-\frac{1}{2}\right)^{\frac{1}{3}}},1\right),$$
$$\left.\left(\left(\frac{1}{18}\sqrt{59}\sqrt{3}-\frac{1}{2}\right)^{\frac{1}{3}}-\frac{2}{3\left(\frac{1}{18}\sqrt{59}\sqrt{3}-\frac{1}{2}\right)^{\frac{1}{3}}},1\right)\right]$$

```
sage: (x^3+2*x+1).roots(x, ring=RR)
[(-0.453397651516404, 1)]
sage: (x^3+2*x+1).roots(x, ring=CC)
[(-0.453397651516404, 1),
 (0.226698825758202 - 1.46771150871022*I, 1),
 (0.226698825758202 + 1.46771150871022*I, 1)]
```

2.2.2. Gleichungen ohne explizite Lösung

Sobald das Gleichungssystem zu kompliziert wird, ist es in den wenigsten Fällen möglich, eine exakte Lösung zu berechnen.

```
sage: solve(x^(1/x)==(1/x)^x, x)
[(1/x)^x == x^(1/x)]
```

Andererseits ist das nicht zwangsläufig eine Beschränkung, wie man meinen könnte. Tatsächlich ist ein Leitmotiv des symbolischen Rechnens, dass man durch Gleichungen definierte Objekte sehr wohl umformen und insbesonerere ihre Eigenschaften berechnen kann, ohne ihre explizite Lösung zu kennen. Noch besser, die Gleichung, die ein Objekt definiert, ist oft die beste Beschreibung dieses Objekts. So ist eine Funktion, die durch eine lineare Differentialgleichung und Anfangsbedingungen definiert ist, vollkommen bestimmt. Die Lösungsmenge von linearen Differentialgleichungen ist (unter anderem) hinsichtlich Summe und Produkt abgeschlossen und bildet so eine wichtige Klasse, wo man den Test auf 0 einfach vornehmen kann. Dafür fällt die Lösung, ist sie ihrer Definitionsgleichung beraubt, in eine größere Klasse, wo nur wenig entschieden werden kann:

```
sage: y = function('y', x)
sage: desolve(diff(y,x,x) + x*diff(y,x) + y == 0, y, [0,0,1])
-1/2*I*sqrt(2)*sqrt(pi)*erf(1/2*I*sqrt(2)*x)*e^(-1/2*x^2)
```

Im 14. Kapitel und im Unterabschnitt 15.1.2 greifen wir diese Überlegungen wieder auf.

2.3. Analysis

In diesem Abschnitt stellen wir die in der reellen Analysis laufend gebrauchten Funktionen kurz vor. Wegen weiter fortgeschrittener Anwendungen oder Ergänzungen verweisen wir auf spätere Kapitel, namentlich jene, die die numerische Integration (Kapitel 14), die Lösung nichtlinearer Gleichungen (Kapitel 12) und Differentialgleichungen (Kapitel 10) behandeln.

2.3.1. Summen

Zur Berechnung symbolischer Summen verwendet man die Funktion `sum`. Berechnen wir beispielsweise die Summe der ersten n natürlichen Zahlen:

```
sage: k, n = var('k, n')
sage: sum(k,k,1,n).factor()
1/2*(n + 1)*n
```

Die Funktion `sum` erlaubt die Vereinfachung des binomischen Lehrsatzes:

```
sage: n, k, y = var('n, k, y') # das symbolische x ist voreingestellt
sage: sum(binomial(n,k) * x^k * y^(n-k), k, 0, n)
(x + y)^n
```

Hier nun weitere Beispiele, darunter die Summe der Zahl der Elemente einer Teilmenge mit n Elementen:

```
sage: k, n = var('k, n')
sage: sum(binomial(n,k), k, 0, n),\
....: sum(k * binomial(n, k), k, 0, n),\
....: sum((-1)\^k*binomial(n,k), k, 0, n)
(2^n, 2^(n - 1)*n, 0)
```

Schließlich noch Beispiele für geometrische Summen:

```
sage: a, q, k, n = var('a, q, k, n')
sage: sum(a*q^k, k, 0, n)
(a*q^(n + 1) - a)/(q - 1)
```

Um die entsprechende Reihe zu berechnen, muss präzisiert werden, dass der Quotient q absolut kleiner als 1 ist:

```
sage: assume(abs(q) < 1)
sage: sum(a*q^k, k, 0, infinity)
-a/(q - 1)
```

```
sage: forget(); assume(q > 1); sum(a*q^k, k, 0, infinity)
Traceback (most recent call last):
...
ValueError: Sum is divergent.
```

Übung 2 *(Rekursive Berechnung der Summe).* Zu berechnen ist die Summe der p-ten Potenzen der ganzen Zahlen von 0 bis n für $p = 1, \ldots, 4$, ohne den Sage-Algorithmus zu verwenden:

$$S_n(p) = \sum_{k=0}^{n} k^p.$$

Zur Berechnung diese Summe kann man folgende Rekursionsgleichung verwenden:

$$S_n(p) = \frac{1}{p+1}\left((n+1)^{p+1} - \sum_{j=0}^{p-1}\binom{p-1}{j} S_n(j)\right).$$

Diese Rekursionsformel erweist sich bei der Berechnung der Teleskopsumme

$$\sum_{0\le k\le n} (k+1)^{p+1} - k^{p+1},$$

als hilfreich, die auf zweierlei Art erfolgen kann.

2.3.2. Grenzwerte

Zur Berechnung eines Grenzwertes verwendet man den Befehl `limit` oder seinen Alias `lim`. Zu berechen sind die folgenden Grenzwerte:

(a) $\lim\limits_{x\to 8} \dfrac{\sqrt[3]{x}-2}{\sqrt[3]{x+19}-3}$;

(b) $\lim\limits_{x\to \frac{\pi}{4}} \dfrac{\cos(\frac{\pi}{4}-x)-\tan x}{1-\sin(\frac{\pi}{4}+x)}$.

```
sage: limit((x**(1/3) - 2)/((x+19)**(1/3) - 3), x = 8)
9/4
sage: f(x) = (cos(pi/4-x)-tan(x))/(1-sin(pi/4+x))
sage: limit(f(x), x = pi/4)
Infinity
```

Die letzte Antwort zeigt an, dass einer der Grenzwerte, der rechte oder der linke, unendlich[4] ist. Um das Resultat zu präzisieren, untersucht man mit Hilfe der Option `dir` die Grenzwerte von links (`minus`) und von rechts (`plus`):

```
sage: limit(f(x), x = pi/4, dir='minus')
+Infinity
sage: limit(f(x), x = pi/4, dir='plus')
-Infinity
```

2.3.3. Folgen

Die bis jetzt behandelten Funktionen erlauben die Untersuchung von Folgen. Als Beispiel vergleichen wir das Wachstum einer exponentiellen Folge und einer geometrischen.

Beispiel. *Untersuchung einer Folge* Wir betrachten die Folge $u_n = \dfrac{n^{100}}{100^n}$. Zu berechnen sind die ersten zehn Terme der Folge. Welche Monotonie weist die Folge auf? Welches ist der Grenzwert der Folge. Für welche n ist $u_n \in]0, 10^{-8}[$?

1. Um den Term der Folge u_n zu definieren, verwenden wir eine symbolische Funktion. Wir führen die Berechnung der ersten 10 Terme „von Hand“ aus (und warten mit den Schleifen bis zum 3. Kapitel):

   ```
   sage: u(n) = n^100/100^n
   sage: u(2.); u(3.); u(4.); u(4.); u(6.); u(7.); u(8.); u(9.); u(10.)
   ```

[4] Statt von einem *unendlichen Grenzwert* zu sprechen (oft wird auch *uneigentlicher Grenzwert* gesagt), ist es auch möglich zu sagen, dass ein Grenzwert nicht existiert oder dass ein Wert über alle Grenzen wächst. Schließlich ist ∞ weder Zahl noch Wert, sondern bezeichnet eine Eigenschaft.

```
1.26765060022823e26
5.15377520732011e41
1.60693804425899e52
1.60693804425899e52
6.53318623500071e65
3.23447650962476e70
2.03703597633449e74
2.65613988875875e77
1.00000000000000e80
```

Daraus könnte man übereilt folgern, dass u_n gegen unendlich strebt.

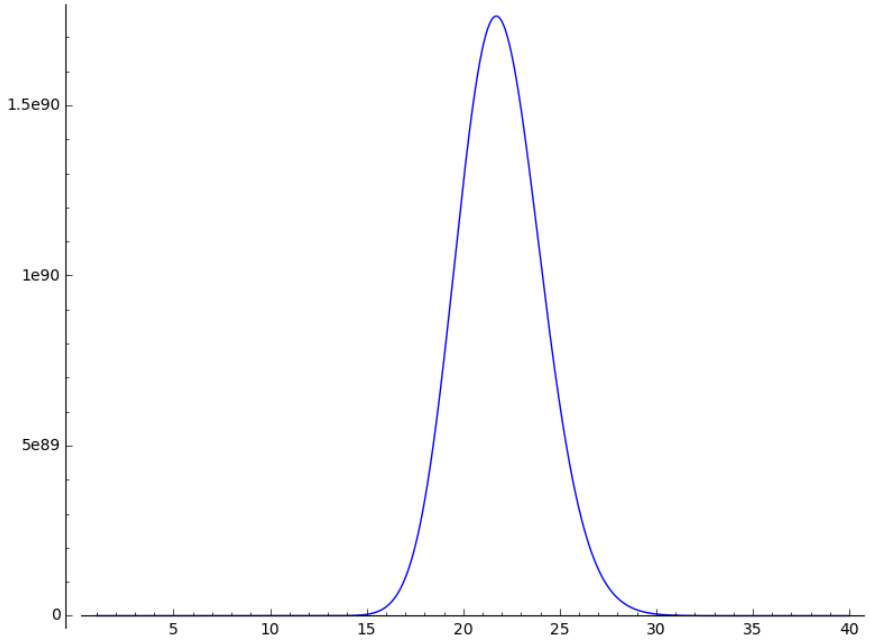


Abb. 2.2 - Graph von $x \mapsto x^{100}/100^x$

2. Um eine Vorstellung von der Monotonie zu haben, kann man die Funktion zeichnen, mit der wir die Folge u_n definiert haben (siehe Abb. 2.2).

   ```
   plot(u(x), x, 1, 40)
   ```

   Wir vermuten, dass die Folge ab $n = 22$ abnehmen wird.

   ```
   sage: v(x) = diff(u(x), x); sol = solve(v(x) == 0, x); sol
   [x = 100/log(100), x = 0]
   sage: floor(sol[0].rhs())
   21
   ```

   Die Folge wächst also bis $n = 21$, um ab $n = 22$ wieder abzunehmen.

3. Wir berechnen nun den Grenzwert.

   ```
   sage: limit(u(n), n=infinity)
   0
   sage: n0 = find_root(u(n) - 1e-8 == 0, 22, 1000); n0
   105.07496210187252
   ```

Die ab $n = 22$ abnehmende Folge verläuft ab $n = 106$ im Intervall $]0, 10^{-8}[$.

2.3.4. Taylor-Reihenentwicklung (*)

Zur Berechnung einer Taylor-Reihe der Ordnung n an der Stelle x_0 verfügt man über die Methode `f(x).series(x==x0, n)`. Interessiert man sich nur für den regulären Teil der Reihenentwicklung der Ordnung n, kann man die Funktion `taylor(f(x), x, x0, n)` verwenden. Bestimmen wir die Reihenentwicklungen folgender Funktionen im Sage-Arbeitsblatt:

a) $(1 + \arctan x)^{\frac{1}{x}}$ der Ordnung 3 um $x_0 = 0$,

b) $\ln(2 \sin x)$ der Ordnung 3 um $x_0 = \frac{\pi}{6}$.

```
taylor((arctan(x))**(1/x), x, 0, 3)
```

$$\frac{1}{810}\left(112\,x^3 + 45\,x^2 - 270\,x + 810\right)e^{\frac{\log(x)}{x}}$$

```
(ln(2*sin(x)).series(x==pi/6, 3))
```

$$(\sqrt{3})(-\frac{1}{6}\,\pi + x) + (-2)(-\frac{1}{6}\,\pi + x)^2 + \mathcal{O}\left(-\frac{1}{216}\,(\pi - 6\,x)^3\right)$$

Hat man die Taylor-Reihe mittels `series` erhalten und möchte den regulären Teil allein haben, verwendet man die Methode `truncate`:

```
(ln(2*sin(x)).series(x==pi/6, 3)).truncate()
```

$$-\frac{1}{18}\,(\pi - 6\,x)^2 - \frac{1}{6}\,\sqrt{3}(\pi - 6\,x)$$

Mit dem Befehl `taylor` ist es auch möglich, asymptotische Entwicklungen zu erhalten. Um bespielsweise den Wert der Funktion $(x^3 + x)^{\frac{1}{3}} - (x^3 - x)^{\frac{1}{3}}$ für $x \to \infty$ zu berechnen, schreibt man:

```
sage: taylor((x^3+x)^(1/3)-(x^3-x)^(1/3), x, infinity, 2)
2/3/x
```

Übung 3 *(Eine symbolische Grenzwertberechnung).* Sei f aus der Klasse $\mathbb{C}^2$ in der Umgebung von $a \in \mathbb{R}$ gegeben. Zu berechnen ist

$$\lim_{h \to 0} \frac{1}{h^3}(f(a + 3h) - 3f(a + 2h) + 3(a + h) - f(a).$$

Verallgemeinerung?

Beispiel (*) *(Die Formel von Machin).* Zu beweisen ist folgende Gleichung:

$$\frac{\pi}{4} = 4 \arctan \frac{1}{5} - \arctan \frac{1}{239}.$$

Mit Hilfe dieser Formel und der `arctan`-Entwicklung als Potenzreihe hat der Astronom John Machin (1680-1752) im Jahre 1706 π auf 100 Dezimalstellen berechnet. Mit seinem Verfahren ist ein Näherungswert für π herzuleiten.

Funktionen und Operatoren	
Ableitung	`diff(f(x), x)`
n-te Ableitung	`diff(f(x), x, n)`
Integration	`integrate(f(x), x)`
numerische Integration	`integral_numerical((f(x), a, b)`
symbolische Summe	`sum(f(i), i, imin, imax)`
Grenzwert	`limit(f(x), x==a)`
Taylorpolynom	`taylor(f(x), x, a, n)`
Taylor-Reihenentwicklung	`f.series(x==a, n)`
Graph einer Funktion	`plot(f(x), x, a, b)`

Tab. 2.4 - Zusammenstellung von in der Analysis hilfreichen Funktionen

Zunächst überzeugt man sich, dass $4\arctan\frac{1}{5}$ und $\frac{\pi}{4}+\arctan\frac{1}{239}$ den gleichen Tangenswert haben.

```
sage: tan(4*arctan(1/5)).simplify_trig()
120/119
sage: tan(pi/4+arctan(1/239)).simplify_trig()
120/119
```

Nun liegen die reellen Zahlen $4\arctan\frac{1}{5}$ und $\frac{\pi}{4}+\arctan\frac{1}{239}$ beide im offenen Intervall $]0,\pi[$, sie sind also gleich. Um einen Näherungswert für π zu erhalten, kann man wie folgt verfahren:

```
sage: f = arctan(x).series(x, 10); f
1*x + (-1/3)*x^3 + 1/5*x^5 + (-1/7)*x^7 + 1/9*x^9 + Order(x^10)
sage: (16*f.substitute(x==1/5) - 4*f.substitute(x==1/239)).n(); pi.n()
3.14159268240440
3.14159265358979
```

Übung 4 *(Eine Formel von Gauß).* Die folgende Formel erfordert in der Ausgabe von Gauß' Werken (*Werke*, ed. Königl. Ges. d. Wiss. Göttingen, vol 2, p. 477-502) immerhin 20 Seiten Tabellenrechnung:

$$\frac{\pi}{4}=12\arctan\frac{1}{38}+20\arctan\frac{1}{57}+7\arctan\frac{1}{239}+24\arctan\frac{1}{268}.$$

1. Man setze $\theta=12\arctan\frac{1}{38}+20\arctan\frac{1}{57}+7\arctan\frac{1}{239}+24\arctan\frac{1}{268}$. Bestätigung mit Sage, dass $\tan\theta=1$.
2. Zu beweisen ist die Ungleichung $\forall x\in\left[0,\frac{\pi}{4}\right]$, $\tan x\leq\frac{\pi}{4}x$ durch Herleitung der Formel von Gauß.
3. Durch Approximation der `arctan`-Funktion vermittels des Taylor-Polynoms der Ordnung 21 an der Stelle 0 ist eine neuer Näherungswert für π anzugeben.

2.3.5. Reihen (*)

Die vorstehenden Befehle können auch auf Reihen angewendet werden. Wir geben einige Beispiele:

Beispiel *(Berechnung der Summe der Riemann-Reihe).*

```
k = var('k')
sum(1/k^2, k, 1, infty),\
```

```
sum(1/k^4, k, 1, infty),\
sum(1/k^5, k, 1, infty)
```
$(\frac{1}{6}\pi^2, \frac{1}{90}\pi^4, \zeta(5))$

Beispiel *(Eine Formel von Ramanujan).* Mit der Partialsumme der ersten 12 Terme der folgenden Summe geben wir eine Näherung für π und vergleichen sie mit dem von Sage für π gefundenen Wert.

$$\frac{1}{\pi} = \frac{2\sqrt{2}}{9801} \sum_{k=0}^{+\infty} \frac{(4k)! \cdot (1103 + 26390k)}{(k!)^4 \cdot 396^{4k}}.$$

```
sage: s = 2*sqrt(2)/9801*(sum((factorial(4*k))*(1103+26390*k)/
..... factorial(k))^4*396^(4*k)) for k in (0..11)))
sage: (1/s).n(digits=100)
3.1415926535897932384626433832795028841971693993751058209 74...
sage: (pi-1/s).n(digits=100).n()
-4.36415445739398e-96
```

Man sieht, dass die Patialsumme der ersten 12 Terme schon 95 signifikante Stellen von π liefert!

Beispiel *(Konvergenz einer Reihe).* Es ist die Natur folgender Reihe zu untersuchen:

$$\sum_{n\geq 0} \sin\left(\pi\sqrt{4n^2+1}\right).$$

Um eine asymptotische Entwicklung des allgemeinen Gliedes zu erhalten, benutzt man eine mit 2π periodische Sinusfunktion, die gegen 0 strebt:

$$u_n = \sin\left(\pi\sqrt{4n^2+1}\right) \Rightarrow \sin\left[\pi\left(\sqrt{4n^2+1} - 2n\right)\right].$$

Nun kann man die Funktion `taylor` auf die neue Gestalt des allgemeinen Terms anwenden

```
n = var('n'); u = sin(pi*(sqrt(4*n^2+1)-2*n))
taylor(u, n, infinity, 3)
```
$$\frac{\pi}{4n} - \frac{6\pi + \pi^2}{384n^3}$$

Daraus bekommt man $u_n \sim \frac{\pi}{4n}$, was nach den Vergleichs-Regeln für Riemann-Reihen zeigt, dass die Reihe $\sum_{n\geq 0} u_n$ divergiert.

Übung 5 *(Asymptotische Entwicklung einer Folge).* Es ist unschwer zu zeigen (beispielsweise mit dem Satz über monotone Bijektionen), dass die Gleichung $\tan x = x$ für jedes $n \in \mathbb{N}$ im Intervall $[n\pi, n\pi + \frac{\pi}{2}]$ genau eine Lösung x_n besitzt. Anzugeben ist eine asymptotische Entwicklung von x_n für $+\infty$ mit der Ordnung 6.

2.3.6. Ableitung

Die Funktion `derivative` (mit dem Alias `diff`) ermöglicht die Ableitung eines symbolischen Ausdrucks oder einer symbolischen Funktion.

```
sage: diff(sin(x^2), x)
2*x*cos(x^2)
sage: function('f', x); function('g', x); diff(f(g(x)), x)
f(x)
g(x)
D[0](f)(g(x))*diff(g(x), x)
sage: diff(ln(f(x)), x)
diff(f(x), x)/f(x)
```

2.3.7. Partielle Ableitungen

Der Befehl `diff` erlaubt auch die Berechnung der n-ten Ableitung oder von partiellen Ableitungen.

```
sage: f(x,y) = x*y + sin(x^2) +e^(-x); derivative(f, x)
(x, y) |--> 2*x*cos(x^2) + y - e^(-x)
sage: derivative(f, y)
(x, y) |--> x
```

Beispiel. Zu verifizieren ist, dass die folgende Funktion harmonisch[5] ist:

$$f(x,y) = \frac{1}{2}\ln(x^2+y^2) \text{ für alle } (x,y) \neq (0,0).$$

```
sage: x, y = var('x, y'); f = ln(x^2 + y^2)/2
sage: delta = diff(f,x,2) + diff(f,y,2)
sage: delta.simplify_full()
0
```

Übung 6 *(Ein Gegenbeispiel von Peano zum Satz von Schwarz).* Sei f eine Abbildung von $\mathbb{R}^2$ nach $\mathbb{R}$, die definiert ist durch

$$f(x,y) = \begin{cases} xy\frac{x^2-y^2}{x^2+y^2}, & \text{falls } (x,y) \neq (0,0) \\ 0, & \text{falls } (x,y) = (0,0) \end{cases}$$

Gilt $\partial_1\partial_2 f(0,0) = \partial_2\partial_1 f(0,0)$?

[5] Eine Funktion heißt harmonisch, wenn gilt $\Delta f = \partial_1^2 f + \partial_2^2 = 0$ mit dem Laplace-Operator Δ.

2.3.8. Integration

Für die Berechnung bestimmter und unbestimmter Integrale verwenden wir die Funktion bzw die Methode `integrate` oder deren Alias `integral`.

```
sage: sin(x).integral((x, 0, pi/2))
1
sage: integrate(1/(1+x^2), x)
arctan(x)
sage: integrate(1/(1+x^2), x, -infinity, infinity)
pi
sage: integrate(exp(-x**2), x, 0, infinity)
1/2*sqrt(pi)
```

```
sage: integrate(exp(-x), -infinity, infinity)
Traceback (most recent call last):
...
ValueError: Integral is divergent.
```

Beispiel. Für $x \in \mathbb{R}$ ist das Integral $\varphi(x) = \int_0^{+\infty} \frac{x \cos u}{u^2 + x^2} du$ zu berechnen.

```
sage: u = var('u'); f = x*cos(u)/(u^2+x^2)
sage: assume(x>0); f.integrate(u, 0, inifinity)
1/2*pi*e^(-x)
sage: forget(); assume(x<0); f.integrate(u, 0, infinity)
-1/2*pi*e^x
```

Somit haben wir: $\forall x \in \mathbb{R}^*, \quad \varphi(x) = \frac{\pi}{2} \cdot \operatorname{sgn}(x) \cdot e^{-|x|}$.

Um eine numerische Integration auf einem Intervall zu bewirken, verfügen wir über die Funktion `integral_numerical`, die ein *Tupel* mit zwei Elementen zurückgibt, dessen erste Komponente ein Näherungswert des Integrals ist und dessen zweite Komponte den Fehler abschätzt.

```
sage: integral_numerical(sin(x)/x, 0, 1)
(0.946083070367183, 1.0503632079297087e-14)
sage: g = integrate(exp(-x^2), x, 0, infinity)
sage: g, g.n()
(1/2*sqrt(pi), 0.886226925452758)
sage: approx = integral_numerical(exp(-x^2), 0, infinity)
sage: approx
(0.8862269254527568, 1.714774436012769e-08)
sage: approx[0] - g.n()
-1.11022302462516e-15
```

Übung 7 *(BBP-Formel).* Wir versuchen, die BBP-Formel (Bailey-Borwein-Plouffe-Formel) durch eine symbolische Rechnung aufzustellen: diese Formel ermöglicht die Berechnung der n-ten Nachkommastelle von π zur Basis 2 (oder 16), ohne die vorangehenden Ziffern zu kennen, und das mit geringem Aufwand an Speicherplatz und Zeit. Für $N \in \mathbb{N}$ setzen wir

$$S_N = \sum_{n=0}^{N} \left(\frac{4}{8n+1} - \frac{2}{8n+4} - \frac{1}{8n+5} - \frac{1}{8n+6} \right) \left(\frac{1}{16} \right)^n .$$

1. Sei die Funktion $f : t \mapsto 4\sqrt{2} - 8t^3 - 4\sqrt{2}t^4 - 8t^5$. Für $N \in \mathbb{N}$ ist das folgende Integral als Funktion von S_N auszudrücken:

$$I_N = \int_0^{1/\sqrt{2}} f(t) \left(\sum_{n=0}^{N} t^{8n} \right) \mathrm{d}t.$$

2. Für $N \in \mathbb{N}$ setze man $J = \int_0^{1/\sqrt{2}} \frac{f(t)}{1-t^8} \mathrm{d}t$. Zu zeigen ist $\lim_{N \to +\infty} S_N = J$.

3. Beweisen Sie die BBP-Formel:

$$\sum_{n=0}^{+\infty} \left(\frac{4}{8n+1} - \frac{2}{8n+4} - \frac{1}{8n+5} - \frac{1}{8n+6} \right) \left(\frac{1}{16} \right)^n = \pi.$$

   Diese bemerkenswerte Formel wurde am 19. September 1995 von Simon Plouffe in Zusammenarbeit mit David Bailey und Peter Blorwein erhalten. Dank einer aus der BBP-Formel abgeleiteten Gleichung wurde im Jahre 2001 die 4.000.000.000.000.000. Stelle von π zur Basis 2 bestimmt.

2.4. Elementare lineare Algebra (*)

In diesem Abschnitt beschreiben wir die grundlegenden Funktionen der lineare Algebra: Operationen auf Vektoren und dann auf Matrizen. Wegen weiterer Details siehe Kapitel 8 über symbolische Matrizenrechnung und Kapitel 13 über numerische Matrizenrechnung.

2.4.1. Lösung linearer Gleichungssysteme

Zur Lösung eines linearen Systems können wir die Funktion `solve` nehmen, die uns schon begegnet ist.

Übung 8 *(Polynomiale Näherung für den Sinus).* Es ist das Polynom höchstens 5. Grades zu bestimmen, das die beste Näherung im Sinne der kleinsten Quadrate im Intervall $[-\pi, \pi]$ für die Sinusfunktion realisiert:

$$\alpha_5 = \min \left\{ \int_{-\pi}^{\pi} |\sin x - P(x)|^2 dx | P \in \mathbb{R}_5[x] \right\}.$$

2.4.2. Vektorrechnung

Die grundlegenden Funktionen für das Rechnen mit Vektoren sind in Tab. 2.5 zusammengestellt.

Auf Vektoren gebräuchliche Funktionen	
Deklaration eines Vektors	`vector`
Vektorprodukt	`cross_product`
Skalarprodukt	`dot_product`
Norm eines Vektors	`norm`

Tabelle 2.5 - Vektorrechnung

Wir können uns dieser Funktionen bei der Behandlung der folgenden Übung bedienen.

Übung 9 *(Das Problem von Gauß).* Wir betrachten einen Satelliten in seiner Umlaufbahn um die Erde, von der wir drei Punkte kennen: A_1, A_2 und A_3. Ausgehend von diesen drei Punkten möchten wir die Parameter des Orbits dieses Satelliten bestimmen.

Wir bezeichnen mit O den Erdmittelpunkt. Die Punkte O, A_1, A_2 und A_3 liegen offenbar in derselben Ebene, nämlich der Bahnebene des Satelliten. Die Bahn des Satelliten ist eine Ellipse und O einer ihrer Brennpunkte. Wir können ein Koordinatensystem $(O, \vec{i}, \vec{j})$, derart wählen, dass die Ellipsengleichung in Polarkoordinaten in diesem Koordinatensystem $r = \dfrac{p}{1 - e\cos\theta}$ ist, wobei e die Exzentrizität der Ellipse bezeichnet und p ihren Parameter. Wir werden schreiben $\vec{r_i} = \vec{OA_i}$ und $r_i = |\vec{r_i}|$ für $i \in \{1, 2, 3\}$. Wir betrachten dann die folgenden drei Vektoren, die sich aus der Kenntnis von A_1, A_2 und A_3 ergeben:

$$\vec{D} = \vec{r_1} \wedge \vec{r_2} + \vec{r_2} \wedge \vec{r_3} + \vec{r_3} \wedge \vec{r_1}$$
$$\vec{S} = (r_1 - r_3) \cdot \vec{r_2} + (r_3 - r_3) \cdot \vec{r_1} + (r_2 - r_1) \cdot \vec{r_3}$$
$$\vec{N} = r_3 \cdot (\vec{r_1} \wedge \vec{r_2}) + r_1 \cdot (\vec{r_2} \wedge \vec{r_3}) + r_2 \cdot (\vec{r_3} \wedge \vec{r_1})$$

1. Zeigen Sie $\vec{i} \wedge \vec{D} = -\frac{1}{e}\vec{S}$ und leiten Sie daraus die Exzentrizität der Ellipse her.
2. Zeigen Sie: $\vec{i}$ ist kollinear zum Vektor $\vec{S} \wedge \vec{D}$.
3. Zeigen Sie $\vec{i} \wedge \vec{N} = -\frac{p}{e}\vec{S}$ und leiten Sie daraus den Parameter p der Ellipse her.
4. Drücken Sie die große Halbachse a der Ellipse als Funktion des Parameters p und der Exzentrizität e aus.
5. *Anwendung mit Zahlen:* In der erwähnten Ebene in einem rechtwinkligen Koordinatensystem betrachten wir die Punkte

$$A_1 \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \quad A_2 \begin{pmatrix} 2 \\ 2 \end{pmatrix}, \quad A_3 \begin{pmatrix} 3 \\ 5 \end{pmatrix}, \quad O\,(0)\,.$$

   Zu bestimmen sind die Zahlenwerte der charakteristischen Größen der Ellipse mit dem Brennpunkt O durch die Punkte A_1, A_2 und A_3.

2.4.3. Matrizenrechnung

Zur Definition einer Matrix verwenden wir den Befehl `matrix` und legen dabei eventuell auch den Ring (oder den Körper) fest, aus dem die Einträge genommen werden.

```
sage: A = matrix(QQ, [[1,2],[3,4]]); A
[1 2]
[3 4]
```

Um für die Matrixgleichung $Ax = b$ (bzw. $xA = b$) eine Lösung zu finden, benutzen wir die Funktion `solve_right` bzw. `solve_left`. Um *alle* Lösungen einer Matrixgleichung zu finden, muss man einer partikulären Lösung die allgemeine Lösung der zugehörigen homogenen Gleichung hinzufügen. Für die Lösung einer homogenen Gleichung der Form $Ax = 0$ (bzw. $xA = 0$) steht die Funktion `right_kernel` bzw. `left_kernel` zur Verfügung, wie das folgende Beispiel zeigt.

Übung 10 *(Basen von Untervektorräumen).*

1. Zu bestimmen ist eine Basis des Lösungsraumes des homogenen linearen Gleichungssystems mit der Matrix
$$A = \begin{pmatrix} 2 & -3 & 2 & -12 & 33 \\ 6 & 1 & 26 & -16 & 69 \\ 10 & -29 & -18 & -53 & 32 \\ 2 & 0 & 8 & -18 & 84 \end{pmatrix}$$
2. Zu bestimmen ist eine Basis des von den Spalten von A aufgespannten Vektorraums F.
3. F ist durch eine oder mehrere Gleichungen zu charakterisieren.

Übung 11 *(Eine Matrixgleichung).*

Wir erinnern uns an das Lemma zur Faktorisierung linearer Anwendungen. Seien E, F, G $\mathbb{K}$-Vektorräume endlicher Dimension. Seien weiter $u \in \mathcal{L}(E, F)$ und $v \in \mathcal{L}(E, G)$. Dann sind die folgenden Aussagen äquivalent:

(i) es existiert $w \in \mathcal{L}(F, G)$, sodass $v = w \circ u$ ist,

(ii) $\ker u \subset \ker v$.

Wir suchen alle Lösungen für dieses Problem in einem konkreten Fall. Es seien
$$A = \begin{pmatrix} -2 & 1 & 1 \\ 8 & 1 & -5 \\ 4 & 3 & -3 \end{pmatrix} \text{ und } C = \begin{pmatrix} 1 & 2 & -1 \\ 2 & -1 & -1 \\ -5 & 0 & 3 \end{pmatrix}.$$
Zu bestimmen sind alle Lösungen $B \in \mathcal{M}_3(\mathbb{R})$ der Gleichung $A = BC$.

Auf Matrizen gebräuchliche Funktionen	
Deklaration einer Matrix	`matrix`
Lösung einer Matrixgleichung	`solve_right, solve_left`
Kern von rechtss, Kern von links	`right_kernel, left_kernel`
Reduktion auf Treppennormalform	`echelon_form`
Spaltenvektorraum	`column_space`
Verkettung von Matrizen	`matrix_block`

Reduktion von Matrizen	
Eigenwerte einer Matrix	`eigenvalues`
Eigenvektoren einer Matrix	`eigenvectors_right`
Reduktion auf Jordan-Normalform	`jordan_form`
Minimalpolynom einer Matrix	`minimal_polynomial`
charakteristisches Polynom einer Matrix	`chrakteristic_polynomial`

Tabelle 2.6 - Matrizenrechnung

2.4.4. Reduktion einer quadratischen Matrix

Für die Untersuchung der Eigenschaften einer Matrix stehen uns die in Tab. 2.6 aufgelisteten Methoden zur Verfügung.

Genauer werden diese Methoden in Kapitel 8 behandelt. Hier geben wir uns mit einigen Beispielen für ihre Anwendung zufrieden.

BEISPIEL. Is die Matrix $\begin{pmatrix} 2 & 4 & 3 \\ -4 & -6 & -3 \\ 3 & 3 & 1 \end{pmatrix}$ diagonalisierbar? Triagonalisierbar?

Wir beginnen mit der Definition der Matrix A auf dem Körper der rationalen Zahlen ($QQ = \mathbb{Q}$). Danach bestimmen wir ihre Eigenschaften.

```
sage: A = matrix(QQ, [[2,4,3],[-4,-6,-3],[3,3,1]])
sage: A.characteristic_polynomial()
x^3 + 3*x^2 - 4
sage: A.eigenvalues()
[1, -2, -2]
sage: A.minimal_polynomial().factor()
(x - 1) * (x + 2)^2
```

Das minimale Polynom von A besitzt eine einfache und eine doppelte Wurzel. Daher ist A nicht diagonalisierbar. Hingegen ist das minimale Polynom von A zusammengesetzt. Daher ist A triagonalisierbar.

```
sage: A.eigenvectors_right()
```

$$[(1, [(1, -1, 1)], 1), (-2, [(1, -1, 0)], 2)]$$

```
sage: A.jordan_form(transformation=True)
```

$$\left(\left(\begin{array}{c|cc} 1 & 0 & 0 \\ \hline 0 & -2 & 1 \\ 0 & 0 & -2 \end{array}\right), \begin{pmatrix} 1 & 1 & 1 \\ -1 & -1 & 0 \\ 1 & 0 & -1 \end{pmatrix}\right)$$

BEISPIEL. Die Matrix $A = \begin{pmatrix} 1 & -1/2 \\ -1/2 & -1 \end{pmatrix}$ ist zu diagonalisieren. Man kann es mit der Methode `jordan_form` versuchen:

```
sage: A = matrix(QQ, [[1,-1/2],[-1/2,-1]])
sage: A.jordan_form()
Traceback (click to the left of this block for traceback)
...
RuntimeError: Some eigenvalue does not exist in Rational Field.
```

Hier ist eine kleine Schwierigkeit aufgetreten: die Eigenwerte sind nicht rational.

```
sage: A.minimal_polynomial()
```

$$x^2 - \frac{5}{4}$$

Wir müssen also den Basiskörper wechseln.

```
sage: R = QQ[sqrt(5)]
sage: A = A.change_ring(R)
```

```
sage: A.jordan_form(transformation=True, subdivide=False)
```
$$\left(\begin{pmatrix} \frac{1}{2}sqrt_5 & 0 \\ 0 & -\frac{1}{2}sqrt_5 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ -sqrt_5+2 & sqrt_5+2 \end{pmatrix}\right)$$

Das interpretieren wir als

$$\left(\begin{pmatrix} \frac{1}{2}\sqrt{5} & 0 \\ 0 & -\frac{1}{2}\sqrt{5} \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ -\sqrt{5}+2 & \sqrt{5}+2 \end{pmatrix}\right)$$

Beispiel. Die Matrix $A = \begin{pmatrix} 2 & \sqrt{6} & \sqrt{2} \\ \sqrt{6} & 3 & \sqrt{3} \\ \sqrt{2} & \sqrt{3} & 1 \end{pmatrix}$ ist zu diagonalisieren.

Dieses Mal müssen wir mit einer Erweiterung 4. Grades des Körpers $\mathbb{Q}$ arbeiten. Man geht nun vor wie folgt:

```
sage: K.<sqrt2> = NumberField(x^2 - 2)
sage: L.<sqrt3> = K.extension(x^2 - 3)
sage: A = matrix(L, [[2, sqrt2*sqrt3, sqrt2], \
                     [sqrt2*sqrt3, 3, sqrt3], \
                     [sqrt2, sqrt3, 1]])
sage: A.jordan_form(transformation=True)
```
$$\left(\left(\begin{array}{c|c|c} 6 & 0 & 0 \\ \hline 0 & 0 & 0 \\ \hline 0 & 0 & 0 \end{array}\right), \begin{pmatrix} 1 & 1 & 0 \\ \frac{1}{2}sqrt_2sqrt_3 & 0 & 1 \\ \frac{1}{2}sqrt_2 & -sqrt_2 & -sqrt_3 \end{pmatrix}\right)$$

oder

$$\left(\begin{pmatrix} 6 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 & 0 \\ \frac{1}{2}\sqrt{2}\sqrt{3} & 0 & 1 \\ \frac{1}{2}\sqrt{2} & -\sqrt{2} & -\sqrt{3} \end{pmatrix}\right).$$

3. Programmierung und Datenstrukturen

Wir haben in den vorangegangenen Kapiteln gesehen, wie mathematische Rechnungen mit isolierten Sage-Befehlen zu bewerkstelligen sind, doch das Systen ermächtigt auch zur Programmierung einer Folge von Anweisungen.

Das System Sage zum symbolischen Rechnen ist in der Tat eine Erweiterung der Programmiersprache Python[1] und erlaubt, wenn auch mit etlichen Veränderungen der Syntax, die Programmiermethoden dieser Sprache auszunutzen.

Die in den vorangegangenen Kapiteln beschriebenen Befehle beweisen, dass es nicht nötig ist, die Sprache Python zu kennen, um Sage zu verwenden; dagegen zeigt dieses Kapitel, wie in Sage die elementaren Programmier-Strukturen von Python angewendet werden. Das beschränkt sich auf die Grundlagen der Programmierung und kann von Menschen, die Python kennen, vielleicht nur überflogen werden: die Beispiele sind aus den klassischen Bereichen der Mathematik ausgewählt, um dem Leser durch Analogie mit der Programmiersprache, die er kennt, eine rasche Anpassung an Python zu ermöglichen.

Dieses Kapitel stellt die algorithmische Methode der strukturierten Programmierung vor mit Schleifen und Abfragen und stellt im folgenden die Funktionen vor, die auf Listen und anderen zusammengesetzten Datenstrukturen operieren. Das Buch *Apprendre à programmer avec Python* von G. Swinnen [Swi09, Swi12] (frei verfügbar) und der *Syllabus* online von T. Massart [Mas13] bieten eine vertiefte Darstellung der Sprache Python.

Schlüsselwörter der Programmierspache Python	
`while, for...in, if...elif...else`	Schleifen und Bedingungen
`continue, break`	vorzeitige Beendigung eines Codeblocks
`try...except...finally`	Behandlung und Auslösung von Fehlern
`assert`	einzuhaltende Bedingung
`pass`	Befehl ohne Wirkung
`def, lambda`	Definition einer Funktion
`return, yield`	Rückgabe eines Wertes
`global, del`	Sichtbarkeit und Löschen einer Variablen
`and, not, or`	logische Operatoren
`print`	Textausgabe
`class, with`	objektorientierte Programmierung, Anwendung von Kontext
`from...import...as`	Zugriff auf eine Bibliothek
`exec...in`	dynamische Auswertung von Code

Tabelle 3.1 - Allgemeine Syntax des Sage-Codes

[1] Sage verwendet Python 2.7, eine Änderung ist vorerst nicht zu erwarten.

3.1. Syntax

3.1.1. Allgemeine Syntax

Die elementaren Anweisungen werden generell Zeile für Zeile ausgeführt. Python betrachtet das Zeichen „#“ als Anfang eines Kommentars und ignoriert den Text von hier an bis zum Ende der Zeile. Das Semikolon „ ;“ trennt Anweisungen auf derselben Zeile:

```
sage: 2*3; 3*4; 4*5     # ein Kommentar, 3 Ergebnisse
6
12
20
```

Bei der Eingabe kann sich eine Anweisung über mehrere Zeilen erstrecken. Dann wird die alte Zeile mit einem umgekehrter Schrägstrich „\“ („backslash“) abgeschlossen, bevor die neue Zeile eröffnet wird. Der Wagenrücklauf („carriage return“) wird als Leerzeichen betrachtet.

```
sage: 123 + \
 ....: 345
468
```

Ein Bezeichner - also der Name einer Variablen, einer Funktion usw. - besteht nur aus Buchstaben, Ziffern oder dem Unterstrich „_“, und darf nicht mit einer Ziffer beginnen (der Name einer Datei aber schon). Die Bezeichner sollen sich von Schlüsselwörtern der Programmiersprache unterscheiden. Die Schlüsselwörter in Tab. 3.1 bilden den Kern von Python 2.7. Eine einfache Liste der Schlüsselwörter erhält man mit

```
sage: import keyword; keyword.kwlist
['and', 'as', 'assert', 'break', 'class', 'continue', 'def', 'del',
'elif', 'else', 'except', 'exec', 'finally', 'for', 'from',
'global', 'if', 'import', 'in', 'is', 'lambda', 'not', 'or', 'pass',
'print', 'raise', 'return', 'try', 'while', 'with', 'yield']
```

Zeichen in Sage mit besonderer Bedeutung und ihre Verwendung	
, ;	Trenner von Argumenten und Anweisungen
:	Eröffnung eines Anweisungsblocks, Raffung
.	Dezimalpunkt, Zugriff auf Felder von Objekten
=	Zuweisung eines Wertes an eine Variable
+ - * /	elementare arithmetische Operatoren
^ **	Potenz
// %	Quotient und Rest einer Ganzzahl-Division
+= -= *= /= **=	arithmetische Operatoren mit Veränderung einer Variablen
== != <> is	Tests auf Gleichheit
< <= > >=	Vergleichsoperatoren
& | ^^ << >>	Mengenoperatoren und logische Bit-Operatoren
#	Kommentar (bis Zeilenende)
[...]	Konstruktion einer Liste, Zugriff auf ein indiziertes Element
(...)	Aufruf einer Funktion oder Methode, unveränderliches Tupel
{...:...}	Konstruktion von Diktionären
\	Maskierung von Sonderzeichen, lineare Algebra
?	Zugriff auf die Hilfe
_ __ ___	Wiederholung der letzten drei Ergebnisse

Tabelle 3.2 - Allgemeine Syntax des Sage-Codes (Fortsetzung)

Diesen Schlüsselwörtern sind noch die Konstanten `None` (Wert „leer“), etwa äquivalent zu `NULL` in anderen Programmiersprachen, `True` und `False`, sowie zahlreiche, in Python und Sage vordefinierte Funktionen wie `len, cos` und `integrate` hinzuzufügen. Es ist sehr zu empfehlen, diese Bezeichner nicht als Namen für Variablen zu verwenden, weil dann der Zugang zu Funktionalitäten des Systems schwierig werden könnte. Der Interpreter akzeptiert noch mehr Befehle wie `quit` zum Verlassen der laufenden Sage-Sitzung. Wir werden davon nach und nach weitere kennenlernen.

Bestimmte Zeichen spielen in Sage eine besondere Rolle. Sie sind in Tab. 3.2 aufgelistet.

3.1.2. Aufruf von Funktionen

Die Auswertung einer Funktion verlangt, dass ihr eventuelle Argumente in Klammern übergeben werden, wie in `cos(pi)` oder in der Funktion ohne Argument `reset()`. Hingegen sind die Klammern für Argumente bei Befehlen überflüssig: `print(6*7)` und `print 6*7` leisten das gleiche[2]. Der Name einer Funktion ohne Argument oder Klammer stellt die Funktion selbst dar und bewirkt keinerlei Rechnung.

3.1.3. Ergänzungen zu Variablen

Wie wir gesehen haben, erkennt Sage im Gleichheitseichen „=“ die Anweisung, einer Variablen einen Wert zuzuweisen. Zuerst wird die Seite rechts von „=“ ausgewertet, dann wird ihr Wert der Variablen zugewiesen, deren Name links steht. So haben wir

```
sage: y = 3; y = 3*y + 1; y = 3*y + 1; y
31
```

Die obigen Zuweisungen verändern den Wert der Variablen `y` ohne einen Zwischenwert auszugeben. Der letzte dieser vier Befehle gibt den Wert der Variablen `y` aus, nachdem alle Rechnungen beendet sind.

Der Befehl `del x` unterbindet Zuweisungen an die Variable `x` und durch die Funktion ohne Argument `reset()` werden die Variablen wieder initialiert. Die gleichzeitige, parallele Zuweisung von Werten an mehrere Variablen ist auch möglich.

```
sage: a, b = 10, 20  # auch möglich: (a, b) = (10, 20) oder [10, 20]
sage: a, b = b, a
sage: a, b
(20, 10)
```

Die Zuweisung `a, b = b, a` vertauscht die Werte von `a` und `b` und ist äquivalent zum sonst üblichen Vorgehen mit einer Zwischenvariablen:

```
sage: temp = a: a = b; b = temp
```

Ein weiteres Beispiel für den Tausch der Werte zweier Variablen ohne Zwischenvariable oder paralle Zweisung, aber mit Summe und Differenz ist das folgende:

```
sage: x, y = var('x, y'); a = x; b = y
sage: a, b
```

[2] In Python 3.x ist `print` eine Funktion und erwartet seine Argumente in Klammern.

```
(x, y)
sage: a = a + b; b = a - b; a = a - b
sage: a, b
(y, x)
```

Die gleichzeitige Zuweisung desselben Wertes an mehrere Variable gelingt mit einer Syntax der Form `a = b = c = 0`; die Befehle `x += 5` und `n += 2` sind äquivalent zu `x = x + 5` bzw. zu `n = n + 2`.

Der Test auf Gleichheit zweier Objekte erfolgt mit dem aus zwei Gleichheitszeichen bestehenden Operator „==“.

```
sage: 2 + 2 == 2^2, 3*3 == 3^3
(True, False)
```

3.2. Algorithmik

Die strukturierte Programmierung besteht in der Beschreibung eines Programm-Ablaufs als eine endliche Folge von Anweisungen, die eine nach der anderen abgearbeitet werden. Diese Anweisungen können elementar oder zusammengesetzt sein.

- eine elementare Anweisung besteht beispielsweise aus der Zuweisung eines Wertes zu einer Variablen (siehe Unterabschnitt 1.2.4) oder der Ausgabe eines Ergebnisses;
- eine zusammengesetzte Anweisung wie eine Schleife oder eine Bedingung wird aus mehreren Anweisungen gebildet, die ihrerseits wieder elementar oder zusammengesetzt sein können.

3.2.1. Schleifen

Aufzählungsschleifen. Eine Aufzählungsschleife führt die gleiche Rechnung für alle ganzzahligen Werte einer Indexfolge $k \in [a, \ldots, b]$ aus. Das folgende Beispiel[3] gibt den Anfang der Multiplikationstabelle mit 7 aus:

```
sage: for k in [1..5]:
....:     print 7*k    # Block mit einer einzigen Annweisung
7
14
21
28
35
```

Der Doppelpunkt „:“ am Ende der ersten Zeile eröffnet den Anweisungsblock, welcher der Reihe nach mit $k = 1, 2, 3, 4, 5$ ausgewertet wird. Bei jeder Iteration gibt Sage das Produkt $7 \times k$ als momentanen Wert des Befehls `print` aus.

In diesem Beispiel besteht der Block der zu wiederholenden Anweisungen nur aus dem Befehl `print`, der gegen das Schlüsselwort `for` eingerückt ist. Ein Block aus mehreren Anweisungen

[3] Wird Sage auf einem Terminal ausgeführt, wird ein Block durch Eingabe einer Leerzeile beendet. Das ist bei der Arbeit mit Sage in einem Skript oder in einem Notebook nicht erforderlich und wird im folgenden stillschweigend so gehandhabt.

ist daran erkennbar, dass die eingegebenen Anweisungen alle mit der gleichen Einrückung untereinander stehen.

Die Kennzeichnung des Blocks ist wichtig: die beiden folgenden Programme unterscheiden sich nur bei der Einrückung einer Zeile und zeitigen unterschiedliche Ergebnisse:

```
sage: S = 0
sage: for k in [1..3]:
....:     S += k
sage: S *= 2
sage: S
```

```
sage: S = 0
sage: for k in [1..3]:
....:     S += k
....:     S *= 2
sage: S
```

Links wird die Anweisung `S *= 2` nach der Schleife einmal ausgeführt, während das rechts bei jeder Iteration geschieht, weswegen die Ergebnisse verschieden sind:

$$S = (0+1+2+3)\cdot 2 = 12 \qquad S = ((((0+1)\cdot 2)+2)\cdot 2+3)\cdot 2 = 22$$

Diese Schleife dient unter anderem zur Berechnung eines Terms einer rekursiven Folge wie am Ende dieses Unterabschnitts gezeigt.

Funktionen für die Iteration der Form `..range` für ganze Zahlen a, b, c	
`for k in [a..b]: ...`	erzeugt eine Liste von ganzen Sage-Zahlen $a \le k \le b$
`for k in srange(a, b): ...`	erzeugt eine Liste von ganzen Sage-Zahlen $a \le k < b$
`for k in range(a, b): ...`	erzeugt eine Liste von ganzen Python-Zahlen (`int`) $a \le k < b$
`for k in xrange(a, b): ...`	zählt die ganzen Python-Zahlen (`int`) auf, ohne die entsprechende Liste explizit zu erzeugen
`for k in sxrange(a, b): ...`	zählt die Sage-Zahlen auf ohne die Liste zu erzeugen
`[a,a+c..b], [a, b, step=c]`	die Sage-Zahlen $a, a+c, a+2c, \ldots, a+kc \le b$
`..range(b)`	äquivalent zu `..range(0, b)`
`..range(a, b, c)`	mit Schrittweite c statt 1

Tabelle 3.3 - Die verschiedenen Aufzählungsschleifen

Die Syntax einer Aufzählungsschleife ist sehr direkt und kann ohne weiteres für 10^4 oder 10^5 Iterationen verwendet werden. Sie eignet sich hingegen nicht für den Aufbau einer Liste aller Werte der Schleifenvariable, bevor die zu iterierenden Anweisungen wiederholt werden, sie hat vielmehr den Vorteil, die ganze Zahl des Sage-Typs `Integer` (siehe Unterabschnitt 5.3.1) zu manipulieren. Mehrere Funktionen `...range` erlauben für diese Iterationen die Auswahl aus zwei möglichen Alternativen. Die erste besteht entweder im Anlegen einer Werteliste im Speicher, bevor die Schleife ausgeführt wird oder in der Bestimmung dieser Werte im Laufe der Zeit. Die andere Alternative arbeitet mit ganzen Sage-Zahlen[4] des Typs `Integer` oder mit ganzen Python-Zahlen des Typs `int`; die beiden Ganzzahl-Typen haben nicht genau dieselben Eigenschaften. Im Zweifel bewahrt die Form `[a..b]` am sichersten vor Überraschungen.

`while`-Schleifen. Die andere Familie der Schleifen wird von den `while`-Schleifen gebildet. Wie auch die Aufzählungsschleife `for` wertet sie einen Anweisungsblock mehrmals aus. Indes ist die Anzahl der Wiederholungen nicht von vornherein festgelegt, sondern hängt von der Erfüllung einer Bedingung ab.

[4] Die Befehle `srange, sxrange` und `[...]` operieren auch auf rationalen und Fließpunkt-Zahlen: was gibt `[pi,pi+4..20]`?

Wie der Name sagt, wiederholt die Schleife die Anweisungen solange, wie eine Bedingung erfüllt ist. Das folgende Beispiel[5] berechnet die Summe der Quadrate der natürlichen Zahlen, solange die Potenz der natürlichen Zahl den Wert 10^6 nicht überschreitet, also $1^2+2^2+\ldots+13^2$.

```
sage: S = 0; k = 0           # Die Summme beginnt mit 0
sage: while e^k <= 10^6:     #      e^13 <= 10^5 < e^14
....:     S = S + k^2        # addiert die Quadrate k^2
....:     k = k + 1

sage: S
819
```

Die letzte Anweisung gibt den Wert der Variablen S aus. Es gilt:

$$S = \sum_{\substack{k\in\mathbb{N}\\ e^k\le 10^6}} k^2 = \sum_{k=0}^{13} k^2 = 819, \quad e^{13}\approx 442413 \le 10^6 < e^{14}\approx 1202604.$$

Der obige Anweisungsblock umfasst zwei Zuweisungen, die erste addiert den neuen Term, die zweite geht zum nächsten Index über. Beide Anweisungen sind untereinander angeordnet und innerhalb der `while`-Struktur auf die gleiche Weise eingerückt.

Das nächste ist wieder ein typisches Beispiel einer `while`-Schleife. Die sucht für eine Zahl $x \ge 1$ nach dem einzigen Wert $n \in \mathbb{N}$ sodass gilt $2^{n-1} \le x \le 2^n$, das heißt die kleinste ganze Zahl, sodass $x < 2^n$ ist. Das untenstehende Programm vergleicht x mit 2^n, dessen Wert $1, 2, 4, 9$ usw. ist. Sie führt diese Rechnung für $x = 10^4$ aus:

```
sage: x = 10^4; u = 1; n= 0              # invariant: u = 2^n
sage: while u <= x: n = n + 1; u = 2*u   # wobei n += 1; u *= 2
sage: n
14
```

Solange, wie die Bedingung $2^n \le x$ erfüllt ist, berechnet das Programm die neuen Werte $n+1$ und $2^{n+1} = 2\cdot 2^n$ der beiden Variablen n und u und speichert sie anstelle von n und 2^n. Diese Schleife endet, sobald die Bedingung nicht mehr erfüllt ist, also bei $x > 2^n$:

$$x = 10^4, \quad \min\{n\in\mathbb{N} | x < 2^n\} = 14, \quad 2^{13} = 8192, \quad 10^{14} = 16384.$$

Der Rumpf einer Schleife wird nie ausgeführt, wenn die Bedingung von Anfang an nicht erfüllt ist.

Einfache Anweisungsblöcke können nach dem Doppelpunkt „:“ auf derselben Zeile eingegeben werden, ohne einen neuen eingerückten Block zu definieren, der mit der neuen Zeile beginnen würde.

Beispiele für die Anwendung bei Folgen und Reihen. Die `for`-Schleife ermöglicht die einfache Berechnung eines Terms einer rekursiv definierten Folge. Es sei beispielsweise (u_n) die durch

$$u_0 = 1, \quad \forall n\in\mathbb{N}: \; u_{n+1} = \frac{1}{1+u_n^2}$$

[5] Die Eingabe einer Leerzeile ist erforderlich, um den Anweisungsblock zu beenden, bevor der Wert von S abgefragt werden kann.

definierte Folge. Das untenstehende Programm bestimmt eine numerische Näherung von u_n für $n = 20$; die Variable U wird bei jeder Schleifeniteration modifiziert, um mit der Rekursionsgleichung vom Wert u_{n-1} auf u_n zu kommen. Die erste Iteration berechnet u_1 aus u_0 für $n = 1$, die zweite macht dasselbe von u_1 nach u_2 mit $n = 2$, und die letzte der n Iterationen modifiziert die Variable U, um von u_{n-1} zu u_n zu gelangen:

```
sage: U = 1.0           # oder U = 1. oder U = 1.000
sage: for n in [1..20]:
....:     U = 1/(1 + U^2)
sage: U
0.68236043761105
```

Dasselbe Programm mit der Ganzzahl $U = 1$ anstelle der numerischen Approximation $U = 1.0$ in der ersten Zeile führt eine exakte Rechnung auf den rationalen Zahlen aus; das exakte Ergebnis für u_{10} ist ein Bruch mit mehr als hundert Ziffern, und u_{20} enthält davon mehrere Hunderttausend. Exakte Rechnungen sind interessant, wenn sich Rundungsfehler in den numerischen Approximationen anhäufen. Die Rechnung mit Näherungswerten, von Hand oder mit dem Rechner, mit Hunderten von Dezimalstellen sind schneller als die mit 500, 1000 oder noch mehr ganzen oder rationalen Zahlen.

Unterbrechung der Ausführung einer Schleife

Die `for`- und die `while`-Schleifen wiederholen dieselben Anweisungen mehrmals. Der Befehl `break` im Inneren einer Schleife beendet diese Schleife vorzeitig, und der Befehl `continue` springt sofort zur nächsten Iteration. So bewirken diese Befehle die Auswertung der Bedingung an beliebiger Stelle des Schleifenkörpers.
Die vier Beispiele hierunter bestimmen die kleinste natürliche Zahl x, die die Bedingung $\ln(x + 1) \leq x/10$ erfüllt. Das erste bringt eine `for`-Schleife mit maximal 100 Versuchen, die nach der ersten Lösung vorzeitig beendet wird; das zweite stellt die Suche nach der kleinsten Zahl dar, welche die Bedingung erfüllt und riskiert dabei, nicht zu enden, falls die Bedingung niemals erfüllt wird; das dritte entspricht dem ersten mit einer komplizierteren Schleifenbedingung, und schließlich besitzt das vierte Beispiel eine unnötig komplizierte Struktur, die nur den Zweck hat, den Befehl `continue` zu verwenden. In jedem dieser Fälle hat x den Endwert 37.0.

```
for x in [1.0..100.0]:
    if log(x+1) <= x/10: break
```

```
x = 1.0
while log(x+1) > x/10:
    x = x + 1
```

```
x = 1.0
while log(x+1) > x/10 and x < 100:
    x = x + 1
```

```
x = 1.0
while True:
    log(x+1) > x/10:
        x = x + 1
        continue
    break
```

Der Befehl `return` (der die Ausführung einer Funktion beendet und ihren Wert zurückgibt, siehe Unterabschnitt 3.2.3), stellt eine weitere Art des vorzeitigen Abbruchs eines Anweisungsblocks dar.

Summen und Produkte können in die Form von rekursiv definierten Folgen gebracht werden und werden auf die gleiche Weise berechnet:

$$S_n = \sum_{k=1}^{n} (2k)(2k+1) = 2 \cdot 3 + 4 \cdot 5 + \ldots + (2n)(2n+1),$$
$$S_0 = 0, \quad S_n = S_{n-1}$$

Diese Reihe wird auf dieselbe Weise programmiert wie die rekursiv definierten Folgen: das Programm bildet die Summen mit 0 beginnend eine nach der anderen, indem es die Terme für $k = 1$, $k = 2$ bis $k = n$ addiert:

```
sage: S = 0; n = 10
sage: for k in [1..n]:
...:     S = S + (2*k)*(2*k+1)
sage: S
1650
```

Dieses Beispiel veranschaulicht ein allgemeines Programmierverfahren für eine Summe, doch in diesem einfachen Fall zeitigt die formale Rechnung ein Ergebnis in voller Allgemeinheit:

```
sage: n, k = var('n k'); res = sum(2*k*(2*k+1), k, 1, n)
sage: res, factor(res)    # das Resultat wird anschließend faktorisiert
(4/3*n^3 + 3*n^2 + 5/3*n, 1/3*(4*n + 5)*(n + 1)*n)
```

Diese Resultate können auch mit *Papier und Bleistift* als wohlbekannte Summen erhalten werden:

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2}, \qquad \sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6},$$
$$\sum_{k=1}^{n} 2k(2k+1) = 4\sum_{k=1}^{n} nk^2 + 2\sum_{k=1}^{n} k = \frac{2n(n+1)(2n+1)}{3} + n(n+1)$$
$$= \frac{n(n+1)((4n+2)+3)}{3} = \frac{n(n+1)(4n+5)}{3}$$

Beispiele für die Annäherung an Grenzwerte von Folgen. Die `for`-Schleife ermöglicht die Berechnung eines gegebenen Terms einer Folge oder einer Reihe, die `while`-Schleife ist dagegen gut geeignet, den Grenzwert einer Folge numerisch anzunähern.

Konvergiert eine Folge $(a_n)_{n\in\mathbb{N}}$ gegen $l \in \mathbb{R}$, sind die Terme a_n für *hinreichend großes* n *Nachbarn* von l. Daher ist es möglich, l durch einen bestimmten Term a_n anzunähern und das mathematische Problem besteht darin, den Fehler $|l - a_n|$ mehr oder weniger leicht zu erhöhen. Diese Erhöhung ist für die benachbarten Folgen $(u_n)_{n\in\mathbb{N}}$ und $(v_n)_{n\in\mathbb{N}}$ unmittelbar, das heißt solche, für die gilt

$$\begin{cases} (u_n)_{n\in\mathbb{N}} \text{ nimmt zu,} \\ (v_n)_{n\in\mathbb{N}} \text{ nimmt ab,} \\ \lim\limits_{n\to+\infty} v_n - u_n = 0. \end{cases}$$

In diesem Fall gilt

$$\begin{cases} \text{beide Folgen konvergieren gegen den gleichen Grenzwert } l, \\ \forall p \in \mathbb{N}: \; u_p \leq \lim\limits_{n\to+\infty} u_n = l = \lim\limits_{n\to+\infty} v_n \leq v_p, \\ \left| l - \frac{u_p+v_p}{2} \right| \leq \frac{u_p-v_p}{2}. \end{cases}$$

Eine mathematische Fingerübung beweist, dass die beiden untenstehenden Folgen benachbart sind und gegen $\sqrt{ab}$ konvergieren, wenn $0 < a < b$ ist:

$$u_0 = a, \quad v_0 = b > a, \quad u_{n+1} = \frac{2u_n v_n}{u_n + v_n}, \quad v_{n+1} = \frac{u_n + v_n}{2}.$$

Der gemeinsame Grenzwert dieser beiden Folgen trägt den Namen arithmetisch-harmonisches Mittel, denn das arithmetische Mittel zweier Zahlen a und b ist Mittelwert im üblichen Sinne $(a + b)/2$ und das harmonische Mittel h ist das Inverse des Mittels der inversen: $1/h = (1/a + 1/b)/2 = (a + b)/(2ab)$. Das folgende Programm bestätigt, dass der Grenzwert für die Zahlenwerte derselbe ist:

```
sage: U = 2.0; V = 50.0
sage: while V-U >= 1.0e-6:      # 1-0e-6 bedeutet 1.0*10^-6
....:     temp = U
....:     U = 2*U*V/(U + V)
....:     V = (temp + V)/2
sage: U, V
(9.99999999989256, 10.0000000001074)
```

Die Werte von u_{n+1} und v_{n+1} hängen von u_n und v_n ab: aus diesem Grund lässt die Hauptschleife dieses Programms eine Zwischenvariable, die hier `temp` heißt, aufrufen, weil die neuen Werte vu_{n+1} und v_{n+1} von U und V von den zwei vorhergehenden Werten u_n und v_n abhängen. Die beiden Blöcke hierunter links definieren dieselben Folgen, während die unten rechts zwei andere Folgen, $(u'_n)_n$ und $(v'_n)_n$, bildet; die parallele Zuweisung vermeidet die Verwendung einer Zwischenvariablen:

```
temp = 2*U*V/(U+V)
V = (U + V)/2
U = temp
```

```
U,V = 2*U*V/(U+V), (U+V)/2
```

(parallele Zuweisung)

```
U = 2*U*V/(U+V)
V = (U+V)/2
```

$u'_{n+1} = \frac{2u'_n v'_n}{u'_n + v'_n}$

$v'_{n+1} = \frac{u'_{n+1} + v'_n}{2}$

Die Reihe $S_n = \sum_{k=0}^{n} (-1)^k a_k$ ist alternierend, da die Reihe $(a_n)_{n\in\mathbb{N}}$ abnimmt und den Grenzwert 0 hat. Zu sagen, dass S alternierend ist, bedeutet, dass die beiden Teilfolgen $(S_{2n})_{n\in\mathbb{N}}$ und $(S_{2n+1})_{n\in\mathbb{N}}$ benachbart sind und denselben Grenzwert l haben. Die Folge $(a_n)_{n\in\mathbb{N}}$ konvergiert daher auch gegen diesen Grenzwert und hat ihn bei $S_{2p+1} \leq l = \lim_{n\to+\infty} S_n \leq S_{2p}$.

Das folgende Programm illustriert dieses Ergebnis für die Folge $a_k = 1/k^3$, beginnend mit $k = 1$ und speichert dazu die aufeinander folgenden Patialsummen S_{2n} und S_{2n+1} der Reihe, die den Grenzwert einschachteln, in den beiden Variablen U und V:

```
sage: U = 0.0  # die Summe S0 ist leer, der Wert 0
sage: V = -1.0 # S1 = -1/1^3
sage: n = 0    # U und V enthalten S(2n) und S(2n+1)
sage: while U-V >= 1.0e-6:
....:     n = n+1             # n += 1 ist äquivalent
....:     U = V + 1/(2*n)^3   # Übergang von S(2n+1) zu S(2n)
....:     V = U + 1/(2*n+1)^3 # Übergang von S(2n) zu S(2n+1)
sage: V, U
(-0.901543155458595, -0.901542184868447)
```

Die Hauptschleife modifiziert den Wert von n, um zum nächsten Index überzugehen, solange die beiden Werte S_{2n} und S_{2n+1} nicht genügend nahe beieinander liegen. Die beidenn Variablen U und V speichern die aufeinander folgenden Terme; der Schleifenrumpf bestimmt beginnend mit S_{2n-1} nacheinander S_{2n} und S_{2n+1}, dann wechseln die Zuweisungen zu U und zu V.

Das Programm endet, sobald zwei aufeinander folgenden Terme S_{2n+1} und S_{2n}, die den Grenzwert einschachteln, einander nahe genug gekommen sind, der Näherungsfehler (ohne Beachtung des Rundungsfehlers) beträgt dann $0 \leq a_{2n+1} = S_{2n} - S_{2n+1} \leq 10^{-6}$.

Die Programmierung dieser fünf alternierenden Reihen geht ähnlich:

$$\sum_{n\geq 2} \frac{(-1)^n}{\ln n}, \quad \sum_{n\geq 1} \frac{(-1)^n}{n}, \quad \sum_{n\geq 1} \frac{(-1)^n}{n^2},$$
$$\sum_{n\geq 1} \frac{(-1)^n}{n^4}, \quad \sum_{n\geq 1} (-1)^n e^{-n\ln n} = \sum_{n\geq 1} \frac{(-1)^n}{n^n}.$$

Die allgemeinen Terme dieser Reihen tendieren mehr oder weniger schnell gegen 0, und die Annäherungen an die Grenzwerte erforden je nach Fall mehr oder weniger Rechnungen.

Die Suche nach den Grenzwerten dieser Reihen mit einer Genauigkeit von 3, 10, 20 oder 100 Dezimalstellen besteht in der Lösung folgender Ungleichungen:

$$\begin{aligned}
1/\ln n &\leq 10^{-3} \iff n \geq e^{(10^3)} \approx 1.97 \cdot 10^{434} \\
1/n &\leq 10^{-3} \iff n \geq 10^3 \qquad & 1/n &\leq 10^{-10} \iff n \geq 10^{10} \\
1/n^2 &\leq 10^{-3} \iff n \geq \sqrt{10^3} \approx 32 \qquad & 1/n^2 &\leq 10^{-10} \iff n \geq 10^5 \\
1/n^4 &\leq 10^{-3} \iff n \geq (10^3)^{1/4} \approx 6 \qquad & 1/n^4 &\leq 10^{-10} \iff n \geq 317 \\
e^{-n\ln n} &\leq 10^{-3} \iff n \geq 5 \qquad & e^{-n\ln n} &\leq 10^{-10} \iff n \geq 30
\end{aligned}$$

$$\begin{aligned}
1/n^2 &\leq 10^{-20} \iff n \geq 10^{10} \qquad & 1/n^2 &\leq 10^{-100} \iff n \geq 10^{50} \\
1/n^4 &\leq 10^{-20} \iff n \geq 10^5 \qquad & 1/n^4 &\leq 10^{-100} \iff n \geq 10^{25} \\
e^{-n\ln n} &\leq 10^{-20} \iff n \geq 17 \qquad & e^{-n\ln n} &\leq 10^{-100} \iff n \geq 57
\end{aligned}$$

In den einfachsten Fällen bestimmt die Lösung dieser Ungleichungen einen Index n, ab dem sich der Wert S_n dem Grenzwert l der Reihe annähern darf, ebenso ist eine `for`-Schleife möglich. Im Gegensatz dazu ist eine `while`-Schleife nötig, sobald sich die algebraische Lösung der Ungleichung in n $a_n \leq 10^{-p}$ als unmöglich erweist.

Bestimmte Näherungslösungen für die vorstehenden Grenzwerte erforden zu viele Rechnungen, um direkt erhalten zu werden, insbesondere sobald der Index n eine Größenordnung von 10^{10} oder 10^{12} überschreitet. Ein vertieftes Studium der Mathematik kann zuweilen die Bestimmung des Grenzwertes oder eine Näherung mit anderen Verfahren ermöglichen, darunter Riemann-Reihen:

$$\lim_{n\to+\infty} \sum_{k=1}^{n} \frac{(-1)^k}{k^3} = -\frac{3}{4}\zeta(3), \qquad \text{mit } \zeta(p) = \lim_{n\to+\infty} \sum_{k=1}^{n} \frac{1}{k^p},$$
$$\lim_{n\to+\infty} \sum_{k=1}^{n} \frac{(-1)^k}{k} = -\ln 2, \qquad \lim_{n\to+\infty} \sum_{k=1}^{n} \frac{(-1)^k}{k^2} = -\frac{\pi^2}{12},$$
$$\lim_{n\to+\infty} \sum_{k=1}^{n} \frac{(-1)^k}{k^4} = -\frac{7\pi^4}{6!}.$$

Außerdem kann Sage einige solcher Gleichungen und einen Näherungswert von $\zeta(3)$ nit 1200 Dezimalstellen in wenigen Sekunden mit viel weniger als 10^{400} Operationen berechnen als bei direkter Anwendung der Definition nötig wären:

```
sage: k = var('k'); sum((-1)^k/k, k, 1, +oo)
-log(2)
sage: sum((-1)^k/k^2, k, 1, +oo), sum((-1)^k/k^3, k, 1, +oo)
(-1/12*pi^2, -3/4*zeta(3))
sage: -3/4*zeta(N(3, digits=1200))
-0.90154267736969571404980362113358749307373971925537416134\
4203666506378654339734817639841905207001443609649368346445539\
5638689969990049624103322976279059251210904563372120200500 39\
...
0199954926528892970698040801518083359081534373107053599192 71\
7989701514061635603285245024246050605197744213902891450545 38\
9019612163591468378139165980642866722553438177035397601703062 62
```

3.2.2. Bedingungen

Weitere wichtige zusammengesetze Anweisungen stellen die Bedingungen dar: welche Anweisung ausgeführt wird, hängt vom Wahrheitswert einer Bedingung ab. Die Struktur und zwei mögliche Formen dieser Anweisung sind die folgenden:

`if` <Bedingung>:	`if` <Bedingung>:
ein Anweisungsblock	ein Anweisungsblock
	`else:` sonst ein anderer Anweisungsblock

Die Syrakuse-Folge (Collatz-Problem) ist mit einer Prüfung auf Parität definiert:

$$u_0 \in \mathbb{N}^* \qquad u_{n+1} = \begin{cases} u_n/2 & \text{falls } u_n \text{ gerade ist,} \\ 3u_n + 1 & \text{falls } u_n \text{ ungerade ist.} \end{cases}$$

Die „tschechische Vermutung“ besagt - seit 2012 bekannt und unbewiesen - dass für jeden Anfangswert $u_0 \in \mathbb{N}^*$ eine Ordnungszahl n existiert, für die $u_n = 1$ wird. Die folgenden Terme sind dann 4, 2, 1, 4, 2 usw. Die Berechnung jedes Terms der Folge erfolgt vermittels der Prüfung einer Bedingung. Diese Prüfung wird im Inneren einer `while`-Schleife angeordnet, die den kleinsten Wert von $n \in \mathbb{N}$ bestimmt, für den $u_n = 1$ wird:

```
sage: u = 6; n = 0
sage: while u != 1:      # eine Prüfung mit <> ("ungleich") ist auch möglich
....:     if u % 2 == 0: # der Operator % ergibt den Rest der Ganzzahldivision
....:         u = u//2   # //: Quotient der Ganzzahldivision
....:     else:
....:         u = 3*u+1
....:     n = n + 1
sage: n
8
```

Die Prüfung, ob u_n gerade ist, erfolgt durch Vergleich des Restes der Division von u_n durch 2 mit 0. Die Anzahl der ausgeführten Iterationen ist der Wert der Variablen n am Ende des

Blocks. Diese Schleife endet, sobald der berechnete Wert von $u_n = 1$ ist. Wenn u_0 beispielsweise 6 ist, wird $u_8 = 1$ und $8 = \min\{p \in \mathbb{N}^* \mid u_p = 1\}$:

p	$= 0$	1	2	3	4	5	6	7	8	9	10 ...
u_p	$= 6$	3	10	5	16	8	4	2	1	4	2 ...

Die schrittweise Verifikation der korrekten Arbeitsweise dieser Zeilen kann durch einen *print-Spion* der Form `print(u, n)` im Schleifenrumpf geleistet werden,

Der Befehl `if` erlaubt außerdem, Prüfungen mit Hilfe des Wortes `elif` in den `else`-Zweig zu legen. Diese beiden Strukturen sind gleichwertig:

```
if eine Bedingung bed1:                    if bed1:
      eine Anweisungsfolge anw1                anw1
else:                                      elif bed2:
      if eine Bedinging bed2:                  anw2
            eine Anweisungsfolge anw2      elif cond3:
      else:                                    anw3
            if eine Bedingung cond3:       else:
                  eine Anweisungsfolge anw3    anwn
            else:
                  sonst anwn
```

Wie bei Schleifen kann eine einfache Anweisung neben dem Doppelpunkt angeordnet werden und nicht als Block darunter.

3.2.3. Prozeduren und Funktionen

Allgemeine Syntax. Wie andere Programmiersprachen auch erlaubt Sage dem Anwender, Funktionen oder Prozeduren *nach Maß* zu definieren. Der Befehl `def`, dessen Syntax hierunter zu erkennen ist, eröffnet die Definition von *Prozeduren* und *Funktionen*, d.h. von Unterprogrammen (die kein Ergebnis zurückgeben bzw. genau dies tun), ohne Argument oder mit einem oder mehreren Argumenten. Das erste Beispiel deklariert die Funktion $(x, y) \mapsto x^2 + y^2$:

```
sage: def fct2 (x, y):
....:     return x^2 + y^2
sage: a = var('a')
sage: fct2 (a, 2*a)
5*a^2
```

Die Auswertung der Funktion wird durch den Befehl `return` beendet. Ihr Ergebnis ist hier $x^2 + y^2$.

Auf dieselbe Weise wird eine Funktion definiert, die nicht explizit ein Resultat zurückgibt, und bei Abwesenheit der Anweisung `return` wird der Funktionsrumpf bis zum Ende abgearbeitet. Dann wird der Wert `None` zurückgegeben.

In der Voreinstellung betrachtet Sage alle Variablen, die in der Funktion auftreten, als lokale Variablen. Diese Variablen werden bei jedem Aufruf der Funktion neu erzeugt und am

Ende wieder zerstört. Sie sind unabhängig von anderen Variablen gleichen Namens, die außerhalb der Funktion existieren mögen. Globale Variablen werden durch Zuweisungen an lokale Variablen innerhalb der Funktion nicht verändert:

```
sage: def versuch(u)
....:     t = u^2
....:     return t*(t+1)
sage: t = 1; u = 2
sage: versuch(3), t, u
(90, 1, 2)
```

Um in einer Funktion eine globale Variable zu verändern, muss man sie mit dem Schlüsselwort `global` explizit deklarieren:

```
sage: a = b = 1
sage: def f(): global a; a = b = 2
sage: f(); a, b
(2,1)
```

Das folgende Beispiel wiederholt die Berechnung des arithmetisch-harmonischen Mittels zweier positiver Zahlen:

```
sage: def MittAH(u, v):
....:     u, v = min(u, v), max(u, v)
....:     while v-u > 2.0e-8:
....:         u, v = 2*u*v/(u+v), (u+v)/2
....:     return (u+v)/2

sage: MittAH(1., 2.)
1.41421...
sage: MittAH          # entspricht einer Funktion
   <function MittAH at ...>
```

Die Funktion `MittAH` erhält die beiden Parameter `u` und `v`, die lokal sind und deren Anfangswerte mit dem Aufruf der Funktion bestimmt sind; beispielsweise beginnt `MittAH` die Ausführung dieser Funktion mit den Werten 1. und 2. der Variablen `u` und `v`.

Bei strukturierter Programmierung soll eine Funktion so geschrieben werden, dass der Befehl `return` die letzte Anweisung des Funktionsrumpfes ist. In seinem Inneren plaziert beendet der Befehl `return` die Ausführung der Funktion und verhindert so die vollständige Ausführung dieses Blocks. Außerdem können verschiedene Zweige von `if - elif - else` - Konstruktionen jeweils mit `return` enden. Die Übertragung des Geistes der Mathematik auf die Informatik legt nahe, Funktionen zu programmieren, die zu ihren Argumenten jeweils ein Resultat zurückgeben und dies nicht etwa durch einen `print`-Befehl bewerkstelligen. Das Programmpaket Sage verfügt zudem über eine Vielzahl von Funktionen, beispielsweise `exp` oder `solve`, die alle ein Ergebnis zurückgeben, eine Zahl zum Beispiel, einen Ausdruck, eine Liste von Lösungen usw.

Iterative und rekursive Verfahren. Eine vom Anwender definierte Funktion ist wie eine Folge von Anweisungen aufgebaut. Eine Funktion heißt rekursiv, wenn zu ihrer Auswertung in bestimmten Fällen erforderlich ist, diese Funktion mit verschiedenen Parametern mehrmals auszuführen. Die Fakultät $(n!)_{n\in\mathbb{N}}$ ist ein einfaches Beispiel:

$$0! = 1, \qquad (n+1)! = (n+1)n! \quad \text{für alle } n \in \mathbb{N}.$$

Die beiden folgenden Funktionen einer natürlichen Zahl liefern dasselbe Ergebnis, aber die erste arbeitet iterativ und verwendet eine `for`-Schleife, während die andere die vorstehende rekursive Definition *wortwörtlich* umsetzt:

```
sage: def fact1(n):
....:     res = 1
....:     for k in [1..n]: res = res*k
....:     return reset

sage: def fact2(n):
....:     if n == 0: return 1
....:     else: return n*fact2(n-1)
```

Die Fibonacci-Folge ist eine rekursive Folge 2. Ordnung, denn der Wert u_{n+2} hängt eindeutig von den Werten u und u_{n+1} ab:

$$u_0 = 0, \qquad u_1 = 1, \qquad u_{n+2} = u_{n+1} + u_n \text{ für alle } n \in \mathbb{N}.$$

Die Funktion `fib1` hierunter verwendet bei der Berechnung der Glieder der Fibonaccifolge ein iteratives Verfahren und arbeitet mit zwei Variablen `U` und `V`, um die beiden vorhergehenden Werte der Folge zwischenzuspeichern, bevor zum nächsten gegangen wird:

```
sage: def fib1(n):
....:     if n == 0 or n == 1: return n
....:     else:
....:         U = 0
....:         V = 1   # die Startwerte u0 und u1
....:         for k in [2..n]:
....:             W = U + V
....:             U = V
....:             V = W
....:         return V
sage: fib1(8)
21
```

Ab $n = 2$ verarbeitet die Schleife die Beziehung $u_n = u_{n-1} + u_{n-2}$. Des weiteren vermeidet die parallele Zuweisung von `U,V = V,U+V` statt `w = U+V; U = V; V = W` die Einführung einer Variablen `W` und übersetzt die Iteration der rekursiven vektoriellen Folge $X_n = (u_n, u_{n+1})$ 1. Ordnung, die durch $X_{n+1} = f(X_n)$ definiert ist, wenn $f(a, b) = (b, a + b)$ ist. Die iterativen Verfahren sind effizient, doch muss bei ihrer Programmierung die Definition der Folge an die Manipulation der Variablen angepasst werden

Im Gegensatz dazu folgt die rekursive Funktion `fib2` der mathematischen Definition dieser Folge viel besser, was ihre Programmierung vereinfacht und das Verständnis verbessert:

```
sage: def fib2(n):
....:     if 0 <= n <= 1: return n      # für n = 0 oder n = 1
....:     else: return fib2(n-1) + fib2(n-2)
```

Das Ergebnis dieser Funktion ist der von der bedingten Anweisung zurückgegebene Wert: 0 und 1 für $n = 0$ bzw. $n = 1$ und sonst die Summe `fib2(n-1)+fib2(n-2)`. Jeder Zweig des Tests endet mit dem Befehl `return`.

Dieses Verfahren ist zwar weniger effizient, denn viele Rechnungen werden unnötigerweise wiederholt. Zum Beispiel werden für `fib2(5)` auch `fib2(3)` und `fib2(4)` ausgewertet, die ebenso berechnet worden sind. Somit wertet Sage `fib2(3)` zweimal aus und `fib2(2)` dreimal. Dieser Prozess endet mit der Auswertung von `fib2(1)` oder `fib2(0)` und die Auswertung von `fib2(n)` besteht schließlich aus der Addition von u_n Einsen und u_{n-1} Nullen. Die Gesamtzahl der Additionen ist daher gleich $u_{n+1} - 1$. Diese Zahl ist von beträchtlicher Größe und wächst sehr schnell. Kein Rechner, und sei er noch so schnell, kann auf diese Weise u_{100} berechnen.

Auch andere Verfahren sind möglich, beispielsweise mit Zwischenspeicherung dank dem Dekorator `@cached_function` oder durch Ausnutzung einer Eigenschaft von Matrixpotenzen: der folgende Abschnitt zum schnellen Potenzieren zeigt, wie der millionste Term dieser Folge zu berechnen ist.

3.2.4. Beispiel: schnelles Potenzieren

Eine naive Methode zur Berechnung von a^n besteht darin, in einer `for`-Schleife $n \in \mathbb{N}$ Multiplikationen mit a auszuführen.

```
sage: a = 2; n = 6; res = 1    # 1 ist das neutrale Element
sage: for k in [1..n]: res = res*a
sage: res     # Der Wert von res ist 2^6
64
```

Ganzzahlige Potenzen gibt es in der Mathematik in drei Zusammenhängen: dieser Abschnitt untersucht eine allgemeine Methode der Berechnung einer ganzzahligen Potenz a^n, die schneller ist als die vorstehende. Die hiernach definierte Folge $(u_n)_{n\in\mathbb{N}}$ verifiziert $u_n = a^n$; man beweist dieses Resultat durch Rekursion mittels dieser Gleichungen $a^{2k} = (a^k)^2$ und $a^{k+1} = aa^k$:

$$u_n = \begin{cases} 1 & \text{für } n = 0, \\ u_{n/2}^2 & \text{für gerades } n > 0, \\ au_{n-1} & \text{für ungerades } n. \end{cases} \tag{3.1}$$

Beispielsweise ist für u_{11}:

$$u_{11} = au_{10}, \quad u_{10} = u_5^2, \quad u_5 = au_4, \quad u_4 = u_2^2,$$
$$u_2 = u_1^2, \quad u_1 = au_0 = a;$$

und somit:

$$u_2 = a^2, \quad u_4 = u_2^2 = a^4, \quad u_5 = aa^4 = a^5,$$
$$u_{10} = u_5^2 = a^{10}, \quad u_{11} = aa^{10} = a^{11}.$$

Die Berechnung von u_n lässt nur Terme u_k mit $k \in \{0, \ldots, n-1\}$ auftreten und ist deshalb mit einer endlichen Anzahl von Operationen gut beschrieben.

Dieses Beispiel zeigt außerdem, dass man den Wert von u_{11} durch Auswertung der 6 Terme u_{10}, u_5, u_4, u_2, u_1 und u_0 erhält, was einzig mit 6 Multiplikationen erfolgt. Die Berechnung von u_n erfordert zwischen $\log n/\log 2$ und $2\log n/\log 2$ Multiplikationen, denn nach einem oder zwei Schritten - jenachdem ob n gerade ist oder ungerade - wird u_n als Funktion von u_k ausgedrückt mit $k \leq n/2$. Dieses Verfahren ist daher unvergleichlich schneller als die naive Methode, sobald n groß ist; etwa 20 Terme für 10^4 und keine 10^4 Produkte:

verarbeitete Indizes:	10000	5000	2500	1250	625	624	312	156	78
	39	38	19	18	9	8	4	2	1

Allerdings ist dieses Verfahren nicht immer das schnellste; die folgende Rechnung mit b, c, d und f braucht 5 Produkte, um a^{15} zu berechnen, obwohl das Verfahren mit u, v, w, x und y 6 Multiplikationen erfordert ohne die Multiplikation mit 1:

$$b = a^2 \quad c = ab = a^3 \quad d = c^2 = a^6 \quad f = cd = a^9 \quad df = a^{15} : 5 \text{ Produkte;}$$
$$u = a^2 \quad v = au = a^3 \quad w = v^2 = a^6 \quad x = aw = a^7 \quad y = x^2 = a^14 \quad ay = a^{15} : 6 \text{ Produkte.}$$

Die rekursive Funktion `pot1` berechnet die rekursive Folge (3.1) nur mit den Multiplikationen zur Programmierung des Operators für die Erhebung zur Potenz:

```
sage: def pot1(a, n):
....:     if n == 0: return 1
....:     elif n % 2 == 0: b = pot1(a, n//2); return b*b
....:     else: return a*pot1(a, n-1)

sage: pot1(2,11)    # a für das Resultat 2^11
2048
```

Die Anzahl der von dieser Funktion ausgeführten Operationen ist die gleiche wie die von Hand berechnete, wenn die schon berechneten Resultate weiterverwendet werden. Wenn andererseits die Anweisungen `b = pot1(a, n//2); return b*b`, die nach der Prüfung von n auf Geradheit ausgeführt werden, durch `pot1(a, n//2)*pot1(a, n//2)` ersetzt würden, müsste Sage weit mehr Rechnungen ausführen, weil wie bei der rekursiven Funktion `fib2`, welche die Fibonacci-Folge berechnet, etliche Berechnungen unnötigerweise wiederholt werden würden. Es gäbe letzten Endes n Multiplikationen, soviele wie mit dem naiven Verfahren.

Außerdem kann der Befehl `return pot1(a*a, n//2)` auf äquivalente Weise diese beiden Anweisungen ersetzen.

Das folgende Programm führt die gleiche Rechnung iterativ aus:

```
sage: def pot2(u, k):
....:     v = 1
....:     while k != 0:
....:         if k % 2 == 0: u = u*u; k = k//2
....:         else: v = v*u; k -= 1
....:     return v

sage: pot2(2, 10)        # a für das Resultat 2^10
1024
```

Dass der Wert von `pot2(a, n)` tatsächlich a^n ist, wird dadurch bewiesen, dass die Variablen u, v und k bei jedem Schritt durch die Gleichung $vu^k = a^n$ verbunden sind, egal, ob k gerade oder ungerade ist. Im ersten Schritt ist $v = 1$, $u = a$ und $k = n$; nach dem letzten Schritt ist $k = 0$, also $v = a^n$.

Die aufeinander folgenden Werte von k sind sämtlich ganz und ≥ 0, und sie bilden eine streng fallende Folge. Diese Variable kann nur eine endliche Anzahl von Werten annehmen, bevor sie 0 wird und die Schleife endet.

Entgegen dem Augenschein - die Funktion `pot1` ist rekursiv, die Funktion `pot` iterativ programmiert - geben beide Funktionen nahezu den gleichen Algorithmus wieder: der einzige Unterschied ist, dass die erste a^{2k} durch $(a^k)^2$ auswertet, die zweite jedoch a^{2k} durch $(a^k)^2$, wie die Modifikation der Variablen u zeigt.

Dieses Verfahren ist nicht auf ganzzahlige Potenzen von Zahlen durch Multiplikation beschränkt, sondern wird auf jedes Gesetzt der internen assoziativen Komposition angewendet. Dieses Gesetz muss assoziativ sein, um die üblichen Eigenschaften bei mehrmaliger Multiplikation zu besitzen. Ersetzte man also die Zahl 1 durch die Einheitsmatrix $\mathbb{I}_n$, würden beide Funktionen die positiven Potenzen quadratischer Matrizen auswerten. Diese Funktionen verdeutlichen, wie der Potenz-Operator „ `^` “ mittels Multiplikationen effizient zu programmieren ist, und sie ähneln dem in Sage implementierten Verfahren.

Beispielsweise ermöglicht eine Matrizenpotenz Terme der Fibonacci-Folge mit Indizes zu erhalten, die noch größer sind als vorhin:

$$A = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}, \qquad X_n = \begin{pmatrix} u_n \\ u_{n+1} \end{pmatrix}, \qquad AX_n = X_{n+1}, \qquad A^n X_0 = X_n.$$

Das entsprechende Sage-Programm gibt sich mit zwei Zeilen zufrieden und das gesuchte Resultat ist der erste Eintrag des Matrizenproduktes $A^n X_0$, was selbst für $n = 10^7$ funktioniert; diese beiden Programme sind äquivalent und ihre Effizienz kommt daher, dass Sage eine Methode der schnellen Potenzierung verwendet:

```
sage: def fib3(n):
....:     A = matrix([[0,1],[1,1]]); X0 = vector([0,1])
....:     return (A^n*X0)[0]

sage: def fib4(n):
....:     return (matrix([[0,1],[1,1]])^n*vector([0,1]))[0]
```

3.2.5. Datenein- und -ausgabe

Die Anweisung `print` ist *der* Befehl zur Datenausgabe. Voreingestellt ist die Ausgabe der Argumente eines nach dem anderen und getrennt durch ein Leerzeichen; am Ende des Befehls macht Sage automatisch einen Zeilenvorschub:

```
sage: print 2^2, 3^3, 4^4; print 5^5, 6^6
4 27 256
3125 46656
```

Ein Komma am Ende des Befehls `print` unterdrückt den Zeilenvorschub, und die nächste `print`-Anweisung wird auf der selben Zeile fortgesetzt:

```
sage: for k in [1..10]: print '+', k,
+ 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10
```

Es ist möglich, die Resultate ohne Zwischenraum auszugeben, indem sie mit der Funktion `str()` in eine Zeichenkette umgewandelt und durch den Verkettungsoperator „ + “ verbunden werden.

```
sage: print 10, 0.5; print 10+0.5; print 10.0, 5
10 0.500000000000000
```

```
10.5000000000000
10.0000000000000 5
sage: print 10+0, 5; print str(10)+str(0.5)
10 5
100.500000000000000
```

Der letzte Abschnitt dieses Kapitels über Listen und andere Datenstrukturen stellt Zeichenketten dann ausführlicher vor.

Der Befehl `print` ermöglicht auch Zahlen zu formatieren, um sie in eine Tabelle zu schreiben. Das folgende Beispiel mit dem Platzhalter `%.d` und dem Operator `%` gibt die Tabelle der 4. Potenzen eine untereinander aus: `sage: for k in [1..6]: print '%2d^4 = %4d' % (k, k^4)`

```
 1^4 =    1
 2^4 =   16
 3^4 =   81
 4^4 =  256
 5^4 =  625
 6^4 = 1296
```

Der Operator `%` setzt die Zahlen zu seiner Rechten in die Zeichenkette zu seiner Linken an der durch `%2d` und `%4d` markierten Stelle ein. Dabei sorgt zum Beispiel `%4d` dafür, dass für die Zahl mindestens 4 Stellen reserviert werden, um noch für `6^4` Platz zu haben. Genauso gibt der Platzhalter `%.4f` in `'pi = %.4f' % n(pi)` die Zeichenkette `pi = 3.1416` mit vier Stellen nach dem Dezimaltrenner aus.

Auf einem Terminal gibt die Funktion `raw_input('message')` den Text `message` aus, erwartet eine gültige Eingabe von der Tastatur, die mit einem Zeilenvorschub abgeschlossen wird und gibt die entsprechende Zeichenkette zurück.

3.3. Listen und zusammengesetzte Strukturen

Dieser Abschnitt behandelt die zusammengesetzten Datenstrukturen in Sage: Zeichenketten, Listen - veränderlichen oder unveränderlichen Typs -, Mengen und Diktionäre.

3.3.1. Definition von Listen und Zugriff auf die Elemente

Der Begriff der Liste ist in der Informatik das n-Tupel der Mathematik und ermöglicht die Aufzählung mathematischer Objekte. Der Begriff des Paares - mit $(a, b) \neq (ba)$ - und des n-Tupels präzisiert im Gegensatz zum Begriff der Menge die Position jedes Elementes.

Wir definieren eine Liste, indem wir ihre Elemente getrennt durch Kommata in eckige Klammern setzen [...]. Die Zuweisung des Tripels $(10, 20, 30)$ zur Variablen L wird auf folgende Weise vorgenommen:

```
sage: L = [10, 20, 30]
sage: L
[10, 20, 30]
```

und die leere Liste, ohne Elemente, wird einfach so definiert:

```
sage: []            # das ist die leere Liste
[]
```

Die Listenelemente werden durch ganze Zahlen indiziert. Das erste Element hat den Index 0, das zweite den Index 1 usw. Der Zugriff auf das Element mit dem Index k einer Liste L erfolgt durch $L[k]$, mathematisch entspricht das der kanonischen Projektion der betrachteten Liste als Element eines kartesischen Produkts auf das Element der Ordnung k. Die Funktion `len` gibt die Anzahl der Elemente einer Liste zurück[6]:

```
sage: L[1], len(L), len([])
(20, 3, 0)
```

Die Modifikation eines Eintrags gelingt auf die gleiche Weise, nämlich durch Zuweisung des entsprechen Eintrags. So verändert der folgende Befehl den mit 2 indizierten dritten Term der Liste:

```
sage: L[2] = 33
sage: L
[20, 3, 33]
```

Mit negativen Indizes wird von hinten auf Listenelemente zugegriffen, wobei das letzte Element den Index -1 hat, das vorletzte den Index -2 usw.

```
sage: L = [11, 22, 33]
sage: L[-1], L[-2], L[-3]
(33,22,11)
```

Der Befehl `L[p:q]` extrahiert die Teilliste `[L[p], L[p+1], ..., L[q-1]]`, welche für $q \leq p$ leer ist. Negative Indizes ermöglichen das Herausziehen der hinteren Listenelemente; schließlich bildet die Angabe `L[p:]` die Teilliste `L[1:len(L)]`, beginnend beim Index p bis zum Ende, und `L[:p]=L[0:p]` zählt die Elemente der Liste vom Anfang bis vor das Element mit dem Index q auf:

```
sage: L = [0, 11, 22, 33, 44, 55]
sage: L[2:4]
22, 33
sage: L[-4:4]
[22, 33]
sage: L[2,-2]
22,33 sage: L[:4]
[0, 11, 22, 33] sage: L[4:]
[44,55]
```

Genauso wie der Befehl `L[n] = ...` ein Element der Liste verändert, ersetzt die Zuweisung `L[p:q] 0 ...` die Teilliste zwischen den Elementen p (einschließlich) und q (ausschließlich):

```
sage: L = [0, 11, 22, 33, 44, 55, 66, 77]
sage: L[2:6] = [12, 13, 14]      # ersetzt [22, 33, 44, 55]
```

[6] Der Wert der Funktion `len` hat den Typ `int` von Python, der Ausdruck `Integer(len(...))` gibt eine ganze Zahl des Typs `Integer` von Sage zurück.

So unterdrücken `L[:1] = []` und `L[-1:]` jeweils das erste bzw. das letzte Element einer Liste und umgekehrt setzen `L[:0] = [a]` und `L[len(L):] = [a]` jeweils ein Element a am Anfang bzw. am Ende der Liste ein. Generell erfüllen die Einträge einer Liste diese Gleichungen:

$$\begin{aligned} \mathrm{L} &= [l_0, l_1, l_2, \cdots, l_{n-1}] = [l_{-n}, l_{1-n}, \ldots, l_{-2}, l_{-1}] \quad \text{mit n = len(L)}, \\ l_k &= l_{k-n} \text{ für } 0 \le k < n, \quad l_j = l_{n+j} \text{ für } -n \le j < 0. \end{aligned}$$

Der Operator `in` testet, ob ein Element in einer Liste vorhanden ist. Sage prüft zwei Listen auf Gleichheit mit „ == “ und vergleicht die Elemente eins nach dem anderen. Die beiden Teillisten mit positiven bzw. negativen indizes sind gleich:

```
sage: L = [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
sage: L[3:len(L)-5] == L[3-len(L):-5]
True
sage: [5 in L, 6 in L]
[True, False]
```

Die obigen Beispiele betreffen Listen mit ganzen Zahlen, doch können beliebige Sage-Objekte Elemente von Listen sein, Zahlen, Ausdrücke, andere Listen usw.

3.3.2. Globale Operationen auf Listen

Der Additionsoperator „ + “ bewirkt die Verkettung zweier Listen, und der Multiplikationsoperator „ * “ bewirkt ein ganzzahliges Vielfaches dieser Verkettung:

```
sage: L = [1, 2, 3]; L + [10, 20, 30]
[1, 2, 3, 10, 20, 30]
sage: 4*[1, 2, 3]
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Die Verkettung der beiden Teillisten `L[:k]` und `L[k:]` bringt die ursprüngliche Liste zurück. Das erklärt, eshalb die linke Grenze mit dem Index `p` einer Teilliste `L[p:q]` eingeschlossen ist, die rechte Grenze `q` aber nicht:

$$\begin{aligned} \mathtt{L} = \mathtt{L[:k]} + \mathtt{L[k:]} &= [l_0.l_1, \ldots, l_{n-1}] \\ &= [l_0, l_1, l_2, \ldots, l_{k-1}] + [l_k, l_{k+1}, l_{k+2}, \ldots, l_{n-1}]. \end{aligned}$$

Das folgende Beispiel verdeutlicht diese Eigenschaft:

```
sage: L = 5*[10, 20, 30]; L[:3] + L[3:] == L
True
```

Der aus zwei Punkten zusammengesetzte Operator „ .. “ automatisiert die Erstellung ganzzahliger Listen, ohne ihre Elemente alle einzeln aufzuzählen. Das folgende Beispiel bildet eine Liste, die aus Aufzählungen und aus isolierten Elementen besteht:

```
sage: 1..3, 7, 10..13
[1, 2, 3, 7, 10, 11, 12, 13]
```

Nun wird beschrieben, wie das Bild einer Liste unter einer Funktion erzeugt wird und eine Teilliste einer Liste. Die zugehörigen Funktionen sind `map` und `filter` sowie die Konstruktion

`[.. for .. x .. in .. ]`. In der Mathematik treten oft Listen auf, die aus Bildern unter der Funktion f ihrer Elemente bestehen:

$$(a_0, a_1, a_2, \ldots, a_{n-1}) \mapsto (f(a_0), f(a_1), \ldots, f(a_{n-1})).$$

Der Befehl `map` erzeugt dieses Bild; das folgende Beispiel wendet die trigonometrische Funktion `cos` auf eine Liste häufig gebrauchter Winkel an:

```
sage: map(cos, [0, pi/6, pi/4, pi/3, pi/2])
[1, 1/2*sqrt(3), 1/2*sqrt(2), 1/2, 0]
```

Es ist auch möglich, eine vom Anwender mit `def` definierte Funktion zu verwenden oder mit `lambda` eine Funktion direkt zu deklarieren; der Befehl hierunter ist zum vorhergehenden äquivalent und benutzt die mit $t \mapsto \cos t$ definierte Funktion:

```
sage: map(lambda t: cos(t), [0, pi/6, pi/4, pi/3, pi/2])
[1, 1/2*sqrt(3), 1/2*sqrt(2), 1/2, 0]
```

Der Befehl `lambda` wird gefolgt von einem oder mehreren durch Kommata getrennten Parametern und kann nach dem Doppelpunkt nur genau einen Ausdruck verarbeiten, der ohne `return` das Ergebnis liefert.

Diese `lambda`-Funktion kann zudem einen Test ausführen; die folgenden Funktionen sind äquivalent:

```
fktTest1 = lambda x: res1 if cond else res2
def fktTest2(x):
    if cond; return res1
    else: return res2
```

Die folgenden drei `map`-Befehle sind äquivalent, die Hintereinanderausführung der Funktionen $M \circ \cos$ geschieht auf mehrere Arten:

```
sage: map(lambda t: N(cos(t)), [0, pi/6, pi/4, pi/3, pi/2])
[1.00000000000000, 0.866025403784439,  0.707106781186548,
0.500000000000000, 0.000000000000000]

sage: map(N, map(cos, [0, pi/6, pi/4, pi/3, pi/2]))
[1.00000000000000, 0.866025403784439, 0.707106781186548,
0.500000000000000, 0.000000000000000]

sage: map(compose(N, cos), [0, pi/6, pi/4, pi/3, pi/2])
[1.00000000000000, 0.866025403784439, 0.707106781186548,
0.500000000000000, 0.000000000000000]
```

Der Befehl `filter` erzeugt eine Teilliste der Elemente, die eine Bedingung erfüllen. Dieses Beispiel wendet den Primzahltest `is_prime` auf die Zahlen $1, \ldots, 55$ an:

```
sage: filter(is_prime, [1..55])
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53]
```

Die Test-Funktion kann auch im Inneren des Befehls `filter` definiert werden. Das Beispiel hierunter bestimmt in erschöpfenden Tests alle vierten Wurzeln von 7 modulo der Primzahl 37; diese Gleichung hat die vier Lösungen 3, 18, 19 und 34:

```
sage: p = 37; filter(lambda n: n^4 % p == 7, [0..p-1])
[3, 18, 19, 34]
```

Außerdem bildet der Befehl `[.. for ,, x .. in ..]` eine Listenraffung (Listencomprehension); diese beiden Befehle zählen auf äquivalente Weise die ungeraden Zahlen von 1 bis 31 auf:

```
sage: map(lambda n: 2*n+1, [0..15])
[1, 3, 5, 7, 9. 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31]
sage: [2*n+1 for n in [0..15]]
[1, 3, 5, 7, 9. 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31]
```

Dieser Befehl ist vom Schleifenbefehl `for` unabhängig. Die mit `for` verbundene Bedingung `if` führt zu einer Konstruktion, die zur Funktion `filter` äquivalent ist:

```
sage: filter(is_prime, [1..55])
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53]
sage: p for p in [1..55] if is_prime(p)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53]
```

Die beiden folgenden Beispiele kombinieren die Tests `filter` und `for`, um eine Liste von Primzahlen zu bestimmen, die zu 1 modulo 4 kongruent sind, dann eine Liste der Quadrate von Primzahlen:

```
sage: filter(is_prime, [4*n+1 for n in [0..20]])
[5, 13, 17, 29, 37, 41, 53, 61, 73]
sage: [n^2 for n in [1..20] if is_prime(n)]
[4, 9, 25, 49, 121, 169, 361]
```

Im ersten Fall wird der Test `is_prime` nach der Berechnung von $4n+1$ ausgeführt, während im zweiten Fall der Test vor der Berechnung des Quadrats n^2 ausgeführt wird.

Die Funktion `reduce` bearbeitet die Elemente einer Liste assoziativ von links nach rechts. Definieren wir das Gesetz der internen Komposition mit $\circ$:

$x \circ y = 10x + y$, wird $((1 \circ 2) \circ 3) \circ 4 = (12 \circ 3) \circ 4 = 10 * (10 * (10 * 1 + 2) + 3) + 4 = 1234$.

Das erste Argument von `reduce` ist eine Funktion mit zwei Parametern und das zweite ist die Liste der Argumente:

```
sage: reduce(lambda x, y: 10*x+y, [1, 2, 3, 4])
1234
```

Ein optionales drittes Argument wird vor das Ergebnis gestellt und dient bei einer leeren Liste als Ergebnis:

```
sage: reduce(lambda x, y: 10*x+y, [9, 8, 7, 6], 1)
[19876]
```

Es entspricht allgemein dem neutralen Element der jeweiligen Operation. So berechnet das nächste Beispiel ein Produkt von ungeraden Zahlen:

```
sage: L = [2*n+1 for n in [0..9]]
sage: reduce(lambda x, y: x*y, L, 1)
654729075
```

Sages Funktionen `add`[7] und `prod` verwenden den Operator `reduce` direkt zur Berechnung von Summen und Produkten; das Produkt ist in den drei Beispielen hierunter jeweils das gleiche, und der Befehl mit einer Liste ermöglicht zudem, einen zweiten, optionalen Term als neutrales Element hinzuzufügen, 1 für das Produkt und 0 für die Summe oder eine Einheitsmatrix für ein Matrizenprodukt:

```
sage: prod([2*n+1 for n in [0..9]], 1)  # eine Liste mit for
654729075
sage: prod(2*n+1 for n in [0..9])
654729075
sage: prod(n for n in [0..19] if n%2 == 1)
654729075
```

Die Funktion `any` mit dem Operator `or` und die Funktion `all` mit dem Operator `and` sind grundsätzlich äquivalent. Allerdings endet die Auswertung, sobald das Resultat `True` oder `False` eines der Terme dieses Resultat erheischt, ohne die Auswertung der folgenden Terme zu beeinflussen.

```
sage: def fkt(x): return 4/x == 2
sage: all(fkt(x) for x in [2, 1, 0])
False sage: any(fkt(x) for x in [2, 1, 0])
True
```

Hingegen führen die Bildung der Liste `[fkt(x) for x in [2, 1, 0]]` und der Befehl `all([fkt(x) for x in [2, 1, 0]])` zu Fehlern (`rational division by zero`), denn es werden alle Terme ausgewertet, auch der letzte mit $x = 0$.

Das Verschachteln mehrerer `for`-Befehle ermöglicht die Bildung des kartesischen Produktes zweier Listen oder die Definition einer Liste von Listen. Das folgende Beispiel zeigt, dass wenn wir mehrere Operatoren `for` in derselben Definition in einer Raffung kombinieren, der am weitesten links stehende der äußersten Schleife entspricht:

```
sage: [[x, y] for x in [1..2] for y in [6..8]]
[[1, 6], [1, 7], [1, 8], [2, 6], [2, 7], [2, 8]]
```

Die Reihenfolge der Schritte ist daher von derjenigen verschieden, die man durch Verschachteln mehrerer Definitionen in einer Raffung erhält.

```
sage: [[[x, y] for x in [1..2]] for y in [6..8]]
[[[1, 6], [2, 6]], [[1, 7], [2, 7]], [[1, 8], [2, 8]]]
```

Der Befehl `map` mit mehreren gleichlangen Listen als Argumenten geht in diesen Listen synchronisiert vor.

```
sage: map(lambda x, y: [x, y], [1..3], [6..8])
[[1, 6], [2, 7], [3, 8]]
```

Schließlich erlaubt der Befehl `flatten` Listen auf einer oder mehreren Ebenenen zu verketten:

[7] Nicht zu verwechseln mit `sum`, das für eine Summe einen symbolischen Ausdruck sucht.

```
sage: L = [[1, 2, [3]], [4, [5, 6]], [7, [8, [9]]]]
sage: flatten(L, max_level = 1)
[1, 2, [3], 4, [5, 6], 7, [8, [9]]]
sage: flatten(L, max_level = 2)
[1, 2, 3, 4, 5, 6, 7, 8, [9]]
sage: flatten(L)      # äquivalent zu flatten(L, max_level = 3)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Solche elementaren Manipulationen von Listen erweisen sich in anderen Zweigen von Sage als nützlich; das folgende Beispiel berechnet Ableitungen von xe^x; das erste Argument von `diff` ist der abzuleitende Ausdruck, der oder die folgenden entsprechen der Variablen, nach der abgeleitet werden soll, diese Parameter können auch die Liste der Variablen sein, nach denen abgeleitet werden soll:

```
sage: x = var('x')
sage: factor(diff(x*exp(x), [x, x]))
(x + 2)*e^x
sage: map(lambda n: factor(diff(x*exp(x), n*[x])), [0..6])
[x*e^x, (x + 1)*e^x, (x + 2)*e^x, (x + 3)*e^x, (x + 4)*e^x,
(x + 5)*e^x, (x + 6)*e^x]
sage: [factor(diff(x*exp(x), n*[x])) for n in [0..6]]
[x*e^x, (x + 1)*e^x, (x + 2)*e^x, (x + 3)*e^x, (x + 4)*e^x,
(x + 5)*e^x, (x + 6)*e^x]
```

Der Befehl `diff` besitzt mehrere Syntaxen. Die Parameter, die der Funktion f folgen, können sowohl die Liste der Ableitungsvariablen sein wie auch die Aufzählung dieser Variablen oder der Grad der Ableitung:

```
sage: diff(f(x), x, x, x),    diff(f(x), [x, x, x]),    diff(f(x), x, 3)
```

Wir können auch `diff(f(x), 3)` für Funktionen nur einer Variablen verwenden. Diese Resultate werden durch die Leibnizformel für mehrfache Ableitungen eines Produktes aus zwei Termen unmittelbar bestätigt, wobei die zweiten und höheren Ableitungen nach x null sind:

$$(xe^x)^{(n)} = \sum_{k=0}^{n} \binom{n}{k} x^{(k)} (e^x)^{(n-k)} = (x+n)e^x.$$

3.3.3. Wichtige Methoden auf Listen

Die Methode `reverse` kehrt die Reihenfolge der Elemente einer Liste um, und die Methode `sort` ordnet die Elemente aufsteigend der Größe nach:

```
sage: L = [1. 8, 5, 2, 9]: L.reverse(); L
[9, 2, 5, 8, 1]
sage: L.sort(); L
1, 2, 5, 8, 9
sage: L.sort(reverse = True); L
[9, 8, 5, 2, 1]
```

Diese beiden Methoden verändern die Liste `L`, und der alte Wert ist verloren.

Ein erstes optionales Argument bei `sort` ermöglicht die Auswahl der Ordnungsrelation in der Form einer Funktion `Ordre(x, y)` mit zwei Parametern. Das Resultat muss vom Typ `int` von Python sein; es ist zum Beispiel -1, 0 oder 1, je nachdem ob $x \prec y$, $x = y$ oder $x \succ y$. Die transformierte Liste $(x_0, x_1, \ldots, x_{n-1})$ erfüllt $x_0 \preceq x_1 \preceq \ldots \preceq x_{n-1}$.

Die lexikographische Ordnung der beiden Zahlenlisten gleicher Länge ist ähnlich der alphabetischen Ordnung und wird durch diese Äquivalenz definiert, wobei ersten Terme vernachlässigt werden, sobald sie paarweise gleich sind:

$$\begin{aligned} &P = (p_0, p_1, \ldots, p_{n-1}) \prec_{textlex} Q = (Q_0, Q_1, \ldots, q_{n-1}) \\ \Longleftrightarrow\ &\exists r \in \{0, \ldots, n-1\} \quad (p_0, p_1, \ldots, p_{r-1}) = (q_0, q_1, \ldots, q_{r-1}) \text{ und } p_r < q_r. \end{aligned}$$

Die folgende Funktion vergleicht zwei Listen, die gleich lang sein müssen. Trotz der Endlosschleife `while True` verlassen die Befehle `return` diese Schleife direkt und beenden die Funktion. Das Ergebnis ist -1, 0 oder 1 je nachdem, ob $P \prec_{\text{lex}} Q$, $P = Q$ oder $P \succ_{\text{lex}} Q$:

```
sage: def alpha(P, Q):      # len(P) = len(Q) vorausgesetzt
....:     i = 0
....:     while True:
....:         if i == len(P): return int(0)
....:         elif P[i] < Q[i]: return int(-1)
....:         elif P[i] > Q[i]: return int(1)
....:         else: i = i+1
sage: alpha([2, 3, 4, 6, 5], [2, 3, 4, 5, 6])
1
```

Der folgende Befehl ordnet eine Liste von gleich langen Listen in lexikographischer Reihenfolge. Diese Funktion entspricht außerdem der in Sage für den Vergleich zweier Listen implementierten Ordnung; der Befehl `L.sort()` ohne optionalen Parameter ist äquivalent:

```
sage: L = [[2, 2, 5], [2, 3, 4], [3, 2, 4], [3, 3, 3], [1, 1, 2], [1, 2, 7]]
sage: L.sort(cmp=alpha); L
[[1, 1, 2], [1, 2, 7], [2, 2, 5], [2, 3, 4], [3, 2, 4], [3, 3, 3]]
```

Die Definition der homogenen lexikographischen Ordnung besteht zunächst im Vergleich der Terme nach ihrem Gewicht vor Anwendung der lexikographischen Ordnung, wobei das Gewicht die Summe der Koeffizienten ist:

$$\begin{aligned} &P = (p_0, p_1, \ldots, p_{n-1}) \prec_{\text{lexH}} Q = (q_0, q_1, \ldots, q_{n-1}) \\ \Longleftrightarrow\ &\sum_{k=0}^{n-1} p_k < \sum_{k=0}^{n-1} q_k \text{ oder } \left(\sum_{k=0}^{n-1} p_k = \sum_{k=0}^{n-1} q_k \text{ und } P \prec_{\text{lex}} Q \right). \end{aligned}$$

Der Code hierunter implementiert diese homogene Ordnung:

```
sage: def homogLex(P. Q):
....: sp = sum(P); sq = sum(Q)
....: if sp < sq: return int(-1)
....: elif sp > sq: return int(1)
....: elde: return alpha(P, Q)

sage: homogLex([2, 3, 4, 6, 4], [2, 3, 4, 5, 6])
-1
```

Sages Funktion `sorted` ist eine Funktion im mathematischen Sinnes des Wortes; sie erhält als erstes Argument eine Liste und gibt, ohne sie zu verändern, als Resultat eine neue, geordnete Liste zurück. Im Unterschied dazu ordnet die Methode `sort` die Liste in place.

Sage bietet noch andere Methoden für Listen, Einfügen eines Elements am Ende der Liste und Abzählen der Anzahl der Wiederholungen eines Elements:

```
L.append(x)     ist äquivalent zu L[len(L):] = [x]
L.extend(L1)    ist äquivalent zu L[len(L):] = L1
L.insert(i, x) ist äquivalent zu L[i:i] = [x]
L.cont(x)       ist äquivalent zu len(select(lambda t: t == x, L))
```

Die Befehle `L.pop(i)` und `L.pop()` löschen das mit i indizierte Objekt oder das letzte Element einer Liste und geben dieses Element zurück; die beiden Funktionen hierunter beschreiben jeweils ihre Wirkungsweise:

```
def pop1(L, i):
    a = L[i]
    L[i:i+1] = []
    return a
```

```
def pop2(L):
    return pop1(L, len(L)-1)
```

Außerdem gibt `L.index(x)` den Index des ersten Terms zurück, der x gleicht, und `L.remove(x)` entfernt das erste Element mit dem Wert x aus der Liste. Diese Befehle führen zu einem Fehler, wenn x nicht in der Liste vorhanden ist. Schließlich ist der Befehl `del L[p:q]` äquivalent zu `L[p:q] = []`, und `del L[i]` löscht das Element mit dem Index i.

Anders als in zahlreichen anderen Programmiersprachen modifizieren diese Funktionen die Liste `L`, ohne eine neue Liste zu erzeugen.

3.3.4. Beispiele für Listenbearbeitung

Das folgende Beispiel erzeugt die Liste der geraden und die Liste der ungeraden Elemente einer gegebenen Liste. Diese erste Lösung durchläuft die gegebene Liste zweimal und führt diese Tests zweimal aus:
```
sage: def fkt1(L):
....: return [filter(lambda n: n % 2 == 0, L), filter(lambda n: n % 2 == 1, L)]

sage: fkt1([1..10])
[[2, 4, 6, 8, 10], [1. 3, 5, 7, 9]]
```

Diese zweite Lösung durchläuft die Liste nur einmal und bildet die beiden Ergebnislisten nach und nach:

```
sage: def fkt2(L):
....:     res0 = []; res1 = []
....:     for k in L:
....:         if k%2 == 0: res0.append(k)  # oder res0[len(res0):] = [k]
....:         else: res1.append(k)         # oder res1[len(res1):] = [k]
....:     return [res0, res1]
```

Dieses Programm ersetzt die `for`-Schleife und die Hilfsvariablen durch einen rekursiven Aufruf und einen zusätzlichen Parameter:

```
sage: def fkt3a(L, res0, res1):
....:     if L == []: return [res0, res1]
....:     elif L[0]%2 == 0: return fkt3a(L[1:], res0+[L[0]], res1)
....:     else: return fkt3a(L[1:], res0, res1+[L[0]])

sage: def fkt3(L): return fkt3a(L, [], [])
```

Die Parameter `res0` und `res1` enthalten die ersten schon gefundenen Elemente, und die Parameterliste `L` verliert ein Element bei jedem rekursiven Aufruf.

Das zweite Beispiel hierunter extrahiert alle wachsenden Folgen einer Zahlenliste. Drei Variable werden verwendet. Die erste, `res`, speichert die bereits erhaltenen wachsenden Folgen, die Variable `anfang` bezeichnet die Position, wo die aktuelle wachsende Folge beginnt, und die Variable `k` ist der Schleifenindex:

```
sage: def teilFolgen(L):
....:     if L == []: return []
....:     res = []; anfang = 0; k = 1
....:     while k < len(L):    # 2 aufeinander folgende Elemente definiert
....:         if L[k-1] > L[k]:
....:             res.append(L[anfang:k]); anfang = k
....:         k = k+1
....:     res.append(L[anfang:k])
....:     return res

sage: teilFolgen([1, 4, 1, 5])
[[1, 4], [1, 5]]
sage: teilFolgen([4, 1, 5, 1])
[[4], [1,5], [1]]
```

Der Schleifenrumpf erlaubt den Übergang zum nächten Element der Liste. Ist der Test positiv, dann endet die aktuelle wachsende Teilfolge und es muss zu einer neuen Teilfolge übergegangen werden, sofern sie nicht durch das folgende Element verlängert wird.

Die Anweisung nach der Schleife fügt dem Endresultat die aktuelle wachsende Teilfolge hinzu, die mindestens ein Element besitzt.

3.3.5. Zeichenketten

Zeichenketten werden in einfache `'...'` oder doppelte `"..."` Anführungszeichen eingeschlossen. Zeichenketten in einfachen Anführungszeichen können doppelte Anführungszeichen enthalten und umgekehrt. Zeichenketten können auch von dreifachen Hochkommata `'''...'''` begrenzt werden, können dann auch mehrere Zeilen umfassen und einfache oder doppelte Anführungszeichen enthalten.

```
sage: S = 'Dies ist eine Zeichenkette.'
```

Das Zeichen \ ermöglicht einen Zeilenvorschub mit `\n`, Anführungszeichen mit `\"` oder mit `\'`, das TAB-Zeichen ist `\t` und das ESC-Zeichen ist `\\`. Die Zeichenketten können auch Buchstaben mit Akzenten enthalten und ganz generell beliebige Unicode-Zeichen.

```
sage: S = 'Ceci est une chaîne de charactères.'; S
'Ceci est une cha\xc3\xaene de charact\xc3\xa8res.'
```

```
sage: print S
Ceci est une chaîne de charactères.
```

Der Vergleich von Zeichenketten erfolgt zeichenweise anhand des Codes der Zeichen. Die Länge einer Kette erhält man mit der Funktion `len`, und die Verkettung wird durch die Operatoren der Addition „ + “ und der Multiplikation „ * “ bewirkt.

Der Zugriff auf eine Teilkette von `S` funktioniert wie bei Listen mit eckigen Klammern `S[n]`, `S[p:q]`, `S[p:]`, `S[:q]`, und das Ergebnis ist eine Zeichenkette. Die Sprache erlaubt aber keine Veränderung der ursprünglichen Zeichenkette durch eine Zuweisung dieser Form, aus diesem Grund wird der Typ der Zeichenketten als *immutabel* bezeichnet.

Die Funktion `str` wandelt ihr Argument in eine Zeichenkette um. Die Methode `split` zerlegt eine Zeichenkette an den Leerzeichen in eine Liste von Teilketten.

```
sage: S = 'eins zwei drei vier fünf sechs sieben'; L = S.split(); L
['eins', 'zwei', 'drei', 'vier', 'f\xc3\xbcnf', 'sechs', 'sieben']
```

Die recht vollständige Python-Bibliothek `re` unterstützt reguläre Ausdrückereguläre Ausdrücke und kann für die Suche nach Teilketten und für die Mustererkennung ebenfalls verwendet werden.

3.3.6. Teilen und Verdoppeln einer Struktur

Eine Liste in eckigen Klammer `[...]` kann durch Zuweisungen zu ihren Elemente, durch die Änderung der Anzahl ihrer Elemente oder mit Methoden wie `sort` oder `reverse` modifiziert werden.

Die Zuweisung einer vorhandenen Liste zu einer weiteren Variablen verändert die Struktur nicht, beide Listen haben Teil an denselben Daten. Im folgenden Beispiel bleiben die Listen `L1` und `L2` identisch; sie entsprechen zwei *Alias*-Namen desselben Objektes, und eine Veränderung der einen zeigt sich auch bei der anderen:

```
sage: L1 = [11, 22, 33]; L2 = L1
sage: L1[1] = 222; L2.sort(); L1, L2
[11, 33, 222], [11, 33, 222]
sage: L1[2:3] = []; L2[0:0] = [6, 7, 8]
sage: L1, L2
([6, 7, 8, 11, 33], [6, 7, 8, 11, 33])
```

Im Gegensatz dazu sind die Werte von `map`, mit `L[p:q]`, `filter` oder `.. for ... if ..` erzeugte Teillisten, Verkettungen mit + und * und Abflachungen mit `flatten` neue Strukturen, sodass von einer Verdoppelung der Daten gesprochen werden kann.

Im vorstehenden Beispiel verändert die Ersetzung von `L2 = L1` in der ersten Zeile durch einen der sechs Befehle hierunter die Ergebnisse komplett, denn die Modifikationen einer Liste wirken sich auf die andere nicht aus. Beide Strukturen bleiben unabhängig, beide Listen sind verschieden, selbst wenn ihre Werte gleich sind. So kopiert die Zuweisung `L2 = L1[:]` die Liste `L1` vom ersten bis zum letzten Eintrag und verdoppelt daher die Struktur von `L1`:

```
L1 = [11, 22, 33]  L2 = copy(L1)  L2 = L1[:]
L2 = [] + L1       L2 = L1 + []   L2 = 1*L1
```

Der Test auf geteilte Strukturen von Sage wird durch den binären Operator `is` bewirkt; ist das Ergebnis des Tests wahr, dann wirken alle Modifikationen auf beide Variablen auf einmal:

```
sage: L1 = [11, 22, 33]; L2 = L1; L3 = L1[:]
sage: L1 is L2, L2 is L1, L1 is L3, L1 == L3
[True, True, False, True]
```

Die Kopieroperationen arbeiten nur auf einer Listenebene. So pflanzt sich die Änderung im Innern einer Liste von Listen fort trotz der Kopie der ersten Ebene der Struktur:

```
sage: La = [1, 2, 3]; L1 = [1, La]; L2 = copy(L1)
sage: L1[1][0] = 5      # [1, [5, 2, 3]] für L1 und L2
sage: [L1 == L2, L1 is L2, L1[1] is L2[1]]
[True, False, True]
```

Die folgende Anweisung kopiert die Struktur mit beiden Ebenen:

```
sage: map(copy, L)
```

während die Funktion `copyRec` die Liste mit allen Ebenen rekursiv kopiert:

```
sage: def copyRec(L):
....:     if type(L) == list: return map(copyRec, L)
....:     else: return L
```

Die inverse lexikographische Ordnung ist als lexikographische Ordnung auf umgekehrt aufgezählten n-Tupeln dadurch definiert, dass die Reihenfolge für jedes Element umgekehrt wird:

$$\begin{aligned} & P = (p_0, p_1, \ldots, p_{n-1}) \prec_{\text{lexInv}} Q = (q_0, q_1, \ldots, q_{n-1}) \\ \Longleftrightarrow & \exists r \in \{0, \ldots, n-1\}, \quad (p_{r+1}, \ldots, p_{n-1}) = (q_{r+1}, \ldots, q_{n-1}) \text{ und } p_r > q_r. \end{aligned}$$

Die Programmierung der inversen lexikographischen Ordnung kann mittels der in Unterabschnitt 3.3.3 definierten Funktion `alpha` erfolgen, welche die lexikographische Ordnung implementiert. Das Kopieren der Listen P und Q ist erforderlich, um die Inversion durchzuführen, ohne die Daten zu verändern. Genauer gesagt kehrt die Funktion `lexInverse` die Reihenfolge der $n-Tupel$ mit `reverse` um und gibt die endgültige Reihenfolge mit dem Ergebnis $-(P_1 \prec_{\text{lex}} Q_1)$ zurück:

```
sage: def lexInverse(P, Q):
....:     P1 = copy(P); P1.reverse()
....:     Q1 = copy(Q); Q1.reverse()
....:     return -alpha(P1, Q1)
```

Die Modifikationen einer als Argument einer Funktion übergebenen Liste wirken sich global auf die Liste aus, denn die Funktionen kopieren nicht die Struktur der als Argument übergebenen Listen. So modifiziert eine Funktion, die einzig `P.reverse()` statt `P1 = copy(P)` und `P1.reverse()` ausführt, die Liste P; dieser Effekt, der *Nebenwirkung*[8] genannt wird, ist im allgemeinen unerwünscht.

Die Variable P ist eine lokale Variable der Funktion und unabhängig von anderen globalen Variablen gleichen Namens, doch das hat mit den Modifikationen nichts zu tun, die im Inneren einer als Argument übergebenen Liste vorgenommen werden.

[8] engl. side effect, wird ebenso häufig wie falsch mit Seiteneffekt übersetzt

Listen - ein Begriff, den Python und Sage verwenden - sind in diesen Systemen tatsächlich in der Form dynamischer Tabellen implementiert und haben eine andere Struktur als in Lisp oder in OCaml, wo sie durch ein Kopfelement t und eine Schwanzliste Q definiert sind. Der elementare Lispbefehl `cons(t,Q)` gibt eine Liste mit dem Term t als Kopf zurück, ohne die Liste Q zu verändern; in Python dagegen verändert das Hinzufügen eines Elementes e zu einer Tabelle T durch `T.append(e)` die Tabelle T. Beide Repräsentationen der Daten haben ihre Vorteile, und der Übergang von einer Darstellung zur anderen ist möglich, doch ist die Effizienz der Algorithmen in den beiden Fällen nicht dieselbe.

3.3.7. Veränderbare und nicht veränderbare Daten

Listen erlauben Daten zu strukturieren und zu manipulieren, weil sie veränderbarbar sind. Aus diesem Grund werden diese Strukturen als modifizierbar oder mutabel eingestuft.

Python ermöglicht auch die Definition von festen oder immutablen Daten. Die immutable Struktur, die Listen entspricht, sind Tupel, die in runden Klammern `(...)` geschrieben werden anstatt in eckigen `[...]`. Ein Tupel mir einem einzigen Element wird mit einem Komma hinter diesem Eintrag geschrieben.

```
sage: S0 = (); S1 = (1, ); S2 = (1, 2)
sage: [1 in S1, 1 == (1)]
[True, True]
```

Die Operationen des Zugriffs auf Tupel sind die gleichen wie die für Listen, beispielsweise die Erzegung des Abbildes eines Tupels durch `map` oder eines Teiltupels durch `filter`. In allen diesen Fällen ist das Ergebnis eine Liste, und mit `for` transformieren wir ein Tupel in eine Liste.

```
sage: S1 = (1, 4, 9, 16, 25); [k for k in S1]
[1, 4, 9, 16, 25]
```

Der Befehl `zip` gruppiert mehrere Listen oder Tupel um, genau wie der Befehl `map`:

```
sage: L1 = [0..4]; L2 = [5..9]
sage: zip(L1, L2)
[(0, 5), (1, 6), (2, 7), (3, 8), (4, 9)]
sage: map(lambda x, y: (x, y), L1, L2)
[(0, 5), (1, 6), (2, 7), (3, 8), (4, 9)]
```

3.3.8. Endliche Mengen

Anders als bei Listen kommt es bei Mengen nur darauf an, ob ein Element vorhanden ist oder nicht. Weder ist seine Position definiert noch wie oft es vorkommt. Sage kann endliche Mengen mittels der Funktion `Set` aus Listen erzeugen. Das Ergebnis wird in geschweifte Klammern eingeschlossen:

```
sage: E = Set([1, 2, 4, 8, 2, 2, 2]); F = Set([7, 5, 3, 1]); E, F
({8, 1, 2, 4}, {1, 3, 5, 7})
```

Der Operator des Enthaltensein `in` testet, ob ein Objekt Element der Menge ist, und Sage erlaubt die Vereinigung zweier Mengen mit + oder |, ermittelt ihre Schnittmenge mit &, ihre Differenz mit − und ihre symmetrische Differenz mit `^^`:

```
sage: 5 in E, 5 in F, E + F == E  F
(False, True, True)
sage: E & F, E - F, E ^^ F
({1}, {8, 2, 4}, {2, 3, 4, 5, 7, 8})
```

Die Funktion `len(E)` gibt die Kardinalität einer endlichen Menge zurück. Die Operationen `map`, `filter` und `for .. if ..` finden auf Mengen genauso Anwendung wie für Tupel. Die Ergebnisse sind Listen. Der Zugriff auf ein Element erfolgt mit `E[k]`. Die beiden Befehle hierunter bilden jeweils die gleiche Liste von Elementen einer Menge:

```
sage: [E[k] for k in [0..len(E)-1]], [t for t in E]
[8, 1, 2, 4], [8, 1, 2, 4]
```

Die folgende Funktion testet die Inklusion, das Enthaltensein einer Menge E in einer anderen Menge F mittels der Vereinigung:

```
sage: def inklus(E, F): return E+F == F
```

Anders als Listen sind Mengen von immutablem Typ und somit nicht modifizierbar; auch ihre Elemente müssen immutabel sein. Es sind also Mengen von Tupeln und Mengen von Mengen möglich, wir können aber keine Mengen von Listen bilden:

```
sage: Set([Set([]), Set([1]), Set([2]), Set([1, 2])])
{{1, 2}, {}, {2}, {1}}
sage: Set([(), (1,), (2,), (1, 2)])
{(1, 2), (2,), (), (1,)}
```

Diese Funktion zählt mit einem rekursiven Verfahren alle Teilmengen einer Menge auf:

```
sage: def TeilM(EE):
....:     if EE == Set([]): return Set([EE])
....:     else: return mitOderOhneElt(EE[0], TeilM(Set(EE[1:])))

sage: def mitOderOhneElt(a, E):
....:     return Set(map(lambda F: Set([a])+F, E)) + E

sage: TeilM(Set(1, 2, 3))
{{3}, {1, 2}, {}, {2, 3}, {1}, {1, 3}, {1, 2, 3}, {2}}
```

Die Funktion `mitOderOhneElt(a, E)` nimmt eine Menge E von Teilmengen und bildet eine Menge, die zweimal mehr Elemente hat, die ihrerseits Teilmengen sind und denen das Element a hinzugefügt ist. Die rekursive Erzeugung beginnt mit der einelementigen Menge E `= {0}`.

3.3.9. Diktionäre

Schließlich integriert Python, und damit Sage, den Begriff des Diktionärs (Dictionary). Wie ein Telefonbuch ordnet ein Diktionär jedem Eintrag einen Wert zu.

Die Einträge eines Diktionärs sind Unveränderliche beliebigen Typs, Zahlen, Zeichenketten, Folgen usw. Die Syntax ist der von Listen vergleichbar, außer dass das leere Diktionär `dict()` als `{}` abgekürzt werden kann.

```
sage: D = {}; D['eins'] = 1, D['zwei'] = 2, D['drei'] = 3; D['zehn'] = 10
sage: D['zwei'] + D['drei']
5
```

Das vorstehende Beispiel sagt also, wie ein neuer Eintrag in ein Diktionär eingefügt wird, und wie auf ein Feld mit `D[...]` zugegriffen wird.

Der Operator `in` prüft, ob ein Eintrag in einem Diktionär enthalten ist und die Befehle `del D[x]` oder `D.pop(x)` liefern den Eintrag x dieses Diktionärs.

Das folgende Beispiel zeigt, wie ein Diktionär eine Anwendung auf einer endlichen Menge darstellen kann:

$$E = \{a_0, a_1, a_2, a_3, a_4, a_5\}, \quad \begin{array}{lll} f(a_0) = b_0, & f(a_1) = b_1, & f(a_2) = b_2 \\ f(a_3) = b_0, & f(a_4) = b_3, & f(a_5) = b_3. \end{array}$$

Die Methoden der Diktionäre sind mit denen der anderen aufzählenden Strukturen vergleichbar. Der untenstehende Code implementiert die vorstehende Funktion und erzeugt die Definitionsmenge E und die Bildmenge $\mathrm{Im}\, f = f(E)$ mittels der Methoden `keys()` und `values()`:

```
sage: D = {'a0':'b0', 'a1':'b1', 'a2':'b2', 'a3':'b0', 'a4':'b3', 'a5':'b3'}
sage: E = Set(D.keys()); Imf = Set(D.values())
sage: Imf == Set(map(lambda t:D[t], E))    # ist äquivalent True
```

Der letzte Befehl überträgt die mathematische Definition $\mathrm{Im}\, f = \{f(x) | x \in E\}$. Diktionäre können auch aus Listen oder Paaren $[Schluessel, Wert]$ durch den folgenden Befehl erzeugt werden:

```
dict(['a0','b0'], ['a1','b1'], ...).
```

Die beiden folgenden Befehle, die auf die Einträge des Diktionärs oder auf das Diktionär selbst angewendet werden, sind mit der Methode `D.values()` äquivalent:

```
map(lambda t:D[t], D)    map(lambda t:D[t], D.keys())
```

Der folgende Test auf die Anzahl der Elemente bestimmt mit `len(D)` die Anzahl der Einträge im Diktionär, falls die durch `D` dargestellte Funktion injektiv ist,

```
sage: def injektiv(D):
....:     return len(D) == len(Set(D.values()))
```

Die ersten beiden Befehle hierunter erzeugen das Bild $f(F)$ und das Urbild $f^{-1}(G)$ der Teilmengen F und G einer durch das Diktionär D definierten Funktion; die letztere definiert

ein Diktionär `DR`, das der Urbildfunktion f^{-1} einer als bijektiv vorausgesetzten Funktion f entspricht:

```
sage: Set([D[t] for t in F])
sage: Set([t for t in D if D[t] in G])
sage: DR = dict((D[t], t) for t in D)
```

4. Graphiken

Die Veranschaulichung von Funktionen zweier oder dreier Veränderlicher oder einer Datenreihe erleichtert das Verständnis mathematischer oder physikalischer Phänomene und ermöglicht, Resultate effektiv zu mutmaßen. In diesem Kapitel illustrieren wir anhand von Beispielen die graphischen Fähigkeiten von Sage.

4.1. Kurven in 2D

Eine ebene Kurve kann auf mehrere Arten definiert werden: als Graph einer Funktion einer Veränderlichen, durch ein System parametrisierter Gleichungen, durch eine Gleichung in Polarkoordinaten oder durch eine implizite Gleichung. Diese vier Fälle stellen wir vor, danach geben wir einige Beispiele für die Veranschaulichung von Daten.

4.1.1. Graphische Darstellung von Funktionen

Um den Graphen einer symbolischen Funktion oder einer Python-Funktion auf einem Intervall $[a, b]$ zu zeichnen, verfügen wir über zwei Möglichkeiten: `plot(f(x),a,b)` oder `plot(f(x),x,a,b)`.

```
sage: plot(x*sin(1/x), x, -2, 2, plot_points=500)
```

Von den zahlreichen Optionen der Funktion `plot` nennen wir

- `plot_points` (voreingestellt 200): Mindestzahl der berechneten Punkte;
- `min` und `max`: Grenzen des Intervalls, auf dem die Funktion gezeichnet wird;
- `color`: Farbe der Zeichnung, ein RGB-Tripel, eine Zeichenkette, z.B. `'blue'` oder eine HTML-Farbe (z.B. `'#aaff0b'`;
- `detect_poles` (voreingestellt `False`): erlaubt oder verbietet das Einzeichnen einer vertikalen Asymptote an den Polstellen einer Funktion;
- `alpha`: Transparenz des Striches;
- `thickness`: Strichstärke;
- `linestyle`: Linienart, punktiert: `(':')`, strichpunktiert: `('-.')` oder durchgezogen `('-')`, der voreingestellte Wert.

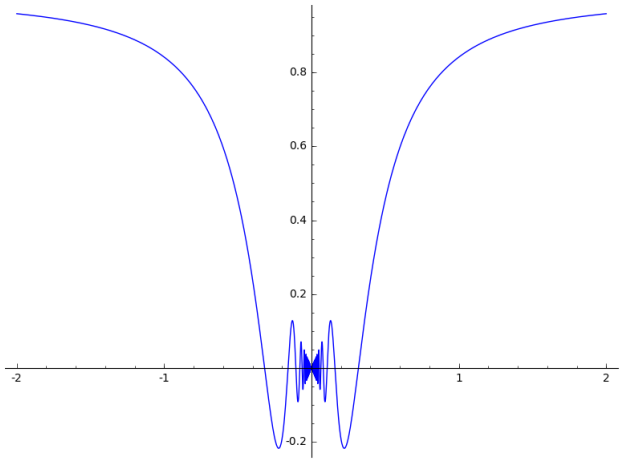


Abb. 4.1 - Graph von $x \mapsto x \sin \frac{1}{x}$

Um die Zeichnung sichtbar zu machen, können wir das graphische Objekt einer Variablen zuweisen, sagen wir `g`, und dann die Methode `show` benutzen, der beispielsweise die minimalen und maximalen Werte der Ordinate übergeben werden können `(g.show(ymin=-1,ymax=3))` oder auch eine bestimmte Streckung wie `(g.show(aspect_ratio=1))` für ein orthonormiertes Koordinatensystem.

Die fertige Zeichnung kann mit dem Befehl `save` in unterschiedlichen Formaten exportiert werden, die durch die Erweiterungen `.pdf`, `.png`, `.ps`, `.eps`, `.svg` und `.sobj` gekennzeichnet sind:

```
g.save(name.png, aspect_ratio=1, xmin=-1,ymin=3, ymin=-1, ymax=3)
```

Um eine Graphik mit dem Befehl `\includegraphics` in ein LaTeX-Dokument einzufügen, wird die Erweiterung `.eps` gewählt (encapsulated PostScript), wenn das Dokument mit `latex` kompiliert wird, oder die Erweiterung `.pdf` (die wegen der besseren Auflösung der Erweiterung `.png` vorzuziehen ist), wenn das Dokument mit `pdflatex` kompiliert wird.

Zeichnen wir nun in dieselbe Graphik die Sinusfunktion und die ersten Polynome der Taylorentwicklung um den Ursprung:

```
sage: def p(x, n):
....:     return (taylor(sin(x), x, 0, n))
sage: xmax=15; n=15
sage: g=plot(sin(x), x, -xmax, xmax)
sage: for d in range(n):
....:     g += plot(p(x, 2*d + 1), x, -xmax, xmax, \
....:       color=(1.7*d/(2*n), 1.5*d/(2*n), 1.3*d/(2*n)))
sage: g.show(ymin=-2, ymax=2)
```

Man hätte auch eine Animation erzeugen können, um zu sehen, wie das Taylor-Polynom sich mit der Erhöhung des Grades der Sinuskurve mehr und mehr annähert. Möchte man eine Animation sehen, genügt dafür die Speicherung als `gif`-Datei.

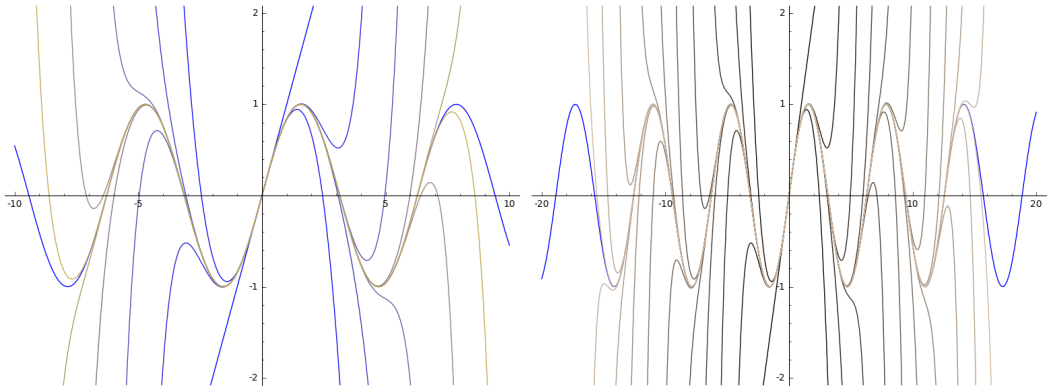


Abb. 4.2 - Einige Taylor-Polynome der Sinusfunktion um den Ursprung

```
sage: a = animate([sin(x), taylor(sin(x), x, 0, 2*k+1)]\
....:     for k in range(0, 14)], xmin=-14, xmax=14),\
| ....:      ymin=-3, ymax=3, figsize=[8, 4])
sage: a.show(); a.save('~/bin/animation.gif')
```

Kehren wir zur Funktion `plot` zurück, um beispielsweise das Phänomen von Gibbs zu beobachten. Wir zeichnen die Partialsumme der Ordnung 20 der Fourierreihe für die Rechteckkurve.

```
sage: f2(x) = 1;  f1(x) = -1
sage: f = piecewise([[(-pi,0),f1], [(0,pi),f2]])
sage: S = f.fourier_series_partial_sum(20,pi)
sage: g = plot(S, x, -8, 8, color='blue')
sage: scie(x) = x - 2*pi*floor((x + pi)/(2*pi))
sage: g += plot(scie(x)/abs(scie(x)), x, -8, 8, color='red')
sage: g
```

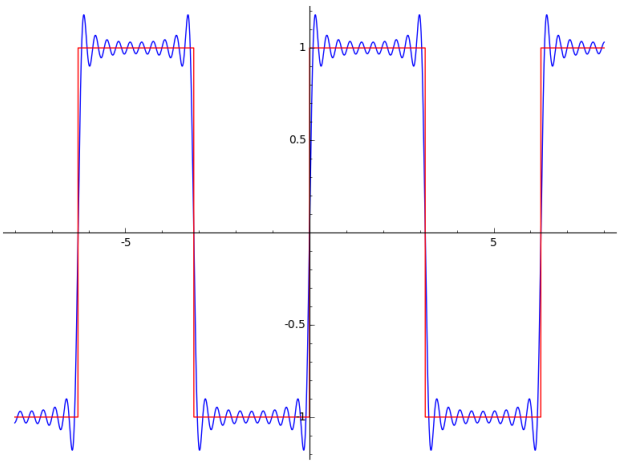


Abb. 4.3 - Zerlegung einer Rechteckfunktion in eine Fourierreihe

Mit obigem Code ist `f` eine mit Hilfe des Befehls `piecewise` stückweise definierte Funktion auf $[-\pi;\pi]$. Um die Verlängerung von `f` auf die mit 2π periodische Funktion zu bewirken, ist es am

einfachsten, dafür einen Ausdruck zu suchen, der für alle reellen Zahlen (im Definitionsbereich von `scie/abs(scie)`) definiert ist. Die ersten 20 Terme der Fourierreihe sind

$$S = \frac{1}{\pi}\left(\sin(x) + \frac{\sin(3x)}{3} + \frac{\sin(5x)}{5} + \ldots + \frac{\sin(19x)}{9}\right).$$

4.1.2. Kurven in Parameterdarstellung

Kurven in Parameterdarstellung ($x = f(t)$, $y = g(t)$) werden mit dem Befehl `parametric_plot((f(t), g(t)), (t, a, b))` gezeichnet, wobei $[a, b]$ das Intervall ist, das der Parameter durchläuft. Stellen wir beispielsweise diese Kurve dar:

$$\begin{cases} x(t) &= \cos(t) + \frac{1}{2}\cos(7t) + \frac{1}{3}\sin(17t) \\ y(t) &= \sin(t) + \frac{1}{2}\sin(7t) + \frac{1}{3}\cos(17t) \end{cases}$$

```
sage: t = var('t')
sage: x = cos(t) + cos(7*t)/2 + sin(17*t)/3
sage: y = sin(t) + sin(7*t)/2 + cos(17*t)/3
sage: g = parametric_plot((x, y), (t, 0, 2*pi))
sage: g.show(aspect_ratio=1)
```

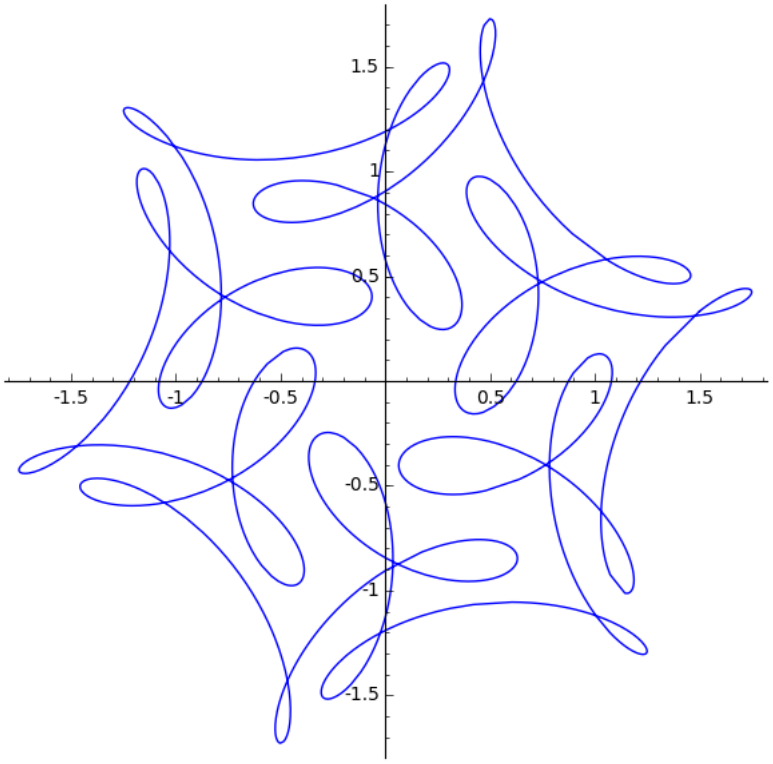


Abb. 4.4 - Kurve in Parameterdarstellung für $x = \cos(t) + \frac{1}{2}\cos(7t) + \frac{1}{3}\sin(17t)$ und $y = \sin(t) + \frac{1}{2}\sin(7t) + \frac{1}{3}\cos(17t)$

4.1.3. Kurven in Polarkoordinaten

Kurven in Polarkoordinaten $\rho(\theta)$ werden mit dem Befehl `polar_plot(rho(theta), (theta, a, b))` gezeichnet, wobei $[a, b]$ das Intervall ist, das vom Parameter durchlaufen wird.

Stellen wir beispielsweise zwei Rosetten mit der Polargleichung $\rho(\theta) = 1 + e\cos n\theta$ dar mit $n = 20/19$ und $e \in \{2, 1/3\}$.

```
sage: t = var('t'); n = 20/19
sage: g1 = polar_plot(1+2*cos(n*t),(t,0,n*36*pi),plotpoints=5000)
sage: g2 = polar_plot(1+1/3*cos(n*t),(t,0,n*36*pi),plot_points=5000)
sage: g1.show(aspect_ratio=1); g2.show(aspect_ratio=1)
```

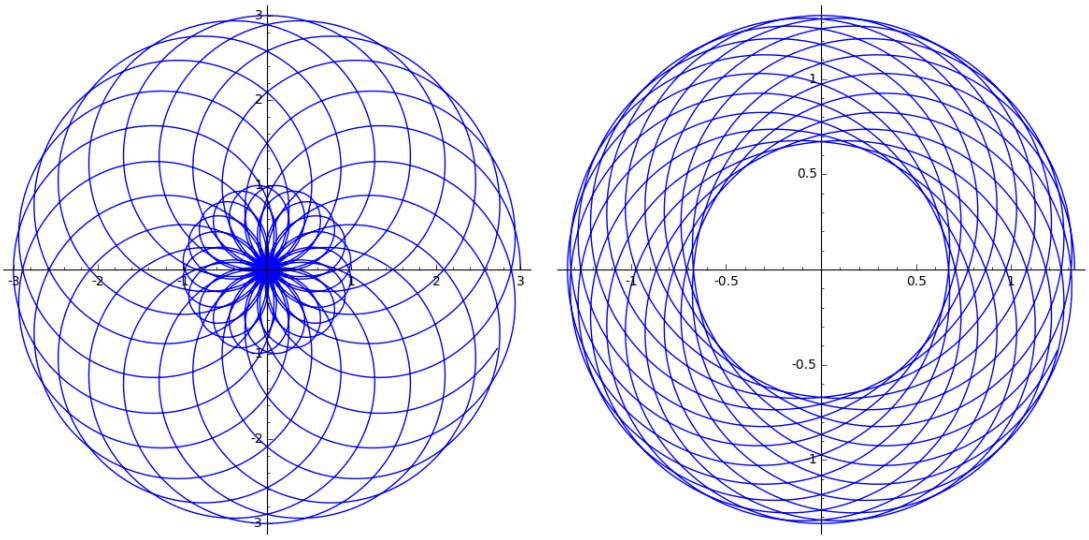


Abb. 4.5 - Rosetten der Gleichung $\rho(\theta) = 1 + e\cos n\theta$

Übung 12. Darzustellen ist eine Familie von Pascalschen Schnecken mit der Polargleichung $\rho(\theta) = a + \cos\theta$, wobei der Parameter a zwischen 0 und 2 mit einer Schrittweite von 0.1 zu variieren ist.

4.1.4. Durch implizite Gleichungen definierte Kurven

Zur Darstellung einer Kurve, die durch eine implizite Gleichung gegeben ist, verwenden wir die Funktion `implicit_plot(f(x, y), (x, a, b), (y, c, d))`; allerdings können wir auch den Befehl `complex_plot` verwenden, der die Veranschaulichung einer Funktion von zwei Veränderlichen mit farbigen Niveaulinien ermöglicht. Wir wollen die durch die implizite Gleichung $\mathcal{C} = \{z \in \mathbb{C} : |cos(z^4)| = 1\}$ gegebene Funktion darstellen.

```
sage: z = var('z')
sage: g1 = complex_plot(abs(cos(z^4))-1,
....:         (-3,3), (-3,3), plot_points=400)
sage: f = lambda x, y: (abs((cos(x +I*y)^4)) - 1)
sage: g2 = implicit_plot(f, (-3,3), (-3,3), plot_points=400)
sage: g1.show(aspect_ratio=1); g2.show(aspect_ratio=1)
```

4.1.5. Die Darstellung von Daten

Zum Zeichnen eines Säulendiagramms verfügen wir über zwei ganz unterschiedliche Funktionen. Zuerst der Befehl `bar_chart`, der als Argument eine Liste ganzer Zahlen aufnimmt und einfach senkrechte Säulen zeichnet, deren Höhe durch die Elemente der Liste gegeben ist (in derselben Reihenfolge wie in der Liste). Halten wir fest, dass die Option `width` gestattet, die Breite der Säulen festzulegen.

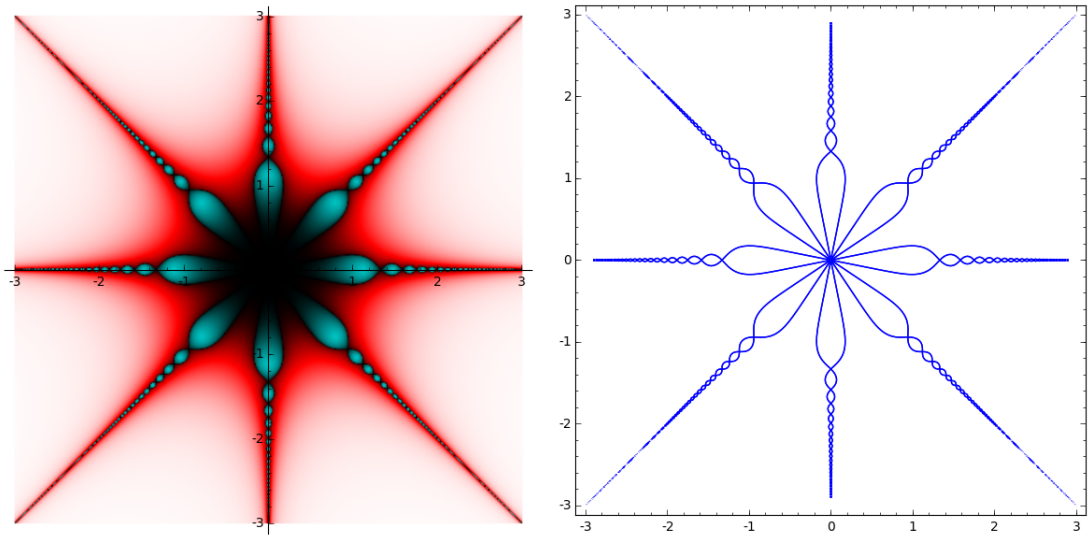


Abb. 4.6 - Durch die Gleichung $|cos(z^4)| = 1$ definierte Funktion

```
sage: bar_chart([randrange(15) for i in range(20)])
sage: bar:chart([x^2 for x in range(1,20)], width=0.2)
```

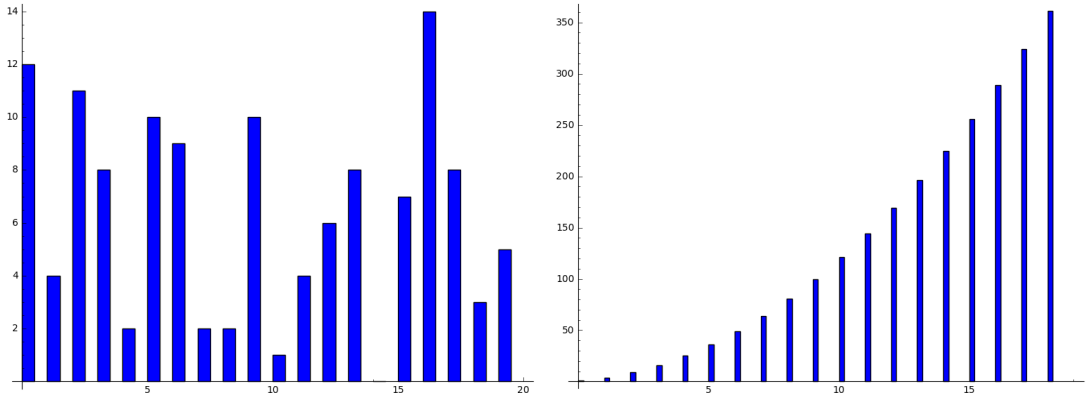
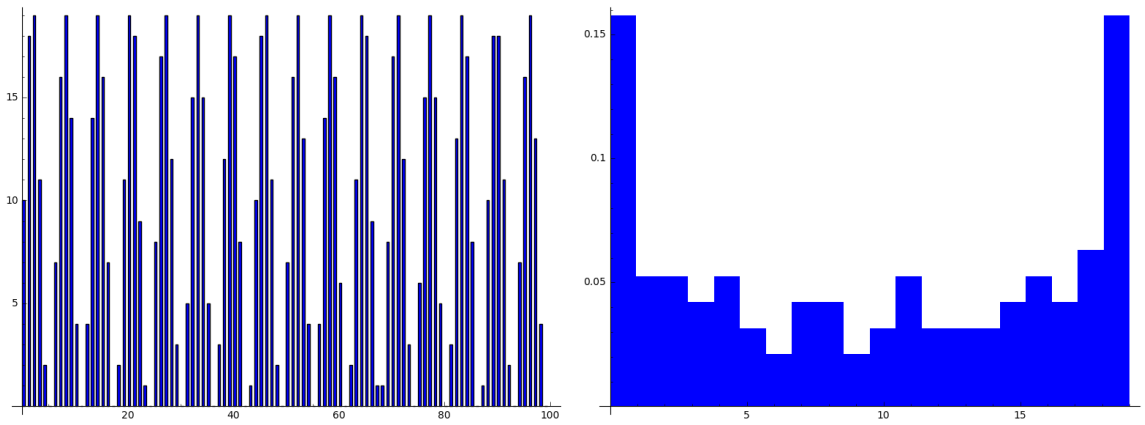


Abb. 4.7 - Säulendiagramme

Wenn hingegen für eine Liste von Fließkommazahlen das Histogram einer Häufigkeitsverteilung zu zeichnen ist, verwenden wir die Funktion `plot_histogram`: die Werte der Liste werden sortiert und in Intervalle eingeteilt (die Anzahl der Intervalle wird durch die Variable `bins` festgelegt, deren Größe mit 50 voreingestellt ist). Die Höhe der Säule jedes Intervalls gleicht der entsprechenden Häufigkeit.

```
sage: liste = [10 + floor(10*sin(i)) for i in range(100)]
sage: bar_chart(liste)
sage: finance.TimeSeries(liste).plot_histogram(bins=20)
```



(a) gezeichnet mit `bar_chart`

(b) gezeichnet mit `plot_histogram`

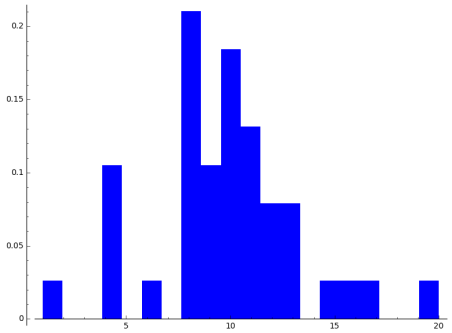
Oft sind die zu untersuchenden statistischen Daten in einem Arbeitsblatt einer Tabellenkalkulation gespeichert. Das Python-Modul `csv` ermöglicht dann den Import der Daten aus einer csv-Datei. Nehmen wir beispielsweise an, wir wollen ein Histogramm für die Zensuren von 40 Schülern berechnen. Die Punkte (französisches Benotungsystem) verteilen sich folgendermaßen (obere Zeile: erreichte Punkte, untere Zeile: Anzahl der Schüler mit dieser Punktzahl):

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	1	0	0	4	0	1	0	8	4	7	5	3	3	0	1	1	1	0	0	1

Die Punkte jedes Schülers werden in die dritte Spalte der Tabellenkalkulation eingetragen. Das Arbeitsblatt wird in der Datei `ds01.csv` gespeichert, wobei darauf zu achten ist, dass beim Speichern als csv-Dokument als Trennelemente Kommas ausgewählt werden.

Um die Punkte in dieser Spalte auszulesen, arbeiten wir mit folgenden Anweisungen (die oberste Zeile enthält normalerweise Text, sodass eventuelle Fehler bei der Umwandlung in Zahlen mit der Anweisung `try` aufgefangen werden müssen):

```
sage: import csv
sage: reader = csv.reader(open("ds01.csv"))
sage: noten = []; liste = []
sage: for zeile in reader:
.....:     noten.append(zeile[2]) # die dritte Spalte hat den Index 2
....: for i in noten:
....:     try:
....:         f = float(i)
....:     except ValueError:
....:         pass
....:     else:
....:         liste.append(f)
sage: finance.TimeSeries(liste).plot_histogram(bins=20)
```



Um eine Liste verbundener Punkte (bzw. eine Punktwolke) zu zeichnen, verwenden wir den Befehl `line(p)` (bzw. `point(p)`), wobei `p` eine Liste von zweielementigen Listen ist, die Abszisse und Ordinate der zu zeichnenden Punkte enthalten.

Beispiel *(Zufallsbewegung).* Ausgehend von einem Startpunkt O verschiebt sich ein Teilchen um den Betrag l in gleichbleibenden Zeitspannen t jedes Mal in eine beliebige Richtung, die von den vorigen Richtungen unabhängig ist. Wir wollen die Spur eines solchen Teilchens beispielhaft aufzeichnen. Die rote Gerade verbindet Start- und Endpunkt.

```
sage: from random import *
sage: n, l, x, y = 1000, 1, 0, 0; p = [[0,0]]
sage: for k in range(n):
....:     theta = (2*pi*random()).n(digits=5)
....:     x, y = x + l*cos(theta), y + l*sin(theta)
....:     p.append([x,y])
sage: g1 = line([p[n], [0,0]], color='red', thickness=2)
sage: g1 += line(p, thickness=.4); g1.show(aspect_ratio=1)
```

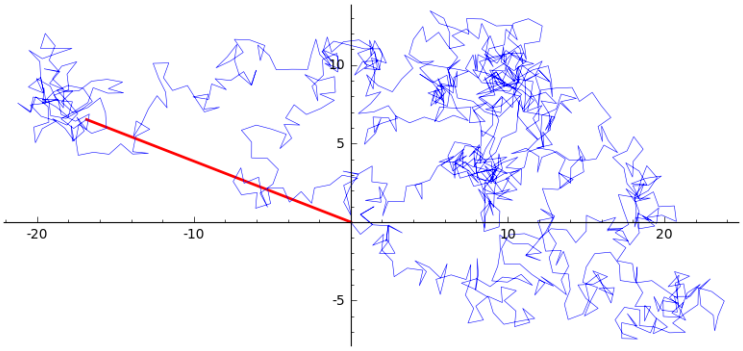


Abb. 4.8 - Zufallsweg

Beispiel *(Gleichverteilte Folgen).* Für eine gegebene reelle Folge $(u_n)_{n\in\mathbb{N}^*}$ konstruieren wir den Polygonzug, dessen aufeinander folgende Ecken die Punkte des Ausdrucks

$$z_N = \sum_{n\leq N} e^{2i\pi u_n}.$$

sind. Ist die Folge gleichverteilt modulo 1, entfernt sich die gebrochene Linie nicht zu schnell vom Startpunkt. Wir können dann aus der Spur der gebrochenen Linie die Regelmäßigkeit der Verteilung der Folge vermuten. Wir wollen den Polygonzug für folgende Fälle zeichnen:

- $u_n = n\sqrt{2}$ und $N = 200$,
- $u_n = n\ln(n)$ und $N = 10000$,
- $u_n = E(n\ln(n))\sqrt{2}$ und $N = 10000$ (wobei E den ganzzahligen Anzeil bezeichnet),
- $u_n = p_n\sqrt{2}$ und $N = 10000$ (hier ist p_n die n-te Primzahl).

Die Abb. 4.9 erhält man auf folgende Weise (hier für $u_n = n\sqrt{2}$):

```
sage: length = 200; n = var('n')
sage: u = lambda n: n*sqrt(2)
sage: z = lambda n: exp(2*I*pi*u(n)).n()
sage: vertices = [CC(0, 0)]
sage: for n in range(1, length):
....:     vertices.append(vertices[n-1] + CC(z(n)))
sage: line(vertices).show(aspect_ratio=1)
```

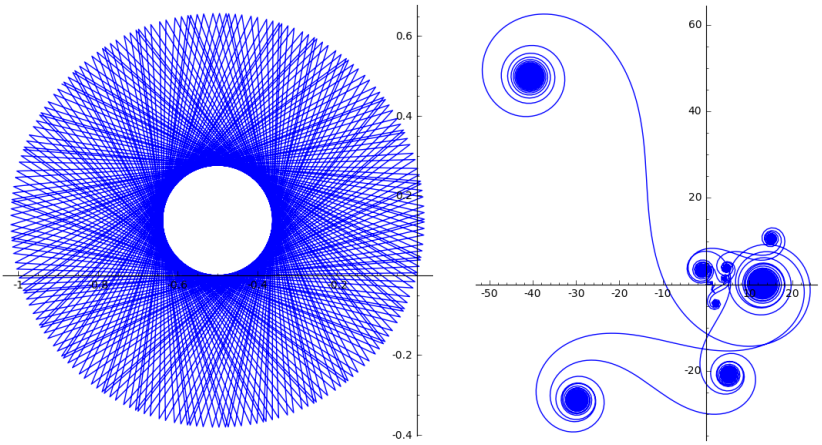
Man bememerkt, dass die Zeichnung Abb. 4.9 (a) besonders regelmäßig ist. Das ermöglicht uns voherzusehen, dass die Gleichverteilung von $n\sqrt{2}$ von deterministischer Natur ist. Bei der Folge $u_n = n\ln(n)\sqrt{2}$ erzeugt die Prüfung der eingegebenen Werte den Eindruck einer wohl eher zufälligen Bildung. Indessen ist die Zeichnung in Abb. 4.9 (b) bemerkenswert gut strukturiert. Die Zeichnung Abb. 4.9 (c) zeigt die gleiche Struktur wie die zweite. Und endlich lässt die Zeichnung in Abb. 4.9 (d) eine vollkommen andere Struktur der Verteilung modulo $1/\sqrt{2}$ der Folge der Primzahlen entstehen: die Spiralen sind verschwunden und das Verhalten erinnert an die Zeichnung in Abb. 4.8, die wir im Falle einer Folge von Zufallszahlen u_n bekommen. Es scheint daher, dass „die Primzahlen den gesamten Zufall in Beschlag nehmen, mit dem sie ausgestattet sind...“

Wegen einer eingehenderen Interpretation der erhaltenen Zeichnungen sei auf das Buch „Les Nombres Premiers“ (Que Sais-je) von Gérard Tenenbaum und Michel Mendès-France verwiesen [TMF00].

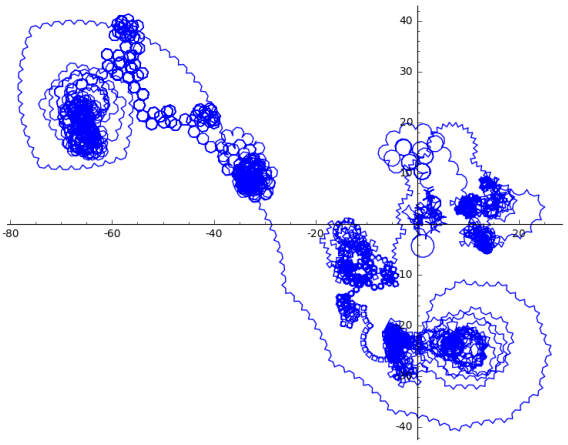
Übung 13 *(Zeichnung von Gliedern einer rekursiv definierten Folge).* Man betrachte die Folge $(u_n)_{n\in\mathbb{N}}$ mit

$$\begin{cases} u_0 = a \\ \forall n \in \mathbb{N}, u_{n+1} = |u_n^2 - \frac{1}{4}| \end{cases}.$$

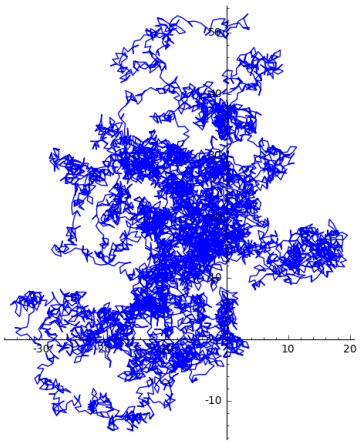
Stellen Sie das Verhalten der Folge graphisch dar, indem Sie eine Liste mit den Punkten $[[u_0, 0], [u_0, u_1], [u_1, u_1], [u_1, u_2], [u_2, u_2], \ldots]$ und $a \in \{-0.4, 1.1, 1.3\}$ aufstellen.



(a) Der Fall $u_n = n\sqrt{2}$.



(b) Der Fall $n\ln(n)\sqrt{2}$



(c) Der Fall $E(n\ln(n))\sqrt{2}$

(d) Der Fall $u_n = p_n\sqrt{2}$

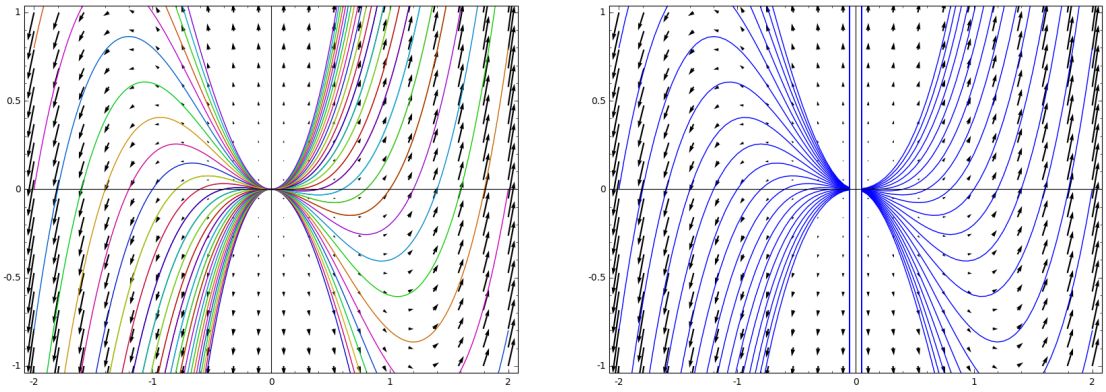
Abb. 4.9 - Gleichverteilte Folgen

4.1.6. Zeichnen der Lösung einer Differentialgleichung

Wir können die vorstehenden Befehle zur Darstellung der Lösung einer Differentialgleichung oder auch eines System von Differentialgleichungen kombinieren. Um eine gewöhnliche Differentialgleichung symbolisch zu lösen, verwenden wir die Funktion `desolve`, deren Studium Gegenstand von Kapitel 10 ist. Um eine Differentialgleichung symbolisch zu lösen, bietet uns Sage mehrere Werkzeuge an: `desolve_rk4` (mit einer Syntax ähnlich wie die Funktion `desolve`), `odeint`, (das `SciPy` verwendet), und schließlich `ode_solver`, (das die Bibliothek GSL aufruft, deren Verwendung in Abschnitt 14.2 behandelt wird). Die Funktionen `desolve_rk4` und `odeint` geben eine Liste von Punkten zurück, die mit dem Befehl `line` unschwer gezeichnet werden können. Auch wir werden mit ihnen in diesem Abschnitt numerische Lösungen zeichnen.

Beispiel *(Ungelöste lineare Differentialgleichung erster Ordnung).* Wir möchten die Integralkurven der Differentialgleichung $xy' - 2y = x^3$ zeichnen.

```
sage: x = var('x'); y = function('y')
sage: DE = x*diff(y(x), x) == 2*y(x) + x^3
sage: desolve(DE, [y(x),x])
(_C + x)*x^2
sage: sol = []
sage: for i in srange(-2, 2, 0.2)
....:     sol.append(desolve(DE, [y, x], ics=[1, i]))
....:     sol.append(desolve(DE, [y, x], ics=[-1, i]))
sage: g = plot(sol, x, -2, 2)
sage: y = var('y')
sage: g += plot_vector_field((x, 2*y+x^3), (x,-2,2), (y,-1,1))
sage: g.show(ymin=-1, ymax=1)
```



(a) Symbolische Lösung (b) Numerische Lösung

Abb. 4.10 - Zeichnung der Integralkurven von $xy' - 2y = x^3$.

Zur Verringerung der Rechenzeit ist hier zu empfehlen, die allgemeine Lösung der Gleichung „von Hand“ zu beschaffen und eine Liste partikulärer Lösungen zu erzeugen (wie das in der Auflösung von Übung 14 gemacht wird) und nicht mehrmals hintereinander die Differentialgleichung mit verschiedenen Anfangswerten zu lösen. Ebenso hätten wir eine numerische Lösung dieser Gleichung berechnen lassen können (mittels der Funktion `desolve_rk4`), um daraus die Integralkurven zu zeichnen:

```
sage: x, y = var('x y'); y = function('y')
sage: DE = x*diff(y(x), x) == 2*y(x) + x^3
sage: g = Graphics()   # Erzeugung einer leeren Graphik
sage: for i in srange(-2, 2, 0.2):
....:     g += line(desolve_rk4(2*y/x + x^2, dvar=y, ivar=x, ics=[1, i],\
                    step=0.05, end_points=[0,2]))
....:     g += line(desolve_rk4(2*y/x + x^2, dvar=y, ivar=x, ics=[-1,i],\
                    step=0.05, end_points=[-2,0]))
sage: y = var('y')
sage: g += plot_vector_field((x, 2*y+x^3), (x,-2,2), (y,-1,1))
sage: g.show(ymin=-1, ymax=1)
```

Wie man an vorstehendem Beispiel sieht, übernimmt die Funktion `desolve_rk4` als Argument eine Differentialgleichung (oder die rechte Seite der Gleichung in der Form $y' = f(x, y)$), den Namen der unbekannten Funktion, die abhängige Variable, die Anfangsbedingung, die Schrittweite und das Intervall der Lösung. Das optionale Argument `output` ermöglicht die Präzisierung der Ausgabe: der voreingestellte Wert `'list'` gibt eine Liste zurück (was zweckmäßig ist, wenn man die Graphiken wie hier überlagern will), `'plot'` zeichnet das Bild der Lösung und `'slope_field'` fügt die Steigungen der Integralkurve hinzu.

Übung 14. Zeichnen Sie die Integralkurven der Gleichung $x^2y' - y = 0$ für $-3 \leq x \leq 3$ und $-5 \leq y \leq 5$.

Wir geben jetzt ein Beispiel für die Verwendung der Funktion `odeint` aus dem Modul SciPy.

Beispiel (*Nichtlineare Differentialgleichung erster Ordnung.*) Zeichnen Sie die Integralkurven der Gleichung $y'(t) + cos(y(t) \cdot t) = 0$.

```
sage: import scipy; from scipy import integrate
sage: f = lambda y, t: -cos(y*t)
```

```
sage: t = srange(0, 5, 0.1); p = Graphics()
sage: for k in srange(0, 10,0.15):
....:     y = integrate.odeint(f, k, t)
....:     p += line(zip(t, flatten(y)))
sage: t = srange(0, -5, -0.1); q = Graphics()
sage: for k in srange(0, 10, 0.15):
....:     y = integrate.odeint(f, k, t)
....:     q += line(zip(t, flatten(y)))
sage: y = var('y')
sage: v = plot_vector_field((1, -cos(x*y)), (x,-5,5), (y,-2,11))
sage: g = p + q + v; g.show()
```

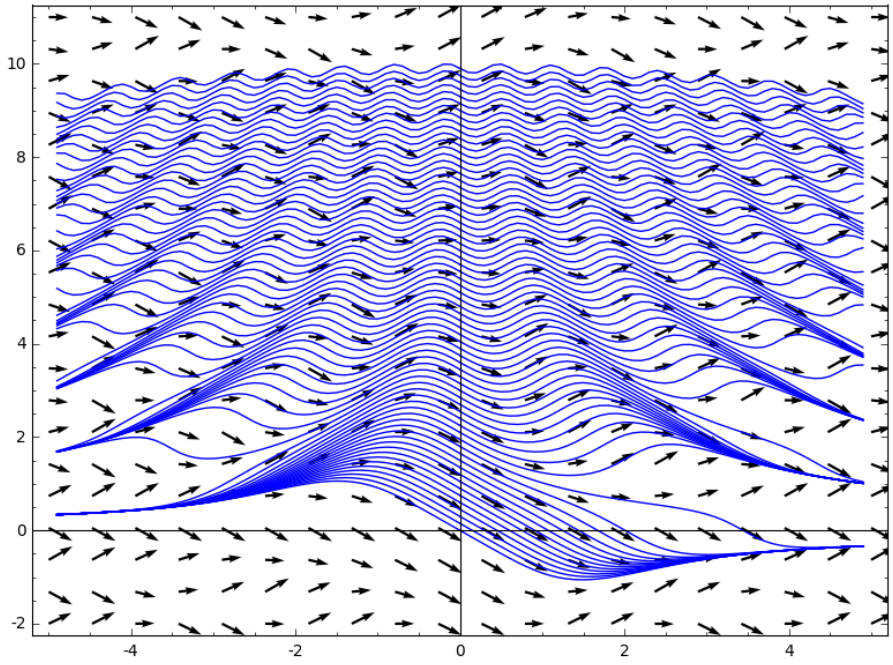


Abb. 4.11 - Zeichnung der Integralkurven von $y'(t) + \cos(y(t) \cdot t) = 0$.

Die Funktion `odeint` übernimmt als Argument das zweite Glied der Differentialgleichung (aufgelöst geschrieben), eine oder mehrere Anfangsbedingungen und auch das Lösungsintervall; sie gibt eine Tabelle des Typs `numpy.ndarray` zurück, die wir mit dem Befehl `flatten`[1] vereinfachen, wie das in Unterabschnitt 3.3.2 zu sehen war, und die wir mit dem Befehl `zip` mit der Tabelle `t` zuaammenführen, bevor wir die Näherungslösung zeichnen. Um das Feld der Tangenten an die Integralkurven hinzuzufügen, verwenden wir den Befehl `plot_vector_field`.

Beispiel (*Räuber-Beute-Modell von Lotka-Volterra.*) Wir möchten die graphische Lösung einer Population von Beute und Räubern darstellen, die einem Gleichungssystem von Lotka-Volterra folgt:

$$\begin{cases} \dfrac{du}{dt} = au - buv \\ \dfrac{dv}{dt} = -cv + dbuv \end{cases}$$

[1] Wir könnten hier auch die Funktion `ravel` von NumPy nehmen, welche die Erzeugung eines neuen Objekts vermeidet und daher den Speicherbedarf optimiert.

wobei u die Anzahl der Beutetiere (z.B. Hasen) bezeichnet, v die Anzahl der Räuber (z.B. Füchse). Außerdem sind a, b, c, d Parameter, die die Entwicklung der Populationen beschreiben: a charakterisiert das natürliche Wachstum der Anzahl der Hasen bei Abwesenheit von Füchsen, b die Abnahme der Anzahl der Hasen bei Anwesenheit der Räuber, c die Abnahme der Anzahl der Füchse bei Fehlen der Beutetiere, und d zeigt schließlich an, wieviel Hasen erforderlich sind, damit ein neuer Fuchs erscheint.

```
sage: import scipy; from scipy import integrate
sage: a, b, c, d = 1., 0.1, 1.5, 0.75
sage: def dX_dt(X, t=0):           # meldet das Anwachsen der Populationen
....:     return [a*X[0] - b*X[0]*X[1], -c*X[1] + d*b*X[0]*X[1]]
sage: t = srange(0, 15, 0.01)                                  # Zeitskala
sage: X[0] = [10, 5]           # Anfangsbedingungen: 10 Hasen und 5 Füchse
sage: X = integrate.odeint(dX_dt, X0, t)               # numerische Lösung
sage: hasen, fuechse = X.T                    # Abkürzung von X.transpose()
sage: p = line(zip(t, hasen), color='red')  # Verlauf der Anzahl der Hasen
sage: p += text("Hasen",(12,37), fontsize=10, color='red')
sage: p += line(zip(t, fuechse), color='blue')       # dito für die Füchse
sage: p += text("Füchse",(12,7), fontsize=10, color='blue')
sage: p.axes_labels(["Zeit", "Population"]); p.show(gridlines=True)
```

Die obigen Anweisungen zeigen die Entwicklung der Anzahl der Hasen und Füchse als Funktion der Zeit (Abb. 4.12 links) und die hierunter zeigen das Vektorfeld (Abb. 4.12 rechts):

```
sage: n = 11; L = srange(6, 18, 12/n); R = srange(3, 9, 6/n)
sage: CI = zip(L, R)                # Liste der Anfangsbedingungen
sage: def g(x,y):
....:     v = vector(dX_dt([x, y]))    # zwecks besserer Lesbarkeit
.....:    return v/v.norm()          # normieren wir das Vektorfeld
sage: x, y = var('x y')
sage: q = plot_vector_field(g(x, y), (x, 0, 60), (y, 0, 36))
sage: for j in range(n):
....:     X = integrate.odeint(dX_dt, CI[j], t)        # graphische
....:     q += line(X, color=hue(.8-float(j)/(1.8*n))) #     Lösung
sage: q.axes_labels(["Hasen","Fuechse"]); q.show()
```

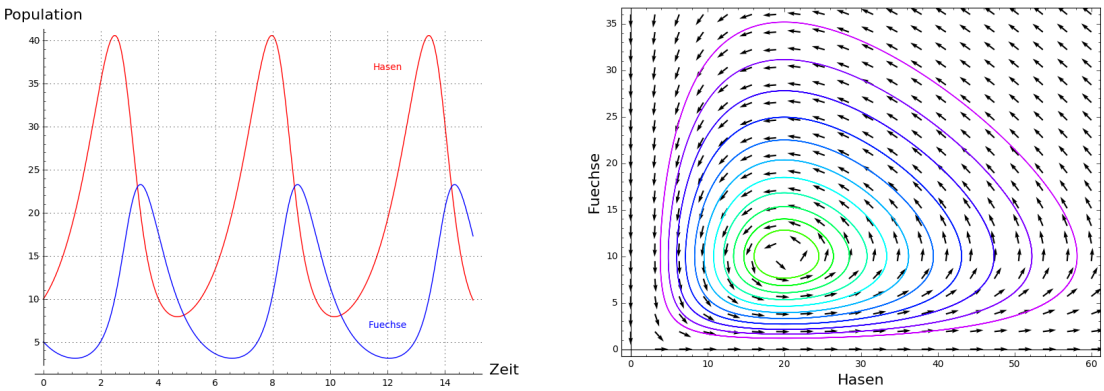


Abb. 4.12 - Untersuchung Räuber-Beute-eines Systems

Übung 15 *(Modell Räuber-Beute).* Reproduzieren Sie die Graphik links in Abb. 4.12 mit `desolve_system_rk4` anstelle von `odeint`.

Übung 16 *(Ein autonomes Differentialsystem).* Zeichnen Sie die Integralkurven des folgenden Differentialsystems:

$$\begin{cases} \dot{x} = y \\ \dot{y} = 0.5y - x - y^3 \end{cases}$$

Übung 17 *(Strömung um einen Zylinder mit Magnus-Effekt).* Wir überlagern einer einfachen Strömung um einen Zylinder mit dem Radius a einen Wirbel mit dem Parameter α, der die zirkulare Geschwindigkeitskomponente verändert. Wir stellen uns in ein Koordinatensystem mit dem Ursprung im Zylinder, und wir arbeiten mit Zylinderkoordinaten in der Ebene $z = 0$, anders gesagt mit Polarkoordinaten. Die Geschwindigkeitskomponenten sind dann:

$$v_r = v_0 \cos(\theta)\left(1 - \frac{a^2}{r^2}\right) \quad \text{und} \quad v_\theta = v_0 \sin(\theta)\left(1 + \frac{a^2}{r^2}\right) + 2\frac{\alpha a v_0}{r}.$$

Die Stromlinien (verschmolzen mit den Trajektorien, denn die Strömung ist stationär) sind parallel zur Geschwindigkeit. Wir suchen einen parametrisierten Ausdruck für die Stromlinien; man muss dazu das Differentialsystem lösen:

$$\frac{dr}{dt} = v_r \quad \text{und} \quad \frac{d\theta}{dt} = \frac{v_0}{r}.$$

Wir benutzen dimensionslose Koordinaten, d.h. bezogen auf a, den Zylinderradius, den wir zu $a = 1$ setzen. Zu zeichen sind die Stromlinien dieser Strömung für $\alpha \in \{0.1, 0.5, 1, 1.25\}$.

Die Nutzung des Magnus-Effekts ist für die Entwicklung von Antriebssystemen vorgeschlagen worden, die aus großen senkrecht stehenden rotierenden Zylindern bestehen, die eine Querkraft erzeugen können, wenn der Wind seitlich auf das Schiff trifft (das war bei dem von Anton Flettner entwickelten Schiff Baden-Baden der Fall, die 1926 den Atlantik überquerte).

4.1.7. Evolute einer Kurve

Wir geben nun ein Beispiel für das Zeichnen einer Evolute eines parametrisierten Bogens (wir erinnern uns, dass die Evolute die Hüllkurve der Normalen ist oder, äquivalent dazu, der geometrische Ort der Krümmungsmittelpunkte der Kurve).

Beispiel *(Evolute der Parabel).* Wir suchen die Gleichung der Evolute der Parabel $\mathcal{P}$ mit der Gleichung $y = x^2/4$ und zeichnen in dieselbe Graphik die Parabel $\mathcal{P}$, einige Normalen an $\mathcal{P}$ und ihre Evolute.

Zur Bestimmung eines Systems von parametrischen Gleichungen $(x(t), y(t))$ der Evolute einer Familie von Geraden Δ_t, die durch kartesische Gleichungen der Form $\alpha(t)X + \beta(t)y = \gamma(t)$ definiert sind, drücken wir den Sachverhalt, dass die Gerade Δ_t Tangente an die Evolute im Punkt $(x(t), y(t))$ ist, so aus:

$$\alpha(t)x(t) + \beta(t)y(t) = \gamma(t) \tag{4.1}$$
$$\alpha(t)x'(t) + \beta(t)y'(t) = 0 \tag{4.2}$$

Wir leiten Gl. (4.1) ab und durch Kombination mit Gl. (4.2) bekommen wir das System:

$$\begin{aligned} \alpha(t)x(t) + \beta(t)y(t) &= \gamma(t) \\ \alpha(t)x'(t) + \beta(t)y'(t) &= \gamma'(t) \end{aligned} \tag{4.3}$$

In unserem Fall hat die Normale $N(t)$ an die Parabel $\mathcal{P}$ in $M(t, t^2/4)$ als Normalenvektor $\vec{v} = (1, t/2)$ (der Tangentenvektor an die Parabel); ihre Gleichung ist daher:

$$\begin{pmatrix} x - t \\ y - t^2/4 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ t/2 \end{pmatrix} = \Leftrightarrow x + \frac{t}{2} y = t + \frac{t^3}{8},$$

anders gesagt, $(\alpha(t), \beta(t), \gamma(t)) = (1, t/2, t^3/8)$. Wir können das obige System nun mit der Funktion `solve` lösen:

```
sage: x, y, t = var('x y t')
sage: alpha(t) = 1; beta(t) = t/2; gamma(t) = t + t^3/8
sage: env = solve([alpha(t)*x + beta(t)*y == gamma(t),\
....:     diff(alpha(t), t)*x + diff(beta(t), t)*y == \
....:     diff(gamma(t), t)], [x,y])
```

$$\left[\left[x = -\frac{1}{4}t^3, y = \frac{3}{4}t^2 + 2\right]\right]$$

Daraus ergibt such eine parametrische Darstellung der Hüllkurve der Normalen:

$$\begin{cases} x(t) = -\frac{1}{4}t^3 \\ y(t) = 2 + \frac{3}{4}t^2 \end{cases}$$

Nun können wir die geforderte Darstellung ausführen und zeichnen dazu einige Normalen an die Parabel (genauer: wir zeichnen Strecken $[M, M + 18\vec{n}]$, worin $M(u, u^2/4)$ ein Punkt von $\mathcal{P}$ ist und $\vec{n} = (-u/2, 1)$ ein Normalenvektor an $\mathcal{P}$):

```
sage: f(x) = x^2/4
sage: p = plot(f, -8, 8, rgbcolor=(0.2,0.2,0.4)) # die Parabel
sage: for u in srange(0, 8, 0.1):   # Normalen an die Parabel
....:     p += line([[u, f(u)], [-8*u, f(u) + 18]], thickness=.3)
....:     p += line([[-u, f(u)], [8*u, f(u) + 18]], thickness=.3)
sage: p += parametric_plot((env[0][0].rhs(),env[0][1].rhs()),\
....:    (t, -8, 8), color='red')     # Zeichnen der Evolute
sage: p.show(xmin=-8, xmax=8, ymin=-1, ymax=12, aspect_ratio=1)
```

Wie weiter oben erwähnt, ist die Evolute einer Kurve auch der Ort ihrer Krümmungsmittelpunkte. Mit Hilfe der Funktion `circle` zeichnen wir einige Schmiegkreise an die Parabel. Wir wissen, der Krümmungsmittelpunkt Ω in einem Kurvenpunkt $M_t = (x(t), y(t))$ hat die Koordinaten:

$$x_\Omega = x - y' \frac{x'^2 + y'^2}{x'y'' - x''y'} \quad \text{und} \quad y_\Omega = y + x' \frac{x'^2 + y'^2}{x'y'' - x''y'}$$

und der Krümmungsradius in M_t ist:

$$R = \frac{(x'^2 + y'^2)^{\frac{3}{2}}}{x'y'' - x''y'}.$$

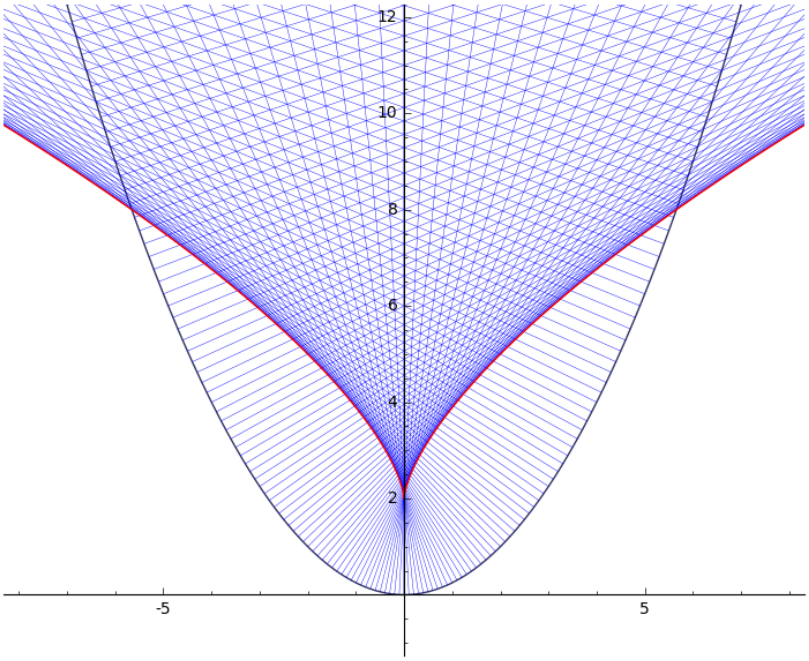


Abb. 4.13 - Die Evolute einer Parabel

Art der Zeichnung	
Graph einer Funktion	`plot`
parametrisierte Kurve	`parametric_plot`
durch eine Polargleichung definierte Kurve	`polar_plot`
durch eine implizite Gleichung definierte Kurve	`implicit_plot`
Höhenlinien einer komplexen Funktion	`complex_plot`
leeres graphisches Objekt	`Graphics()`
Integralkurven einer Differentialgleichung	`odeint, desolve_rk4`
Stabdiagramm	`bar_chart`
Häufigkeitsverteilung einer statistischen Reihe	`plot_histogramm`
Zeichnen eines Kurvenstücks	`line`
Zeichnen einer Punktwolke	`points`
Kreis	`circle`
Polygon	`polygon`
Text	`text`

Tab. 4.1 - Zusammenfassung der graphischen Funktionen in 2D

```
sage: t = var('t'); p = 2
sage: x(t) = t; y(t) = t^2/(2*p); f(t) = [x(t), y(t)]
sage: df(t) = [x(t).diff(t), y(t).diff(t)]
sage: d2f(t) = [x(t).diff(t, 2), y(t).diff(t, 2)]
sage: T(t) = [df(t)[0]/df(t).norm(), df[1](t)/df(t).norm()]
sage: R(t) = (df(t).norm())^3/(df(t)[0]*d2f(t)[1]-df(t)[1]*d2f(t)[0])
sage: Omega(t) = [f(t)[0] + R(t)*N(t)[0], f(t)[1] + R(t)*N(t)[1]]
sage: g = parametric_plot(f(t), (t, -8, 8), color='green', thickness=2)
sage: for u in srange(.4, 4, .2):
....:     g += line([f(t=u), Omega(t=u)], color='red', alpha=.5)
....:     g += circle(Omega(t=u), R(t=u), color='blue')
sage: g.show(aspect_ratio=1, xmin=-12, xmax=7, ymin=-3, ymax=12)
```

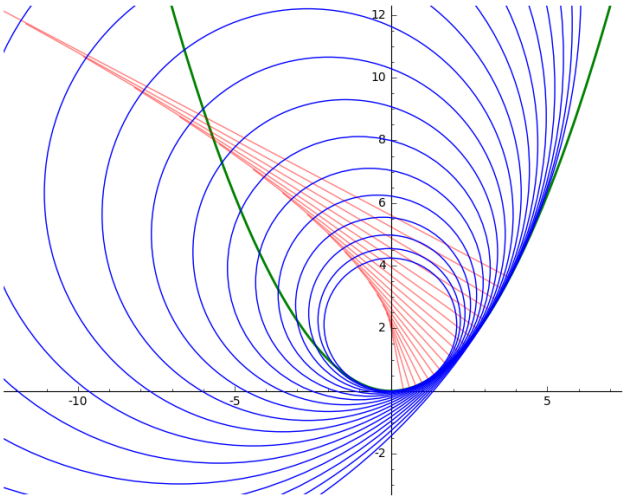


Abb. 4.14 - Schmiegkreise an die Parabel

Die Tabelle 4.1 stellt die in diesem Abschnitt verwendeten Funktionen zusammen. Wir führen dort auch den Befehl `text` auf, mit dem wir eine Zeichenkette in eine Graphik einfügen können, wie auch den Befehl `polygon`, mit dem wir Vielecke zeichnen können.

4.2. Graphik in 3D

Zum Zeichnen von Flächen in drei Dimensionen verfügen wir über den Befehl `plot3d(f(x,y), (x,a,b), (y,c,d))`. Die erhaltene Fläche wird mit der Voreinstellung dank der Anwendung *Jmol* visualisiert; wir können aber auch den *Tachyon 3D Ray Tracer* mit Hilfe des optionalen Arguments `viewer='tachyon'` des Befehls `show` verwenden. Es folgt ein erstes Beispiel für die Zeichnung einer parametrisierten Fläche (Abb. 4.15).

```
sage: u, v = var('u v')
sage: h = lambda u,v: u^2 + 2*v^2
sage: f = plot3d(h, (u,-1,1), (v,-1,1), aspect_ratio=[1,1,1])
sage: f.show(viewer='tachyon')
```

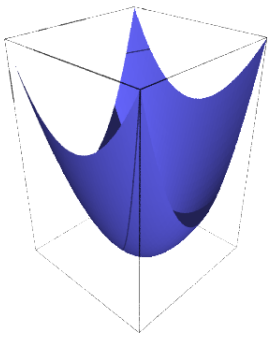


Abb. 4.15 - Die mit $(u, v) \mapsto u^2 + 2v^2$ parametrisierte Fläche

Die Visualisierung der Fläche einer Funktion zweier Variablen ermöglicht die Diskussion einer solchen Funktion, wie wir das im nächsten Beispiel sehen werden.

Beispiel *(Eine diskontinuierliche Funktion, deren Richtungsableitungen überall existieren).* Zu untersuchen ist die Existenz der Richtungsableitungen in $(0,0)$ und die Stetigkeit der Funktion f von $\mathbb{R}^2$ nach $\mathbb{R}$, die definiert ist durch:

$$f(x,y) = \begin{cases} \frac{x^2 y}{x^4+y^2} & \text{für } (x,y) \neq (0,0) \\ 0 & \text{für } (x,y) = (0,0) \end{cases}$$

Für $H = \begin{pmatrix} \cos\theta \\ \sin\theta \end{pmatrix}$ ist die Funktion $\varphi(t) = f(tH) = f(t\cos\theta.t\sin\theta)$ in $t = 0$ für jeden Wert von θ differenzierbar:

```
sage: f(x, y) = x^2*y/(x^5 + y^2)
sage: t, theta = var('t theta')
sage: limit(f(t*cos(theta), t*sin(theta))/t, t=0)
cos(theta)^2/sin(theta)
```

Daher lässt f an der Stelle $(0,0)$ Richtungsableitungen nach beliebigen Vektoren zu. Für eine bessere Darstellung der Fläche von f kann man damit beginnen, einige Höhenlinien zu suchen; beispielsweise die zum Wert $\frac{1}{2}$ gehörende Höhenlinie:

```
sage: solve(f(x,y)==1/2,y)
[y == x^2]
sage: a = var('a'); h = f(x, a*x^2).simplify_rational(); h
a/(a^2 + 1)
```

Entlang der Parabel mit der Gleichung $y = ax^2$, ausgenommen der Ursprung, hat f den konstanten Wert $f(x, ax^2) = \frac{a}{1+a^2}$. Wir zeichnen daraufhin die Funktion $h : a \mapsto \frac{a}{1+a^2}$:

```
sage: plot(h, a, -4, 4)
```

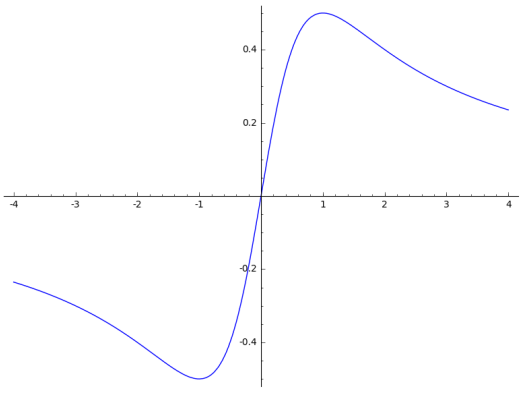


Abb. 4.16 - Ein vertikaler Schnitt durch die untersuchte Fläche

Die Funktion h erreicht ihr Maximum bei $a = 1$ und ihr Minimum bei $a = -1$. Die Beschränkung von f auf die Parabel mit der Gleichung $y = x^2$ entspricht dem „Kammweg“, der sich in der Höhe $\frac{1}{2}$ befindet; was die Restriktion von f auf die Parabel mit der Gleichung $y = -x^2$ angeht, so entspricht sie dem „Talweg“, der sich in der Höhe $-\frac{1}{2}$ befindet. Abschließend können

wir, so dicht wir an $(0,0)$ auch sein mögen, Punkte finden, an denen f die Werte $\frac{1}{2}$ bzw. $-\frac{1}{2}$ annimmt. Infolgedessen ist die Funktion im Ursprung nicht stetig.

```
sage: p = plot3d(f(x,y), (x,-2,2), (y,-2,2), plot_points=[150,150])
sage: p.show(viewer='tachyon')
```

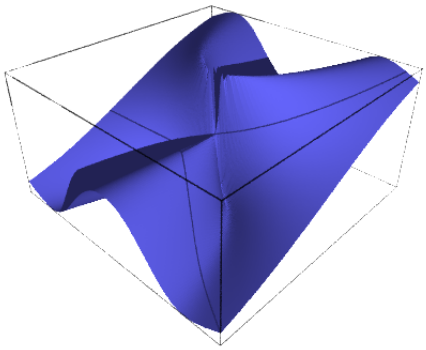


Abb. 4.17 - Die Fläche von $f : (x,y) \mapsto \frac{x^2 y}{x^4+x^2}$.

Wir hätten ebensogut horizontale Ebenen zeichnen können, um die Höhenlinien dieser Funktion sichtbar zu machen, indem wir diesen Code ausführen:

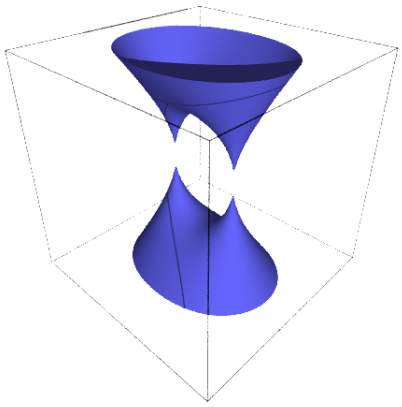
```
sage: for i in range(1,4):
....:     p += plot3d(-0.5 + i/4, (x, -2, 2), (y, -2,2),\
....:                 color=hue(i/10), opacity=.1)
sage: p.show(viewer='tachyon')
```

Von den übrigen Befehlen für 3D-Graphik erwähnen wir `implicit_plot3d`, der uns erlaubt Flächen zu zeichnen, die durch eine implizite Gleichung der Form $f(x,y,z) = 0$ definiert sind. Als Beispiel wollen wir die cassinische Fläche zeichnen (Abb. 4.18a), die durch die implizite Gleichung $(a^2 + x^2 + y^2)^2 = 4a^2x^2 + z^4$ definiert ist:

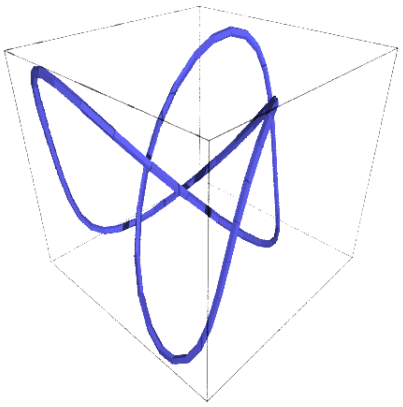
```
sage: x, y, z = var('x y z'); a = 1
sage: h = lambda x, y, z: (a^2+x^2+y^2)^2-4*a^2*x^2-z^4
sage: f = implicit_plot3d(h, (x, -3, 3), (y, -3, 3), (z, -2, 2),\
....:                 plot_points=100, adaptative=True)
sage: f.show(viewer='tachyon')
```

Schließlich geben wir noch das Beispiel einer Raumkurve (Abb. 4.18b) mittels des Befehls `line3d`:

```
sage: g1 = line3d([(-10*cos(t)-2*cos(5*t)+15*sin(2*t),\
....:      -15*cos(2*t)+10*sin(t)-2*sin(5*t),\
....:      10*cos(3*t)) for t in srange(0,6.4,.1)], radius=0.5)
sage: g1.show(viewer='tachyon')
```



(a) Die cassinische Fläche



(b) Ein Knoten im Raum

5. Definitionsmengen für das Rechnen

Das Schreiben mathematischer Texte auf Papier oder Tafel erfordert vor allem die Suche nach einem guten Kompromiss zwischen Flexibilität, Leichtigkeit der Schreibweise und Strenge. Das ist für den tagtäglichen Gebrauch eines Berechnungssystems nicht anders. Sage versucht die Wahl des Kompromisses dem Anwender zu überlassen insbesondere dadurch, dass es ihm mehr oder weniger streng erlaubt, den Definitionbereich der Rechnung festzulegen: welches ist die Natur der betrachteten Objekte, zu welcher Menge gehören sie, welche Operationen sind mit ihnen ausführbar?

5.1. Sage ist objektorientiert

Python und Sage machen von der objektorientierten Programmierung intensiven Gebrauch. Auch wenn das für den normalen Gebrauch relativ transparent bleibt, ist es sinnvoll, darüber ein Minimum an Wissen zu haben, zumal dies in mathematischen Zusammenhängen ganz natürlich ist.

5.1.1. Objekte, Klassen und Methoden

Die objektorientierte Programmierung beruht auf der Idee, jede physikalische oder abstrakte Entität zu modellieren, die wir in einem Programm durch ein *Objekt* genanntes Sprachkonstrukt manipulieren möchten. Meistens, und in Python ist das der Fall, ist jedes Objekt Instanz einer *Klasse*. So ist die rationale Zahl 12/35 modelliert durch ein Objekt, welches eine Instanz der Klasse `Rational` ist.

```
sage: o = 12/35
sage: print type(o)
<type 'sage.rings.rational.Rational'>
```

Merken wir uns, dass diese Klasse dem Objekt 12/35 zugeordnet ist und nicht der Variablen `o`, die es enthält:

```
sage: print type(12/35)
<type 'sage.rings.rational.Rational'>
```

Präzisieren wir die Definitionen. Ein *Objekt* ist ein Teil des Speichers des Rechners, das die erforderliche Information für die Darstellung der Entität enthält, die vom Objekt modelliert wird. Was die Klasse betrifft, so definiert sie zwei Dinge:

1. die *Datenstruktur* eines Objektes, d.h. wie die Information im Speicherblock organisiert ist. Beispielsweise spezifiziert die Klasse `Rational`, dass eine rationale Zahl wie 12/35 durch zwei ganze Zahlen dargestellt wird: ihren Zähler und ihren Nenner;

2. ihr *Verhalten*, insbesondere die *Operationen* auf diesem Objekt: beispielsweise wie man den Zähler einer rationalen Zahl bekommt, wie ihr Absolutwert berechnet wird, wie man zwei rationale Zahlen addiert oder multipliziert. Jede dieser Operationen wird durch eine *Methode* implementiert (hier durch `numer` und `numerator, abs, __mult__, __add__`).

Um eine ganze Zahl zu faktorisieren wird man daher die Methode `factor` mit folgender Syntax aufrufen[1]:

```
sage: o = 720
sage: o.factor()
2^4 * 3^2 * 5
```

was man lesen kann als: „nimm den Wert von `o` und wende die Methode `factor` ohne ein weiteres Argument darauf an“. Unter der Haube führt Python folgende Rechnung aus:

```
sage: type(o).factor(o)
2^4 * 3^2 * 5
```

Von links nach rechts: „fordere von der Klasse von `o` (`type(o)`) die Methode `factor` die für die Faktorisierung geeignete Methode an (`type(o).factor`) und wende sie auf `o` an“.

Wir bemerken bei dieser Gelegenheit, dass wir eine Methode auf einen Wert anwenden können, ohne ihn an eine Variable zu übergeben:

```
sage: 720.factor()
2^4 * 3^2 * 5
```

und daher auch Operationen von links nach rechts hintereinander ausführen können:

```
sage: o = 720/133
sage: o.numerator().factor()
2^4 * 3^2 * 5
```

5.1.2. Objekte und Polymorphie

Was geht uns das an? Zunächst sind in Sage alle Operationen *polymorph*, d.h. sie können auf verschiedene Typen angewendet werden. So wenden wir auf welches Objekt auch immer, das wir „faktorisieren“ wollen, dieselbe Schreibweise `o.factor()` an (oder seine Abkürzung `factor(o)`). Die auszuführenden Operationen sind dennoch im Fall einer ganzen Zahl und eines Polynoms nicht die gleichen! Sie unterscheiden sich außerdem auch darin, ob das Polynom rationale Koeffizienten hat oder Koeffizienten aus einem endlichen Körper. Es ist die Klasse des Objektes, die entscheidet, welche Version von `factor` letztendlich ausgeführt wird.

[1] Für die Bequemlichkeit des Anwenders bietet Sage auch eine Funktion `factor` derart, dass `factor(o)` eine Abkürzung von `o.factor()` ist. Das gilt auch für eine große Anzahl häufig gebrauchter Funktionen, es ist durchweg möglich, ihre Abkürzungen zu nehmen.

Ebenso, und wir übernehmen die normale mathematische Schreibweise, kann das Produkt zweier Objekte `a` und `b` immer `a*b` geschrieben werden, auch wenn die in jedem der Fälle verwendeten Algorithmen verschieden[2] sind. Hier ein Produkt zweier ganzer Zahlen:

```
sage: 3*7
21
```

hier ein Produkt zweier rationaler Zahlen, das durch Multiplikation der Zähler und der Nenner und anschließendes Kürzen erhalten wird:

```
sage: (2/3)*(6/5)
4/5
```

und hier ein Produkt zweier komplexer Zahlen, wobei die Beziehung $i^2 = -1$ benutzt wird:

```
sage: (1 + I)*(1 - I)
2
```

sowie kommutative Produkte zweier symbolischer Ausdrücke:

```
sage: (x + 1)*(x + 2)
(x + 2)*(x + 1)
sage: (x + 2)*(x + 1)
(x + 2)*(x + 1)
```

Außer der einfachen Schreibweise ermöglicht diese Form der Polymorphie das Schreiben *generischer*, d.h. allgemeingültiger Programme, die auf jedes Objekt anwendbar sind, das diese Operation zulässt (hier die Multiplikation):

```
sage: def vierte_potenz(a)
....:     a = a*a
....:     a = a*a
....:     return a

sage: vierte_pozenz(2)
16
sage: vierte_potent(3/2)
81/16
sage: vierte_potenz(I)
1
sage: vierte_potenz(x+1)
(x + 1)^4
sage: M = matrix([[0, -1], [1, 0]]); M
[ 0 -1]
[ 1  0]
sage: vierte_potenz(M)
[1 0]
[0 1]
```

[2] Für eine binäre arithmetische Operation wie das Produkt ist das Verfahren zur Auswahl der passenden Methode ein wenig komplizierter als das oben beschriebene. Sie muss nämlich gemischte Operationen wie die Summe $2 + 3/4$ einer ganzen und einer rationalen Zahl managen. Hier wird die 2 vor der Addition in eine rationale Zahl $2/1$ konvertiert. Die Regeln für die Entscheidung, welche Operanden umzuwandeln sind und wie, heißen *Zwangsmodell*.

5.1.3. Introspektion

Die Objekte von Python, und damit auch von Sage, haben die Funktionalitäten der *Introspektion*. Das bedeuten, dass wir ein Objekt bei der Ausführung nach seiner Klasse und seinen Methoden „befragen“ können und auch die erhaltenen Informationen mit den üblichen Konstruktionen der Programmierung manipulieren. So ist die Klasse eines Objektes ihrerseits ein Python-Objekt wie (fast) alle anderen, die wir mit `type(o)` gewinnen können:

```
sage: t = type(5/1); print t
<type 'sage.rings,rational.Rational'>
sage: t == type(5)
False
```

Wir sehen hier, dass der Ausdruck 5/1 die rationale Zahl 5 bildet, die ein anderes Objekt ist als die ganze Zahl 5.

Es gibt auch Werkzeuge der Introspektion, auf die online zugegriffen werden kann, wie hier am Beispiel der Faktorisierung einer ganzen Zahl gezeigt wird:

```
sage: o = 720
sage: o.factor?
...
Definition: o.factor(algorithm='pari', proof=None, ... )

Docstring:

    Return the prime factorization of this integer as a formal
    Factorization object.
...
```

Den Code dieser Funktion zeigt

```
sage: o.factor??
...
def factor(self, algorithm='pari', proof=None, ...):
    ...
    if algorithm not in ['pari', 'kash', 'magma', 'qsieve', 'ecm']:
    ....
```

Wenn wir dazu die technischen Details durchgehen, erkennen wir, dass Sage einen Großteil der Rechnungen an andere Programme delegiert (PARI, kash, Magna).

Ebenso können wir die automatische Vervollständigung nutzen, um zu einem Objekt die Operationen zu erfragen, die wir mit ihm verwenden können. Hier gibt es viele davon; wir zeigen die, welche mit `n` beginnen:

```
sage: o.n<tab>
o.n                    o.next_prime           o.nth_root
o.nbits                o.next_prime_power     o.numerator
o.ndigits              o.next_probable_prime  o.numerical_approx
```

Auch hier handelt es sich um eine Form der Introspektion.

5.2. Elemente, Vorfahren, Kategorien

5.2.1. Elemente und Vorfahren

Im vorigen Abschnitt sind wir dem Begriff der *Klasse* einnes Objektes begegnet. In der Praxis genügt es zu wissen, dass dieser Begriff existiert; selten muss man den Typ eines Objektes explizit betrachten. Andererseits führt Sage ein Gegenstück zu diesem Begriff ein, dem wir und nun widmen werden, dem des *Vorfahren* eines Objektes.

Zum Beispiel nehmen wir an, wir wollten entscheiden, ob ein Element a *invertierbar* ist. Die Antwort wird nicht nur vom Element selbst abhängen, sondern auch von der Menge A, der es angehört (wie auch sein potentielles Inverses). Die Zahl 5 ist beispielsweise in der Menge $\mathbb{Z}$ der ganzen Zahlen nicht invertierbar, weil ihr Inverses 1/5 keine ganze Zahl ist:

```
sage: a = 5; a
5
sage: a.is_unit()
False
```

Vielmehr ist 5 in der Menge der rationalen Zahlen invertierbar:

```
sage: a = 5/1; a
5
sage: a.is_unit()
True
```

Sage antwortet auf diese beiden Fragen unterschiedlich, denn wir haben im vorigen Abschnitt gesehen, dass die Elemente 5 und 5/1 Instanzen verschiedener Klassen sind.

In einigen CAS wie MuPAD oder Axiom wird die Menge X, zu der x gehört (hier $\mathbb{Z}$ oder $\mathbb{Q}$), einfach durch die Klasse von x modelliert. Sage folgt dem Ansatz von Magma und modelliert X durch ein ergänzendes, mit x verbundenes Objekt, seinem *Vorfahren*:

```
sage: parent(5)
Integer Ring
sage: parant(5/1)
Rational Field
```

Wir können diese beiden Mengen auch mit ihren Abkürzungen finden:

```
sage: ZZ
Integer Ring
sage: QQ
Rational Field
```

und sie für die einfache *Konvertierung* eines Elementes von der einen in die andere benutzen, sofern das sinnvoll ist:

```
sage: QQ(5).parent()
Rational Field
sage: ZZ(5/1).parent()
Integer Ring
sage: ZZ(1/5)
```

```
Traceback (most recent call last)
  ...
TypeError: no conversion of this rational to integer
```

Allgemein versucht die Syntax `P(x)`, wobei `P` ein Vorfahr ist, das Objekt `x` in ein Element von `P` zu konvertieren. Für die ganze Zahl $1 \in \mathbb{Z}$, die rationale Zahl $1 \in \mathbb{Q}$, die Gleitpunkt-Näherung der reellen Zahl $1,0 \in \mathbb{R}$ oder die komplexe Zahl $1,0 + 0,0i \in \mathbb{C}$ gilt:

```
sage: ZZ(1), QQ(1), RR(1), CC(1)
(1, 1, 1.00000000000000, 1.00000000000000)
```

Übung 18. Welches ist die Klasse des Ringes der ganzen Zahlen $\mathbb{Z}$?

5.2.2. Konstruktionen

Die Vorfahren sind ihrerseits Objekte, mit denen wir Operationen ausführen können. So können wir das kartesische Produkt $\mathbb{Q}^2$ bilden:

```
sage: cartesian_product([QQ, QQ])
The Cartesian product of (Rational Field, Rational Field)
```

$\mathbb{Q}$ zurückgewinnen als Körper der Brüche von $\mathbb{Z}$:

```
sage: ZZ.fraction_field()
Rational Field
```

den Polynomring in x mit Koeffizienten aus $\mathbb{Z}$:

```
sage: ZZ['x']
Univariate Polynomial Ring in x over Integer Ring
```

Durch schrittweises Vorgehen können wir komplexe algebraische Strukturen bilden wie den Vektorraum der 3×3-Matrizen mit polynomialen Koeffizienten aus einem endlichen Körper:

```
sage: Z5 = GF(5); Z5
Finite Field of size 5
sage: P = Z5['x']; P
Univariate Polynomial Ring in x over Finite Field of size 5
sage: M = MatrixSpace(P, 3, 3); M
Full MatrixSpace of 3 by 3 dense matrices over Univariate Polynomial
Ring in x over Finite Field of size 5
```

Hier ein Element:

```
sage: M.random_element()
[          x + 4         2*x + 1   3*x^2 + x + 2]
[3*x^2 + 4*x + 1           x + 2 4*x^2 + 3*x + 2]
[4*x^2 + 2*x + 2   x^2 + 2*x + 3 3*x^2 + 4*x + 4]
```

5.2.3. Ergänzung: Kategorien

Ein Vorfahr hat im allgemeinen selbst keinen Vorfahren, sondern eine *Kategorie*, die seine Eigenschaften angibt:

```
sage: QQ.category()
Category of quotient fields
```

In der Tat weiß Sage, dass $\mathbb{Q}$ ein Körper ist:

```
sage: QQ in Fields
True
```

und somit beispielsweise auch eine additive kommutative Gruppe ist:

```
sage: QQ in CommutativeAdditiveGroups()
True
```

Daraus folgt, dass $\mathbb{Q}[x]$ ein euklidischer Ring ist:

```
sage: QQ['x'] in EuclideanDomains()
True
```

Alle diese Eigenschaften werden bei strengen und effizienten Rechnungen mit Elementen dieser Systeme verwendet.

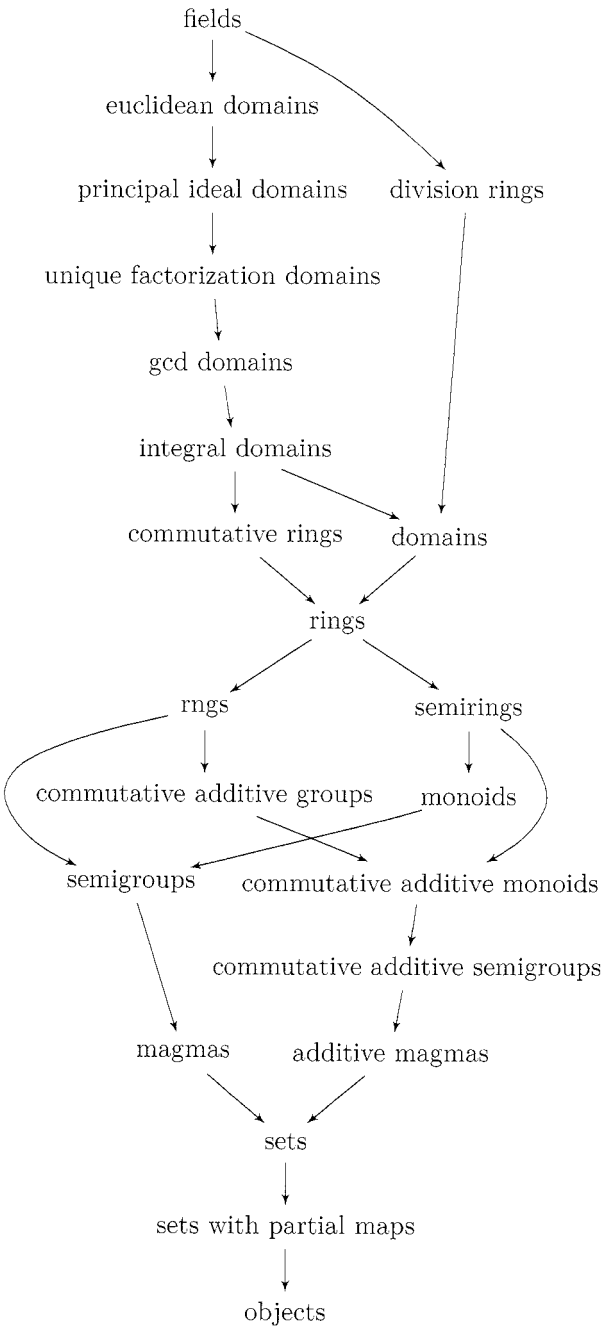


Abb. 5.1 - Ein kleiner Ausschnitt aus der Darstellung der Kategorien in Sage

5.3. Definitionsmengen für das Rechnen und die Darstellung in Normalform

Wir gehen nun über zur Betrachtung einiger Vorfahren, die in Sage zu finden sind.

In Abschnitt 2.1 haben wir die Bedeutung der Normalformen für das symbolische Rechnen erkannt, die uns die Entscheidung ermöglichen, ob zwei Objekte bei Vergleich ihrer Darstellungen mathematisch gleich sind. Jeder der hiernach aufgeführten grundlegenden Vorfahren entspricht einer *Menge von Zahlen für die Rechnung mit Normalformen*, d.h. einer Menge

solcher mathematischer Objekte, die eine Normalform erlauben. Das ermöglicht Sage, die Elemente jedes dieser Vorfahren unzweideutig darzustellen[3].

Einige Basistypen von Python	
ganze Zahlen in Python	`int`
Gleitpunktzahlen in Python	`float`
Wahrheitswerte `(True, False)`	`bool`
Zeichenketten	`str`
Definitionsmengen von Zahlen	
ganze Zahlen $\mathbb{Z}$	`ZZ` oder `IntegerRing()`
rationale Zahlen $\mathbb{Q}$	`QQ` oder `RationalField()`
Gleitpunktzahlen mit der Genauigkeit p	`Reals(p)` oder `RealField(p)`
komplexe Gleitpunktzahlen mit Genauigkeit p	`Complexes(p)` oder `ComplexField(p)`
Ringe und endliche Körper	
Rest modulo n $\mathbb{Z}/n\mathbb{Z}$	`Integers(n)` oder `IntegerModRing(n)`
endlicher Körper $\mathbb{F}_q$	`GF(q)` oder `FiniteField(q)`
algebraische Zahlen	
algebraische Zahlen $\bar{\mathbb{Q}}$	`QQbar` oder `AlgebraicField()`
reelle algebraische Zahlen	`AA` oder `AlgebraicRealField()`
Körper der Zahlen $\mathbb{Q}[x]/(p)$	`NumberField(p)`
Symbolisches Rechnen	
$m \times n$-Matrizen mit Einträgen aus A	`MatrixSpace(A, m, n)`
Polynome $A[x, y]$	`A['x,y']` oder `PolynomialRing(A, 'x,y')`
Reihen $A[[x]]$	`A[['x']]` oder `PowerSeriesRing(A, 'x')`
symbolische Ausdrücke	`SR`

Tab. 5.1 - Die wichtigsten Zahlenarten für Berechnungen und die Vorfahren

5.3.1. Elementare Zahlenarten

Elementare Zahlenarten nennen wir die klassischen Mengen von Konstanten: ganze Zahlen, rationale Zahlen, Gleitpunktzahlen, boolesche Werte, Reste modulo n ...

Ganze Zahlen. Die ganzen Zahlen werden rechnerintern zur Basis zwei dargestellt und auf dem Bildschirm zur Basis zehn. Wie wir gesehen haben, sind die ganzen Zahlen in Sage vom Typ `Integer`. Ihr Vorfahr ist der Ring $\mathbb{Z}$:

```
sage: 5.parent()
Integer Ring
```

Die ganzen Zahlen werden in Normalform dargestellt; die Gleichheit ist daher einfach zu prüfen. Um ganze Zahlen in faktorisierter Form darstellen zu können, verwendet der Befehl `factor` eine besondere Klasse:

```
sage: print type(factor(4))
<class
'sage.structure.factorization_integer.IntegerFactorization'>
```

[3] Die meisten anderen der in Sage verfügbaren Vorfahren entsprechen Definitionsmengen für die Rechnung mit Normalformen, doch ist dies nicht bei allen der Fall. Es kommt auch vor, dass Sage aus Gründen der Effizienz die Elemente als Normalformen nur auf besondere Anforderung darstellt.

Die Klasse `Integer` ist Sage-eigen: in der Voreinstellung verwendet Python ganze Zahlen des Typs `int`. Generell erfolgt die Konversion des einen in den anderen automatisch, doch kann es notwendig werden, die Konversion explizit vorzuschreiben mit

```
sage: int(5)
5
sage: print type(int(5)) <type 'int'>
```

oder umgekehrt

```
sage: Integer(5)
5
sage: print type(Integer(5))
<type 'sage.rings.integer.Integer'>
```

Rationale Zahlen. Die Eigenschaft der Normalform haben auch die rationalen Zahlen, die Elemente von `QQ`, die immer in Normalform dargetellt werden. So werden durch die Anweisung

```
sage: factorial(99)/factorial(100) -1/50
-1/100
```

Zuerst werden die Fakultäten ausgewertet, dann wird von dem erhaltenen Bruch 1/100 die rationale Zahl 1/50 abgezogen und gekürzt (was hier nicht nötig ist).

Gleitpunktzahlen. Reelle Zahlen können nicht exakt dargestellt werden. Ihre genäherten Zahlenwerte werden als Gleitpunktzahlen dargestellt. Sie werden in Kapitel 11 eingehend behandelt.

In Sage werden Gleitpunktzahlen zur Basis zwei dargestellt. Als eine Folge wird die als 0.1 dargestellte Gleitpunktzahl nicht genau gleich 1/10 sein, denn 1/10 kann zur Basis 2 nicht exakt dargestell werden. Jede Gleitpunktzahl hat ihre eigene Genauigkeit. Der Vorfahr der Gleitpunktzahlen mit p signifikanten Bits wird als `Reals(p)` geschrieben. Die Gleitpunktzahl mit der voreingestellten Präzision ($p = 53$) heißt auch `RR`. Wie im Fall der ganzen Zahlen unterscheiden sich die Gleitpunktzahlen in Sage von denen in Python.

Wenn sie in einer Summe oder einem Produkt zusammen mit ganzen oder rationalen Zahlen auftreten, sind Gleitpunktzahlen „ansteckend“: der gesamte Ausdruck wird dann als Gleitpunktzahl berechnet:

```
sage: 72/53 - 5/3*2.7
-3.14150943396227
```

Ebenso, wenn das Argument einer Funktion eine Gleitpunktzahl ist, erscheint auch das Ergebnis als Gleitpunktzahl:

```
sage: cos(1), cos(1.)
(cos(1), 0.540302305868140)
```

Die Methode `numerical_approx` (oder ihre Alias `n` und `N`) dient zur numerischen Auswertung weiterer Ausdrücke. Ein optionales Argument erlaubt die Präzisierung der bei der Rechnung gebrauchten signifikanten Stellen. Als Beispiel π mit 50 signifikanten Ziffern:

```
sage: pi.n(digits=50)    # Variante: n(pi,digits=50)
3.1415926535897932384626433832795028841971693993751
```

Komplexe Zahlen. Genauso sind die Näherungen von komplexen Zahlen mittels Gleitpunktzahlen mit der Genauigkeit p Elemente von `Complexes(p)`, wobei `CC` die voreingestellte Genauigkeit besitzt. Wir können eine komplexe Zahl bilden und ihr Argument berechnen:

```
sage: z = CC(1,2); z.arg()
1.10714871779409
```

Komplexe symbolische Ausdrücke

Die imaginäre Einheit i (geschrieben `I` oder `i`), die uns in den vorigen Kapiteln schon begegnet ist, ist kein Element von `CC`, sondern ein symbolischer Ausdruck (siehe Unterabschnitt 5.4.1):

```
sage: I.parent()
```
Symbolic Ring

Wir können damit eine komplexe Gleitpunktzahl schreiben, wenn wir eine explizite Konversion vorschreiben:

```
sage: I.parent()
Symbolic Ring
sage: CC(1.+2.*I).parent()
Complex Field with 53 bits precision
```

In der Welt der symbolischen Ausdrücke ergeben die Methoden `real, imag` und `abs` den Realteil, den Imaginärteil bzw. die Norm oder auch den Modul einer komplexen Zahl:

```
sage: z = 3*exp(I*pi/4)
sage: z.real(), z.imag(), z.abs().simplify()
 (3/2*sqrt(2), 3/2*sqrt(2), 3)
```

Boolesche Werte. Logische Ausdrücke bilden auch eine Zahlenart in Normalform, doch ist die Klasse der Wahrheitswerte oder booleschen Werte ein Basistyp von Python ohne Vorfahren in Sage. Die beiden Normalformen sind `True` und `False`:

```
sage: a, b, c = 0, 2, 3
True
```

In Tests und Schleifen werden die zusammengesetzten Bedingungen mit den Operatoren `or` und `and` einfach von links nach rechts ausgewertet. Das bedeutet, dass die Auswertung einer zusammengesetzten Bedingung mit `or` nach dem ersten Teilergebnis `True` beendet wird, ohne die rechts noch folgenden Terme zu berücksichtigen; genauso bei `and` und `False`. So beschreibt der nächste Test die Teilbarkeit $a|b$ von ganzen Zahlen und führt auch bei $a = 0$ nicht zu einem Fehler:

```
sage: a = 0; b = 12; (a == 0) and (b == 0) or (a != 0) and (b%a == 0)
```

Der Operator `not` hat Vorrang vor `and`, das wiederum Vorrang vor `or` hat, und die Tests auf Gleichheit haben Vorrang vor allen booleschen Operatoren. Die beiden folgenden Tests sind deshalb zum vorigen äquivalent:

```
sage: ((a == 0) and (b == 0)) or ((a != 0) and (b%a == 0))
sage: a == 0 and b == 0 or not a == 0 and b%a == 0
```

Des weiteren gestattet Sage Tests auf Enthaltensein in mehrfachen Intervallen genauso, wie das in der Mathematik geschieht:

$x \leq y \leq z$	wird codiert als	`x <= y <= z`
$x = y = z \neq t$		`x == y == z != t`

In einfachen Fällen werden diese Test unmittelbar ausgeführt; wenn nicht, muss mit dem Befehl `bool` die Auswertung erzwungen werden:

```
sage: x, y = var('x, y')
sage: bool((x-y)*(x+y) == x^2 - y^2)
True
```

Reste modulo n. Bei der Definition einer Kongruenz beginnen wir mit der Bildung ihres Vorfahren, des Ringes $\mathbb{Z}/n\mathbb{Z}$:

```
sage: Z4 = IntegerModRing(4); Z4
Ring of integers modulo 4
sage: m = Z4(7); m
3
```

Wie bei Gleitpunktzahlen wird in Rechnungen mit m automatisch in Zahlen modulo 4 umgewandelt. So werden im folgenden Beispiel 3 und 1 automatisch in Elemente von $\mathbb{Z}/n\mathbb{Z}$ konvertiert:

```
sage: 3*m + 1
2
```

Ist p eine Primzahl, können wir $\mathbb{Z}/p\mathbb{Z}$ als Körper bilden:

```
sage: Z3 = GF(3); Z3
Finite Field of size 3
```

In beiden Fällen handelt es sich um Zahlenarten in Normalform: die Reduktionen modulo n oder p werden bei jeder Erzeugung eines Elementes automatisch ausgeführt. Rechnungen auf endlichen Ringen und Körpern werden in Kapitel 6 ausführlich behandelt.

5.3.2. Zusammengesetzte Objektklassen

Mit wohldefinierten Konstanten können Klassen symbolischer Objekte gebildet werden, die als Variable vorkommen und eine Normalform kennen. Die wichtigsten sind die Matrizen, die Polynome, die gebrochen rationalen Ausdrücke und die abbrechenden Reihen.

Die entsprechenden Vorfahren sind durch die Zahlenart der Koeffizienten parametrisiert. So unterscheiden sich beispielsweise Matrizen mit ganzzahligen Einträgen von solchen mit Einträgen aus $\mathbb{Z}/n\mathbb{Z}$ und die zugehörigen Rechenregeln werden automatisch angewendet, ohne dass eine Funktion für die Reduktion modulo n explizit aufgerufen werden müsste.

Teil II dieses Buches ist vor allem diesen Objekten gewidmet.

Matrizen. Die Normalform einer Matrix liegt vor, wenn ihre Einträge ihrerseits in Normalform sind. Auch wird jede Matrix auf einem Körper oder einem Ring in normaler Darstellung automatisch in Normalform ausgegeben:

```
sage: a = matrix(QQ, [[1, 2, 3], [2, 4, 8], [3, 9, 27]])
sage: (a^2 + 1)*a^(-1)
```

```
[  -5 13/2  7/3]
[   7    1 25/3]
[   2 19/2   27]
```

Der Aufruf der Funktion `matrix` ist eine Abkürzung. Intern bildet Sage den entsprechenden Vorfahren, nämlich den Vektorraum der 3×3-Matrizen mit Einträgen aus $\mathbb{Q}$ (in normaler Darstellung), dann wird der Vorfahr zur Erzeugung der Matrix verwendet:

```
sage: M = MatrixSpace(QQ,3,3); M
Full MatrixSpace of 3 by 3 dense matrices over Rational Field
sage: a = M[[1, 2, 3], [2, 4, 8], [3, 9, 27]])
sage: (a^2 + 1)*a^(-1)
[  -5 13/2  7/3]
[   7    1 25/3]
[   2 19/2   27]
```

Die Operationen mit symbolischen Matrizen werden in Kapitel 8 beschrieben, die numerische lineare Algebra in Kapitel 13.

Polynome und gebrochen rationale Ausdrücke. Genau wie Matrizen den Typ ihrer Einträge „kennen“ Polynome in Sage den Typ ihrer Koeffizienten. Ihre Vorfahren sind Polynomringe wie $\mathbb{Z}[x]$ oder $\mathbb{C}[x, y, z]$, die in den Kapiteln 7 und 9 eingehend vorgestellt werden, und die wir erzeugen können mit

```
sage: P = ZZ['x']; P
Univariate Polynomial Ring in x over Integer Ring
sage: F = P.fraction_field(); F
Fraction Field of Univariate Polynomial Ring in x over Integer Ring
sage: p = P(x+1)*P(x); P
x^2 + x
sage: p + 1/p
(x^4 + 2*x^3 + x^2 + 1)/(x^2 + x)
sage: parent(p + 1/p)
Fraction Field of Univariate Polynomial Ring in x over Integer Ring
```

Wie wir in Unterabschnitt 5.4.2 sehen werden, gibt es keine ideale Darstellung für Polynome und gebrochen rationale Ausdrücke. Die Elemente der Polynomringe werden in entwickelter Form dargestellt. Diese Ringe sind daher normal dargestellt, sobald die Koeffizienten in einer Zahlenart ihreseits in Normalform vorliegen.

Diese Polynome unterscheiden sich von polynomialen Ausdrücken, denen wir in Kapitel 2 begegnet sind, und deren Koeeffizienten weder einen wohldefinierten Typ haben noch einen Vorfahren, der den Typ widerspiegelt. Sie repräsentieren eine Alternative zu den „echten“ Polynomen, die nützlich sein kann, um Polynome und andere mathematische Ausdrücke zu mischen. Wir betonen, dass wenn wir mit diesen Ausdrücken arbeiten, dann anders als bei Elementen von Polynomringen, explizit ein Befehl zur Reduktion wie `expand` erfolgen muss, um sie in Normalform zu bringen.

Reihen. Abbrechende Potenzreihen sind Objekte der Form

$$a_0 + a_1 x + a_2 a^2 + \cdots + a_n x^n + \mathcal{O}(x^{n+1})$$

Sie werden beispielsweise für die Darstellung abbrechender Reihenentwicklungen verwendet und ihre Manipulation in Sage wird in Abschnitt 7.5 beschrieben. Der Vorfahr der abbrechenden Reihen in x mit der Genauigkeit n und Koeffizienten aus A ist der Ring $A[[x]]$, der mit `PowerSeriesRing(A, 'x', n)` erzeugt wird. Wie die Polynome haben auch die abbrechenden Reihen eine Entsprechung in der Welt der symbolischen Ausdrücke. Der zugehörige Befehl zur Reduktion auf die Normalform ist `series`.

```
sage: f = cos(x).series(x == 0, 6); 1/f
```

$$\frac{1}{1+(-\frac{1}{2})x^2+\frac{1}{24}x^4+\mathcal{O}\left(x^6\right)}$$

```
sage: (1/f).series(x == 0, 6)
```

$$1+\frac{1}{2}x^2+\frac{5}{24}x^4+\mathcal{O}\left(x^6\right)$$

Algebraische Zahlen. Eine algebraische Zahl ist als Wurzel eines Polynoms definiert. Sobald der Grad eines Polynoms 5 oder größer ist, ist es im allgemeinen nicht möglich, die Wurzeln mit Hilfe der Operatoren $+, -, *, /, \sqrt{}$ explizit hinzuschreiben. Jedoch können zahlreiche Rechnungen mit Wurzeln mit keinen anderen Informationen als dem Polynom selbst sehr wohl ausgeführt werden.

```
sage: k.<a> = NumberField(x^3 + x + 1); a^3; a^4 + 3*a
-a - 1
-a^2 + 2*a
```

Die Manipulation algebraischer Zahlen in Sage wird in diesem Buch nicht detailliert behandelt, einige Beispiele sind jedoch in den Kapiteln 7 und 9 zu finden.

5.4. Ausdrücke versus Zahlenarten für das Rechnen

Es sind also in Sage mehrere Ansätze für die Manipulation von Objekten wie Polynome möglich. Man kann sie als besondere formale Ausdrücke ansehen, wie wir das in den ersten Kapiteln dieses Buches getan haben, man kann aber auch einen speziellen Polynomring einführen und mit dessen Elementen rechnen. Zum Abschluss dieses Kapitels beschreiben wir kurz den Vorfahr der symbolischen Ausdrücke, den Bereich `SR`, sodann erläutern wir anhand einiger Beispiele wie wichtig es ist, die Definitionsmengen für die Rechnung zu kontrollieren und die Unterschiede zwischen diesen Ansätzen zu beachten.

5.4.1. Ausdrücke als Definitionsmenge für das Rechnen

Die symbolischen Ausdrücke bilden selber eine Definitionsmenge für Rechnungen! In Sage ist ihr Vorfahr der *symbolische Ring*:

```
sage: parent(sin(x))
Symbolic Ring
```

was wir auch erhalten könnten mit

```
sage: SR
Symbolic Ring
```

Die Eigenschaften dieses Ringes sind recht verschwommen; er ist kommutativ:

```
sage: SR.category()
Category of commutative rings
```

und die Rechenregeln machen im großen Ganzen die Annahme, dass die Werte aller symbolischen Variablen in $\mathbb{C}$ liegen.

Die Form der Ausdrücke, die wir in `SR` manipulieren (polynomiale, rationale und trigonometrische Ausdrücke), die nicht offensichtlich zu ihrer Klasse oder ihrem Vorfahr gehören, bedarf als Resultat einer Rechnung sehr oft einer manuellen Umformung, zum Beispiel mit `expand, combine, collect` und `simplify`, um in die gewünschte Form gebracht zu werden (siehe Abschnitt 2.1). Für den angemessenen Gebrauch dieser Funktionen müssen wir wissen, welchen Typ der Umformung sie ausführen, auf welche *Unterklassen*[4] von Ausdrücken sie angewendet werden, und welche dieser Unterklassen Definitionsbereiche der Normalform bilden. Somit kann der blinde Gebrauch der Funktion `simplify` zu falschen Ergebnissen führen. Varianten von `simplify` ermöglichen eine zielsichere Vereinfachung.

5.4.2. Beispiele: Polynome und Normalformen

Wir konstruieren den Polynomring $\mathbb{Q}[x_1, x_2, x_3, x_4]$ in vier Variablen:

```
sage: R = QQ['x1,x2,x3,x4']; R
Multivariate Polynomial Ring in x1, x2, x3, x4 over Rational Field
sage: x1, x2, x3, x4 = R.gens()
```

Die Elemente von R werden automatisch in entwickelter Form dargestellt:

```
sage: x1*(x2 - x3)
x1*x2 - x1*x3
```

was eine Normalform ist, wie wir wissen. Insbesondere ist der Test auf 0 in R unmittelbar:

```
sage: (x1 + x2)*(x1 - x2) - (x1^2 - x2^2)
0
```

Das ist nicht immer ein Vorteil. Wenn wir beispielsweise die vandermondesche Determinante $\prod_{1\le i<j\le n}(x_i - x_j)$ bilden:

```
sage: prod((a - b) for (a,b) in Subsets([x1,x2,x3,x4],2))
x1^3*x2^2*x3 - x1^2*x2^3*x3 - x1^3*x2*x3^2 + x1*x2^3*x3^2
+ x1^2*x2*x3^3 - x1*x2^2*x3^3 - x1^3*x2^2*x4 + x1^2*x2^3*x4
+ x1^3*x3^2*x4 - x2^3*x3^2*x4 - x1^2*x3^3*x4 + x2^2*x3^3*x4
+ x1^3*x2*x4^2 - x1*x2^3*x4^2 - x1^3*x3*x4^2 + x2^3*x3*x4^2
+ x1*x3^3*x4^2 - x2*x3^3*x4^2 - x1^2*x2*x4^3 + x1*x2^2*x4^3
+ x1^2*x3*x4^3 - x2^2*x3*x4^3 - x1*x3^2*x4^3 + x2*x3^2*x4^3
```

bekommen wir $4! = 24$ Terme. Dieselbe Konstruktion mit einem Ausdruck verbleibt in der faktorisierten Form und ist viel kompakter und besser lesbar:

[4] im Sinne von Familien, nicht von Pythons Objektklassen

```
sage: x1, x2, x3, x4 = SR.var('x1, x2, x3, x4')
sage: prod((a - b) for (a,b) in Subsets([x1,x2,x3,x4],2))
(x1 - x2)*(x1 - x3)*(x1 - x4)*(x2 - x3)*(x2 - x4)*(x3 - x4)
```

Ebenso ermöglicht eine faktorisierte oder teilfaktorisierte Darstellung eine schnellere Berechnung des ggT. Es wäre jedoch nicht sinnvoll, jedes Polynom automatisch in faktorisierte Form zu bringen, selbst wenn es sich dabei um eine Normalform handelte, denn die Faktorisierung kostet Rechenzeit und macht Additionen kompliziert.

Je nach der gewünschten Rechenart ist die ideale Darstellung eines Elementes nicht immer seine Normalform. Das führt die symbolischen Rechensysteme zu einem Kompromiss mit den Ausdrücken. Etliche grundlegende Vereinfachungen wie das Kürzen von Brüchen oder die Multiplikation mit null werden hier automatisch ausgeführt; die übrigen Transformationen werden der Entscheidung des Anwenders überlassen, dem dafür spezifische Befehle zur Verfügung stehen.

5.4.3. Beispiel: Faktorisierung von Polynomen

Betrachten wir die Faktorisierung des folgenden polynomialen Ausdrucks:

```
sage: x = var('x')
sage: p = 54*x^4 + 36*x^3 - 102*x^2 - 72*x -12
sage: factor(p)
6*(x^2 - 2)*(3*x + 1)^2
```

Ist diese Antwort zufriedenstellend? Zwar handelt es sich um eine Faktorisierung von p, doch hängt ihre Optimalität stark vom Kontext ab! Für den Moment betrachtet Sage p als symbolischen Ausdruck, der sich als polynomial erweist. Er kann beispielsweise nicht wissen, ob wir p als Produkt von Polynomen mit ganzzahligen Koeffizienten faktorisieren möchten oder mit rationalen Koeffizienten.

Um hier die Kontrolle zu übernehmen, werden wir präzisieren, in welcher Menge wir p sehen wollen. Zunächst soll p ein Polynom mit ganzzahligen Koeffizienten sein. Wir definieren daher den Ring $R = \mathbb{Z}[x]$ solcher Polynome:

```
sage: R = ZZ['x']; R
Univariate Polynomial Ring in x over Integer Ring
```

Dann konvertieren wir p in diesen Ring:

```
sage: q = R(p); q
54*x^4 + 36*x^3 - 102*x^2 - 72*x - 12
```

An der Ausgabe sehen wir keinen Unterschied, doch q weiß, dass es ein Element von R ist:

```
sage: parent(q)
Univariate Polynomial Ring in x over Integer Ring
```

Damit ist die Faktorisierung eindeutig:

```
sage: factor(q)
2 * 3 * (3*x + 1)^2 * (x^2 - 2)
```

Wir verfahren genauso auf dem Körper der rationalen Zahlen:

```
sage: R = QQ['x'], R
Univariate Polynomial Ring in x over Rational Field
sage: q = R(p); q
54*x^4 + 36*x^3 - 102*x^2 - 72*x - 12
sage: factor(q)
(54) * (x + 1/3)^2 * (x^2 - 2)
```

In diesem neuen Kontext ist die Faktorisierung wieder eindeutig, doch verschieden von der vorigen.

Suchen wir jetzt eine vollständige Zerlegung auf den komplexen Zahlen mit 16 Bits Genauigkeit:

```
sage: R = ComplexField(16)['x']; R
Univariate Polynomial Ring in x over Complex Field with 16 bits of
precision
sage: q = R(p); q
54.00*x^4 + 36.00*x^3 - 102.0*x^2 - 72.00*x - 12.00
sage: factor(q)
(54.00) * (x - 1.414) * (x + 0.3333)^2 * (x + 1.414)
```

Anders ist es, wenn wir den Körper der rationalen Zahlen ein wenig erweitern; hier fügen wir $\sqrt{2}$ hinzu.

```
sage: R = QQ[sqrt(2)]['x']; R
Univariate Polynomial Ring in x over Number Field in sqrt2 with defining
polynomial x^2 - 2
sage: q = R(p); q
54*x^4 + 36*x^3 - 102*x^2 - 72*x - 12
sage: factor(q)
(54) * (x - sqrt2) * (x + sqrt2) * (x + 1/3)^2
```

Wollen wir schließlich sogar, dass die Koeffizienten modulo 5 sind?

```
sage: R = GF(5)['x']; R
Univariate Polynomial Ring in x over Finite Field of size 5
sage: q = R(p); q
4*x^4 + x^3 + 3*x^2 + 3*x + 3
sage: factor(q)
(4) * (x + 2)^2 * (x^2 + 3)
```

5.4.4. Zusammenfassung

In den vorstehenden Beispielen haben wir illustriert, wie der Anwender das Niveau der Strenge in seinen Rechnungen steuern kann.

Einerseits kann er symbolische Ausdrücke verwenden. Diese Ausdrücke sind Elemente des Ringes `SR`. Sie bieten zahlreiche Methoden (vorgestellt in Kapitel 2), die auf bestimmte Unterklassen von Ausdrücken wie die polynomialen Ausdrücke angewendet werden. Die Erkenntnis, dass ein Ausdruck zu dieser oder jener Klasse gehört, erlaubt zu wissen, welche Funktionen für die Anwendung darauf infrage kommen. Ein Problem, bei dem dieses Wissen wichtig ist, ist die Vereinfachung von Ausdrücken. Wegen dieser Aufgaben wurden die wichtigsten Klassen

der Ausdrücke im CAS definiert, und dieser Ansatz ist es, dem wir im Verlauf dieses Buches sehr oft den Vorzug geben.

Andererseits kann der Anwender einen Vorfahren *konstruieren*, der die Definitionsbereiche für die Rechnung festlegt. Das ist besonders interessant, wenn dieser Vorfahr in *Normalform* ist: das heißt, dass zwei Objektelemente genau dann mathematisch gleich sind, wenn sie die gleiche Darstellung haben.

Um es kurz zu sagen, die Flexibilität ist der Hauptvorteil der Ausdrücke: keine explizite Deklaration einer Definitionsmenge, die Einführung neuer Variablen und symbolischer Funktionen oder der Wechsel der Definitionsmenge während des Programmablaufs (wenn wir beispielsweise den Sinus eines polynomialen Ausdrucks nehmen), die Verwendung der gesamten Palette von Werkzeugen der Analysis (Integration usw.). Die Vorteile der expliziten Deklaration der Definitionsmenge bestehen in den didaktischen Möglichkeiten, einer oftmals größeren Strenge[5], die automatische Umwandlung in die Normalform (die auch ein Nachteil sein kann!), wie auch die Möglichkeit komplizierterer Konstruktionen, die mit Ausdrücken schwierig wären (Rechnungen auf einem endlichen Körper oder algebraische Erweiterungen von $\mathbb{Q}$ in einem nichtkommutativen Ring usw.).

[5] Sage ist kein *zertifiziertes* CAS

Teil II.

Algebra und symbolisches Rechnen

6. Endliche Körper und elementare Zahlentheorie

> Die natürlichen Zahlen hat der liebe Gott gemacht.
> Alles andere ist Menschenwerk.
> Leopold Kronecker (1823 - 1891)

Dieses Kapitel beschreibt die Verwendung von Sage in der elementaren Zahlentheorie zur Bearbeitung von Objekten auf Ringen oder endlichen Körpern (Abschnitt 6.1), zu Primaltests (Abschnitt 6.2) oder zur Faktorisierung einer ganzen Zahl (Abschnitt 6.3); schließlich diskutieren wir einige Anwendungen (Abschnitt 6.4).

6.1. Ringe und endliche Körper

Ringe und endliche Körper sind ein grundlegendes Objekt in der Zahlentheorie und ganz allgemein beim symbolischen Rechnen. Tatsächlich stammen zahlreiche Algorithmen des symbolischen Rechnens vom Rechnen auf endlichen Körpern, dann nutzt man die mit Techniken wie dem Aufstieg von Hensel oder der Rekonstruktion mit chinesischen Resten erhaltene Information. Erwähnen wir beispielsweise den Algorithmus von Cantor-Zassenhaus zur Faktorisierung univariater Polynome mit ganzzahligen Koeffizienten, der mit der Faktorisierung des gegebenen Polynoms auf einem endlichen Körper beginnt.

6.1.1. Ring der ganzen Zahlen modulo n

In Sage wird der Restklassenring $\mathbb{Z}/n\mathbb{Z}$ der ganzen Zahlen modulo n mi Hilfe des Konstruktors `IntegerModRing` (oder einfacher `Integers`) definiert. Alle mit diesem Konstruktor erzeugten Objekte und ihre Ableitungen werden systematisch modulo n reduziert und haben somit eine kanonische Form, das heißt, dass zwei Variable, die den gleichen Wert modulo n darstellen, auch die gleiche interne Darstellung besitzen. In bestimmten, sehr seltenen Fällen ist es effizient, die Reduktionen modulo n zu verzögern, beispielsweise wenn wir Matrizen mit solchen Koeffizienten multiplizieren; man wird dann vorziehen, mit ganzen Zahlen zu arbeiten und die Reduktion modulo n „von Hand“ via $a\%n$ vornehmen. Aber Achtung, der Modul n erscheint nicht explizit im ausgegebenen Wert:

```
sage: a = IntegerModRing(15)(3); b = IntegerModRing(17)(3); a, b
(3, 3)
sage: a == b
False
```

Eine Konsequenz ist, dass wenn wir ganze Zahlen modulo n kopieren, wir die Information über n verlieren. Ist eine Variable gegeben, die eine Zahl modulo n enthält, gewinnen wir die Information über n mittels der Methoden `base_ring` oder `parent` zurück und erhalten den Wert von n mit der Methode `characteristic`.

```
sage: R = a.base_ring(); R
Ring of integers modulo 15
sage: R.characteristic()
15
```

Die Grundrechenarten Addition, Subraktion, Multiplikation und Division sind für ganze Zahlen modulo n überladen und rufen die entsprechenden Funktionen auf, ebenso werden die ganzen Zahlen automatisch konvertiert, sobald einer der Operanden eine ganze Zahl modulo n ist:

```
sage: a + a, a - 17, a*a + 1, a^3
(6, 1, 10, 12)
```

Was die Inversion $1/a$ mod n oder die Division b/a mod n betrifft, so führt Sage sie aus, wenn das möglich ist, sonst wird ein `ZeroDivisionError` gemeldet, d.h. wenn a und b einen nichttrivialen ggT haben.

```
sage: 1/(a + 1)
4
sage: 1/a
Traceback (click to the left of this block for traceback)
...
ZeroDivisionError: Inverse does not exist.
```

Um den Wert von a zu erhalten - sofern er ganzzahlig ist - ausgehend vom Rest a mod n, können wir die Methode `lift` nutzen oder auch `ZZ`:

```
sage: z = lift(a); y = ZZ(a); print y, type(y), y == z
3 <type 'sage.rings.integer.Integer'> True
```

Die *additive Ordnung* von a modulo n ist die kleinste ganze Zahl $k > 0$, sodass $ka = 0$ mod n ist. Es gilt $k = n/g$ mit $g = \text{ggT}(a, n)$ und wird durch die Methode `additive_order` angegeben (man sieht nebenbei, dass man auch mit `Mod` oder `mod` arbeiten kann, um die ganzen Zahlen modulo n zu bekommen):

```
sage: [Mod(x,15).additive_order() for x in range(15)]
[1, 15, 15, 5, 15, 3, 5, 15, 15, 5, 3, 15, 5, 15, 15]
```

Die *multiplikative Ordnung* von a modulo n für a teilerfremd zu n ist die kleinste ganze Zahl $k > 0$, für die $a^k = 1$ ist. (Wenn a mit n einen gemeinsamen Teiler p hat, dann ist a^k mod n ein Vielfaches von p für jedes k.) Wenn diese multiplikative Ordnung gleich $\varphi(n)$ ist, nämlich die multiplikative Gruppe modulo n, sagen wir, dass a ein *Generator* dieser Gruppe ist. So gibt es für $n = 15$ keinen Generator, und die maximale Ordnung ist $4 < 8 = \varphi(15)$.

```
sage: [[x, Mod(x,15).multiplicative_order()]
....:     for x in range(1,15) if gcd(x,15) == 1]
[[1, 1], [2, 4], [4, 2], [7, 4], [8, 4], [11, 2], [13, 4], [14, 2]]
```

Hier ein Beispiel mit $n = p$ prim mit dem Generator 3:

```
sage: p = 10^20+ 39; mod(2,p).multiplicative_order()
50000000000000000019
sage: mod(3,p).multiplicative_order()
100000000000000000038
```

Eine wichtige Operation auf $\mathbb{Z}/n\mathbb{Z}$ ist die *modulare Potenz*, die darin besteht, a^e mod n zu berechnen. Das Kryptosystem RSA beruht auf dieser Operation. Zur effizienten Berechnung von a^e mod n benötigen die effizientesten Algorithmen Multiplikationen der Ordnung $\log e$ oder Quadrate modulo n. Es kommt darauf an, alle Rechnungen systematisch modulo n zu reduzieren, anstatt erst a^e als ganze Zahl zu berechnen, wie dies das folgende Beispiel zeigt:

```
sage: n = 3^100000; a = n-1; e = 100
sage: timeit('(a^e) % n')
5 loops, best of 3: 373 ms per loop
sage: timeit('power_mod(a,e,n)')
25 loops, best of 3: 6.56 ms per loop
```

6.1.2. Endliche Körper

Die endlichen Körper[1] werden mit dem Konstruktor `FiniteField` definiert (oder einfacher mit `GF`). Wir können auch den *Primzahlkörper* `GF(p)` mit der Primzahl p wie die zusammengesetzten Körper mit `GF(q)` erzeugen. Dabei ist $q = p^k$, p prim und $k > 1$ ganz. Wie bei den Ringen haben die auf einem solchen Körper generierten Objekte eine kanonische Form, daher erfolgt eine Reduktion bei jeder Operation. Die endlichen Körper haben die gleichen Eigenschaften wie die Ringe (Unterabschnitt 6.1.1) und bieten zusätzlich die Möglichkeit, ein von null verschiedenes Element zu invertieren:

```
sage: R = GF(17); [1/R(x) for x in range(1,17)]
[1, 9, 6, 13, 7, 3, 5, 15, 2, 12, 14, 10, 4, 11, 8, 16]
```

Ein Körper F_{p^k}, der kein Primzahlkörper ist, mit p prim und $k > 1$ ist zum Quotientenring der Polynome von $F_p[x]$ modulo eines unitären und irreduziblen Polynoms f vom Grad k isomorph. In diesem Fall verlangt Sage einen Namen für den *Generator* des Körpers, also die Variable x:

```
sage: R = GF(9,name='x'); R
Finite Field in x of size 3^2
```

Hier wählt Sage das Polynom f automatisch:

```
sage: R.polynomial()
x^2 + 2*x + 2
```

Die Elemente des Körpers werden nun durch die Polynome $a_{k-1}x^{k-1}+\ldots+a_1x+a_0$ dargestellt, wobei die a_i Elemente von $\mathbb{F}_p$ sind:

```
sage: Set([r for r in R])
{0, 1, 2, x, x + 1, x + 2, 2*x, 2*x + 1, 2*x + 2}
```

Wir können Sage auch das irreduzible Polynom f vorschreiben:

```
sage: Q.<x> = PolynomialRing(GF(3))
sage: R2 = GF(9, name='x', modulus=x^2+1); R2
Finite Field in x of size 3^2
```

[1] In der französischen (und in der deutschen) Literatur werden endliche Körper mit q Elementen üblicherweise mit $\mathbb{F}_q$ bezeichnet, während man in der englischsprachigen meistens GF(q) findet. Wir verwenden hier die französische (und deutsche) Schreibweise für die mathematischen Begriffe und die englische im Sage-Code.

Trotzdem ist Vorsicht geboten, denn wenn die beiden oben erzeugten Instanzen `R` und `R2` auch zu $\mathbb{F}_9$ isomorph sind, ist der Isomorphismus dennoch nicht nutzbar:

```
sage: p = R(x+1); R2(p)
Traceback (click to the left of this block for traceback)
...
TypeError: unable to coerce <class 'sage.interfaces.r.R'>
```

6.1.3. Rationale Rekonstruktion

Das Problem der rationalen Rekonstruktion ist eine hübsche Anwendung des Rechnens mit Resten. Ist ein Rest modulo m gegeben, geht es darum, eine „kleine“ rationale Zahl x/y zu finden, sodass $x/y \equiv a \mod m$. Wenn wir wissen, dass eine solche kleine rationale Zahl existiert, dann, anstatt x/y direkt zu berechnen, insofern es rational ist, berechnen wir x/y modulo m, was den Rest a ergibt, dann finden wir x/y wieder durch rationale Rekonstruktion. Dieser zweite Weg ist oft effektiver, denn wir ersetzen rationale Berechnungen - bei denen aufwendige ggT auftreten - durch modulare Berechnungen.

Lemma. Seien $a, m \in \mathbb{N}$, mit $0 < a < m$. Es existiert höchstens ein Paar teilerfremder ganzer Zahlen $x, y \in \mathbb{Z}$, sodass $x/y \equiv a \mod m$ mit $0 < |x|$, $y \leq \sqrt{m/2}$.

Ein solches Paar x, y existiert nicht immer, beispielsweise für $a = 2$ und $m = 5$. Der Algorithmus der rationalen Rekonstruktion basiert auf dem Algorithmus des erweiterten ggT. Der erweiterte ggT von m und a berechnet eine Folge ganzer Zahlen, $a_i = \alpha_i m + \beta_i a$, wobei die ganzen Zahlen a_i abnehmen, und die Absolutwerte der Koeffizienten $\alpha i, \beta i$ anwachsen. Daher genügt es aufzuhören, sobald $|\alpha i|, |\beta i| \leq \sqrt{m/2}$ ist, und die Lösung ist dann $x/y = \alpha_i/\beta_i$. Dieser Algorithmus steht in Sage in Gestalt der Funktion `rational_reconstruction` zu Verfügung. Sie gibt x/y zurück, falls eine solche Lösung existiert und einen Fehler, falls nicht:

```
sage: rational_reconstruction(411,1000)
-13/17
sage: rational_reconstruction(409,1000)
Traceback (click to the left of this block for traceback)
...
ArithmeticError: rational reconstruction of 409 (mod 1000) does not exist
```

Um die rationale Rekonstruktion zu veranschaulichen, betrachten wir die Berechnung der harmonischen Zahl $H_n = 1 + 1/2 + \ldots + 1/n$. Die naive Rechnung mit den rationalen Zahlen geht so:

```
sage: def harmonic(n):
....:     return add([1/x for x in range(1,n+1)])
```

Nun wissen wir, dass H_n in der Form p_n/q_n mit p_n, q_n ganz geschrieben werden kann, wobei q_n das kgV von $1, 2, \ldots, n$ ist. Weiter wissen wir, dass $H_n \leq \log n + 1$ ist, was p_n einzugrenzen erlaubt. Daraus leiten wir folgende Funktion ab, die H_n durch modulare Rechnung und rationale Rekonstruktion bestimmt:

```
sage: def harmonic_mod(n,m):
....:     return add([1/x % m for x in range(1,n+1)])
sage: def harmonic2(n):
....:     q = lcm(range(1,n+1))
```

```
....:     pmax = RR(q*(log(n)+1))
....:     m = ZZ(2*pmax^2)
....:     m = ceil(m/q)*q + 1
....:     a = harmonic_mod(n,m)
....:     return rational_reconstruction(a,m)
```

Die Zeile `m = ZZ(2*pmax^2)` garantiert, dass die rationale Rekonstruktion den Wert $p \leq \sqrt{m/2}$ findet, während die folgende Zeile sicherstellt, dass m prim ist mit $x = 1, 2, \ldots, n$, wenn $1/x \bmod n$ keinen Fehler auslöst.

```
sage: harmonic(100) == harmonic2(100)
True
```

In diesem Beispiel ist die Funktion `harmonic2` nicht effizienter als die Funktion `harmonic`, es verdeutlicht aber unsere Absicht. Es ist nicht immer erforderlich, eine strenge Grenze für x und y zu kennen, eine Schätzung „π mal Daumen“ reicht hin, wenn man u.a. einfach verifizieren will, dass x/y die gesuchte Lösung ist.

Wir können die rationale Rekonsruktion mit einem Zähler x und einem Nenner y unterschiedlicher Größe verallgemeinern (siehe Abschnitt 5.10 in [vzGG03]).

6.1.4. Chinesische Reste

Eine weitere nützliche Anwendung der modularen Rechnung ist diejenige, die gemeinhin „chinesische Reste“ heißt. Seien zwei teilerfremde Moduln m und n gegeben, eine ganzzahlige Unbekannte x, sodass $x \equiv m \bmod m$ und $x \equiv b \bmod n$. Dann erlaubt der *Chinesische Restsatz* die eindeutige Rekonstruktion des Wertes von x modulo dem Produkt mn. In der Tat leitet man aus $x \equiv a \bmod n$ her, dass x in der Form $x = a + \lambda m$ geschrieben wird mit $\lambda \in \mathbb{Z}$. Setzen wir diesen Wert in $x \equiv b \bmod n$ ein, bekommen wir $\lambda \equiv \lambda_0 \bmod n$ mit $\lambda_0 = (b-a)/m \bmod n$. Daraus resultiert $x = x_0 + \mu mn$, wobei $x_0 = a + \lambda_0 m$ ist und μ eine beliebige ganze Zahl.

Wir beschreiben hier die einfachste Variante der „chinesischen Reste“. Ebenso können wir den Fall mehrerer Moduln $m_1, m_2, \ldots, m_k$ betrachten. Der Befehl zum Auffinden von x_0 aus a, b, m, n ist in Sage `crt(a,b,m,n)`.

```
sage: a = 2; b = 3; m = 5; n= 7; lambda0 = (b-a)/m % n; a + \lambda0*m
17
sage: crt(a,b,m,n)
17
```

Kommen wir auf die Berechnung von H_n zurück. Zunächst berechnen wir $H_n \bmod m$ für $i = 1, 2, \ldots, k$. Dann leiten wir $H_n \bmod (m_1 \cdots m_k)$ mit den chinesischen Resten her und finden H_n schließlich durch rationale Rekonstruktion:

```
sage: def harmonic3(n):
....:     q = lcm(range(1,n+1))
....:     pmax = RR(q*(log(n)+1))
....:     B = ZZ(2*pmax^2)
....:     a = 0; m = 1; p = 2^63
....:     while m < B:
....:         p = next_prime(p)
```

```
....:         b = harmonic_mod(n,p)
....:         a = crt(a,b,m,p)
....:         m = m*p
....:     return rational_reconstruction(a,m)
sage: harmonic(100) == harmonic3(100)
True
```

Sages Funktion `crt` funktioniert auch dann, wenn die Moduln m und n nicht teilerfremd sind. Es sei $g = \gcd(m, n)$. Dann gibt es eine Lösung genau dann, wenn $a \bmod g \equiv b \bmod g$ gilt:

```
sage: crt(15,1,30,4)
45
sage: crt(15,2,30,4)
Traceback (click to the left of this block for traceback)
...
ValueError: No solution to crt problem since gcd(30,4) does not divide 15-2
```

In Übung 22 stellen wir eine komplexere Anwendung der chinesischen Reste vor.

6.2. Primzahltests

Die Prüfung, ob eine ganze Zahl eine Primzahl ist, stellt eine fundamentale Operation eines Programms für symbolische Rechnungen dar. Selbst wenn der Anwender nicht darauf achtet, werden solche Prüfungen vom Programm tausende Male pro Sekunde ausgeführt. Um beispielsweise ein Polynom aus $\mathbb{Z}[x]$ zu faktorisieren, beginnen wir damit, es in $\mathbb{F}_p[x]$ nach einer Primzahl p zu zerlegen und müssen daher eine solche Zahl p finden.

Die wichtigsten Befehle	
Ring der ganzen Zahlen modulo n	`IntegerModRing(n)`
endlicher Körper mit q Elementen	`GF(q)`
Test auf Pseudoprimalität	`is_pseudoprime(n)`
Test auf Primalität	`is_prime(n)`

Tab. 6.1 - Wiederholung.

Es existieren zwei Klassen von Primzahltests. Am effizientesten sind die *Pseudoprimzahltests*, die im allgemeinen auf dem kleinen Satz von Fermat basieren, welcher besagt, dass, wenn p prim ist, dann jede ganze Zahl $0 < a < p$ von der Ordnung ist, die $p-1$ in der multiplikativen Gruppe $(\mathbb{Z}/p\mathbb{Z})*$ teilt, und $a^{p-1} \equiv 1 \bmod p$ gilt. Generell verwenden wir einen kleinen Wert von a $(2, 3, \ldots)$, um die Berechnung von $a^{p-1} \bmod p$ zu beschleunigen. Ist $a^{p-1} \not\equiv 1 \bmod p$, ist p sicher keine Primzahl. Falls $a^{p-1} \equiv 1 \bmod p$ ist, können wir nichts folgern: wir sagen dann, p ist pseudoprim zur Basis a. Die Annahme ist, dass wenn eine ganze Zahl p pseudoprim ist zu mehreren Basen, sie dann große Chancen hat prim zu sein (siehe jedoch später). Allen Pseudoprimzahltest ist gemeinsam, dass wenn sie `False` ergeben, die Zahl mit Sicherheit zusammenngesetzt ist, und bei `True` nichts gefolgert werden kann.

Die zweite Klasse wird von den *echten Primzahltests* gebildet. Diese Test geben immer eine korrekte Antwort, können aber bei Zahlen, die zu vielen Basen pseudoprim sind, weniger effizient sein als die Pseudoprimzahltests, besonders aber dann, wenn sie echte Primzahlen sind. Etliche Programme führen nur Pseudoprimzahltests aus. So lässt der Name der entsprechenden Funktion (zum Beispiel `isprime`) den Anwender glauben, es handele sich um

einen (echten) Primzahltest. Sage bietet zwei verschiedene Funktionen, den Pseudoprimzahltest `is_pseudoprime` und den Primzahltest `i_prime`.

```
sage: p = previous_prime(2^400)
sage: timeit('is_pseudoprime(p)')
625 loops, best of 3: 603 µs per loop
sage: timeit('is_prime(p)')
5 loops, best of 3: 318 ms per loop
```

Wir sehen an diesem Beispiel, dass der Primzahltest sehr viel aufwendiger ist; wenn möglich, werden wir `is_pseudoprime` vorziehen.

Bestimmte Test-Algorithmen liefern ein *Zertifikat*, das unabhängig verifiziert werden kann, was oft effizienter ist als der Test selbst. Sage liefert in der aktuellen Version ein solches Zertifikat noch nicht, doch mit dem Satz von Pocklington können wir eines erzeugen:

Satz. Sei $n > 1$ eine ganze Zahl, sodass $n-1 = FR$ mit $F \geq \sqrt{n}$. Wenn für jeden Primfaktor von F ein a existiert, sodass $a^{p-1} \equiv 1 \bmod n$ gilt, und $a^{\frac{n-1}{p}} - 1$ teilerfremd zu n ist, dann ist n eine Primzahl.

Sei beispielsweise $n = 2^{31} - 1$. Die Zerlegung von $n - 1$ ist $2 \cdot 3^2 \cdot 7 \cdot 11 \cdot 31 \cdot 151 \cdot 331$. Wir können $F = 151 \cdot 331$ nehmen. $a = 3$ passt zu den beiden Faktoren $p = 151$ und $p = 331$. Nun reicht der Nachweis hin, dass 151 und 331 prim sind. Dieser Test macht intensiven Gebrauch von der modularen Potenzierung.

Carmichael-Zahlen

Die *Carmichael-Zahlen* sind zusammengesetzte ganze Zahlen, die zu allen Basen pseudoprim sind. Der kleine fermatsche Satz erlaubt nicht, sie von Primzahlen zu unterscheiden, wieviele Basen auch probiert werden. Die kleinste Carmichael-Zahl ist $561 = 3 \cdot 11 \cdot 17$. Eine Carmichael-Zahl hat mindestens drei Primfaktoren. In der Tat, wenn wir annehmen, $n = pq$ sei eine Carmichael-Zahl mit p, q prim und $p < q$, dann haben wir nach Definition der Carmichael-Zahlen für jedes a, das $(\mathbb{Z}/p\mathbb{Z})$ erzeugt, die Gleichung $a^{n-1} \equiv 1$ modulo n und dann auch modulo q, was impliziert, dass $n - 1$ ein Vielfaches von $q - 1$ ist, was mit $n = pq$ wegen $p < q$ aber nicht vereinbar ist. Haben wir jedoch $n = pqr$, dann genügt $a^{n-1} \equiv 1 \bmod p$ - ebenso mit q und r - und wir erhalten mit dem chinesischen Restsatz $a^{n-1} \equiv 1 \bmod n$. Eine hinreichende Bedingung ist, dass $n - 1$ ein Vielfaches von $p - 1$, $q - 1$ und $r - 1$ ist.

```
sage: [560 % (x-1) for x in [3, 11, 17]]
[0, 0, 0]
```

Übung 19. Schreiben Sie in Sage eine Funktion, die die Carmichael-Zahlen $n = pqr \leq N$ zählt mit verschiedenen Primzahlen p, q, r. Wieviele finden Sie für $N = 10^4, 10^5, 10^6, 10^7$? (Richard Pinch hat 20138200 Carmichael-Zahlen unterhalb 10^{21} gezählt.)

Um eine Operation auf einem Intervall von Primzahlen auszuführen, ist es besser, mit der Konstruktion `prime_range` zu arbeiten, die mit einem Sieb eine Primzahlliste liefert, statt eine Schleife mit `next_probable_prime` oder `next_prime` zu verwenden.

```
sage: def count_primes1(n):
....:     return add([1 for p in range(n+1) if is_prime(p)])
sage: timeit('count_primes(10^5)')
5 loops, best of 3: 556 ms per loop
```

Die Funktion ist mit `is_pseudoprime` schneller als mit `is_prime`:

```
sage: def count_primes2(n):
....:     return add([1 for p in range(n+1) if is_pseudoprime(p)])
sage: timeit('count_primes2(10^5)')
5 loops, best of 3: 204 ms per loop
```

In nächsten Beispiel erweist es sich als besser, eine Schleife zu verwenden, die es vermeidet, eine Liste mit 10^5 Elementen zu erzeugen, und auch hier ist `is_pseudoprime` schneller als `is_prime`:

```
sage: def count_prime3(n):
....:     s = 0; p = 2
....:     while p <= n: s += 1; p = next_prime(p)
....:     return s
sage: timeit('count_prime3(10^5)')
5 loops, best of 3: 67.4 ms per loop
sage: def count_prime4(n):
....:     s = 0; p = 2
....:     while p <= n: s += 1; p = next_probable_prime(p)
....:     return s
sage: timeit('count_prime4(10^5)')
5 loops, best of 3: 53.2 ms per loop
```

Noch schneller geht es mit `prime_range`:

```
sage: def count_prime5(n):
....:     s = 0
....:     for p in prime_range(n): s += 1
....:     return s
sage: timeit('count_prime5(10^5)')
25 loops, best of 3: 4.7 ms per loop
```

6.3. Faktorisierung und diskreter Logarithmus

Wir sagen, eine ganze Zahl a ist ein Quadrat - oder ein quadratischer Rest - modulo n, wenn ein x existiert mit $0 \leq x < n$, sodass $a \equiv x^2 \bmod n$ ist. Andernfalls sagen wir, a ist kein quadratischer Rest modulo n. Ist $n = p$ eine Primzahl, kann dieser Test dank der Berechnung des Jacobi-Symbols von a und p, geschrieben $(a|p)$, das die Werte `{-1,0,1}` annehmen kann, effizient entschieden werden, Dabei bedeutet $(a|p) = 0$, dass a ein Vielfaches von p ist, und $(a|p) = 1$ (bzw. $(a|p) = -1$), dass a ein quadratischer Rest modulo p ist (bzw. nicht ist). Die Komplexität der Berechnung des Jacobi-Symbols $(a|n)$ ist im wesentlichen die gleiche wie die der Berechnung des ggT von a und n, nämlich $\mathcal{O}(M(l)\log l)$, wobei l die Größe von n ist und $M(l)$ der Aufwand für das Produkt der beiden ganzen Zahlen der Größe l. Alle Implementierungen des Jacobi-Symbols - wie auch des ggT - haben etwa dieselbe Komplexität (`a.jacobi(n)` berechnet $(a|n)$):

```
sage: p = (2^42737+1)//3; a = 3^42737
sage: timeit('a.gcd(p)')
125 loops, best of 3: 3.6 ms per loop
sage: timeit('a.jacobi(p)')
125 loops, best of 3: 3.59 ms per loop
```

Wenn n zusammengesetzt ist, ist die Suche nach den Lösungen von $x^2 \equiv a \bmod n$ genauso schwierig wie die Zerlegung von n. In jedem Fall gibt das Jacobi-Symbol, das relativ leicht zu berechnen ist, eine Teil-Information. Ist nämlich $(a|n) = -1$, gibt es keine Lösung, denn für eine Lösung muss $(a|p) = 1$ für alle Primfaktoren p von n gelten.

Der diskrete Logarithmus. Seien n eine natürliche Zahl, g ein *Generator* der multiplikativen Gruppe modulo n und a teilerfremd zu n mit $0 < a < n$. Da g nach Voraussetzung ein Generator ist, gibt es eine ganze Zahl x, sodass $g^x = a \bmod n$. Das Problem des *diskreten Logarithmus* besteht darin, eine solche ganze Zahl x zu finden. Die Methode `log` erlaubt das Problem zu lösen:

```
sage: p = 10^10+19; a = mod(17,p); a.log(2)
6954104378
sage: mod(2,p)^6954104378;
17
```

Die besten bekannten Algorithmen zur Berechnung eines diskreten Logarithmus sind von gleicher Komplexitätsordnung - als Funktion der Größe von n - wie diejenigen zur Zerlegung von n. Die aktuelle Implementierung in Sage ist durchaus annehmbar:

```
sage: p = 10^20+39; a = mod(17,9)
sage: time r = a.log(3)
Time: CPU 0.54 s, Wall: 1.00 s
```

Aliquot-Folgen (Inhaltsketten)

Die einer natürlichen Zahl n zugeordnete *Aliquot-Folge* ist die rekursiv definierte Folge (s_k) mit $s_0 = n$ und $s_{k+1} = \sigma(s_k) - s_k$, wobei $\sigma(k)$ die Summe der Teiler von s_k ist, d.h. s_{k+1} ist die Summe der *echten* Teiler von s_k, also ohne s_k selbst. Wir halten die Iteration an, sobald $s_k = 1$ ist - dann ist s_{k-1} prim - oder wenn die Folge (s_k) einen Zyklus beschreibt. Ausgehend von $n = 30$ erhalten wir beispielsweise:

$$30, 42, 54, 66, 78, 90, 144, 259, 45, 33, 15, 9, 4, 3, 1.$$

Hat der Zyklus die Länge eins, nennen wir die entsprechende Zahl *vollkommen*, so sind zum Beispiel $6 = 1+2+3$ und $28 = 1+2+4+7+14$ vollkommene Zahlen. Wenn der Zyklus die Länge zwei hat, nennen wir die beiden Zahlen *befreundet*, wie 220 und 284. Hat der Zyklus die Länge drei oder mehr, heißen die den Zyklus bildenden Zahlen *gesellig*.

Übung 20. Berechnen Sie die mit 840 beginnende Inhaltskette, geben Sie die 5 Primzahlen und die letzten 5 Elemente aus, und zeichnen Sie den Graphen von $\log_{10} s_k$ als Funktion von k (Sie können mit der Funktion `sigma` arbeiten).

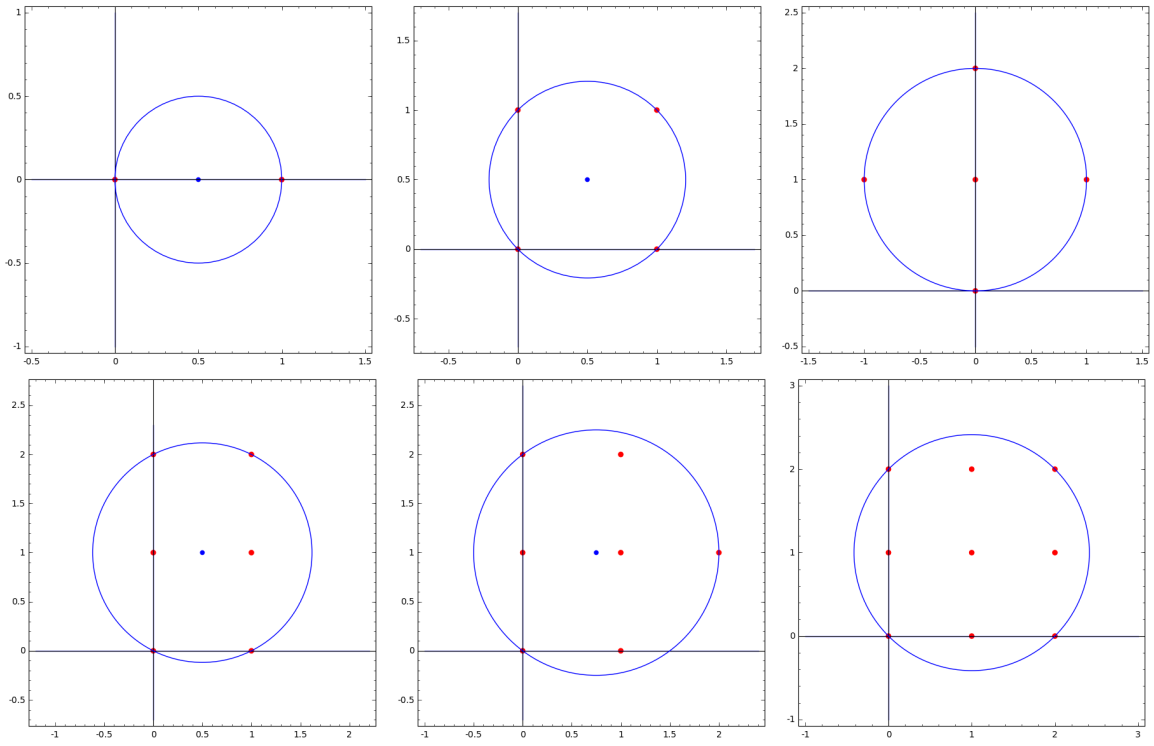
6.4. Anwendungen

6.4.1. Die Konstante δ

Die Konstante δ ist eine zweidimensionale Verallgemeinerung der Konstanten γ von Euler. Sie ist definiert wie folgt:

$$\delta = \lim_{n\to\infty} \left(\sum_{k=2}^{n} \frac{1}{\pi r_k^2} - \log n \right), \tag{6.1}$$

wobei r_k der Radius der kleinsten Scheibe der affinen Ebene $\mathbb{R}^2$ ist, die mindestens k Punkte aus $\mathbb{Z}^2$ enthält, zum Beispiel $r_2 = 1/2$, $r_3 = r_4 = \sqrt{2}/2$, $r_5 = 1$, $r_6 = \sqrt{5}/2$, $r_7 = 5/4$, $r8 = r9 = \sqrt{2}$:



Übung 21 *(Konstante von Masser-Gramain)*

1. Schreiben Sie eine Funktion, die als Eingabe eine natürliche Zahl k erhält und den Radius r_k und den Mittelpunkt (x_k, y_k) einer kleinsten Scheibe zurückgibt, die mindestens k Punkte aus $\mathbb{Z}^2$ enthält. Wir nehmen $r_k < \sqrt{k/\pi}$ an.

2. Schreiben Sie eine Funktion, die den Kreis mit dem Zentrum (x_k, y_k) und dem Radius r_k mit $m \geq k$ Punkten in $\mathbb{Z}^2$ zeichnet - wie oben zu sehen.

3. Berechnen Sie im Rahmen von

$$\frac{\sqrt{\pi(k-6)+2} - \sqrt{2}}{\pi} < r_k < \sqrt{\frac{k-1}{\pi}} \tag{6.2}$$

eine Näherung von δ mit einer Fehlerschranke von 0.3.

6.4.2. Berechnung mehrfacher Integrale durch rationale Rekonstruktion

Diese Anwendung ist durch den Artikel [Bea09] inspiriert. Seien k und $n_1, n_2, \ldots, n_k$ natürliche Zahlen oder null. Wir wollen das Integral

$$\int_V x_1^{n_1} x_2^{n_2} \cdots x_k^{n_k} dx_1 dx_2 \cdots dx_k$$

berechnen, wobei der Integrationsbereich durch $V = \{x_1 \geq x_2 \geq \ldots \geq x_k \geq 0, x_1 + \ldots + x_x \leq 1\}$ definiert ist. Wir erhalten beispielsweise für $k = 2$, $n_1 = 3$, $n_2 = 5$ den Wert

$$I = \int_{x_2=0}^{1/2} \int_{x_1=x_2}^{1-x_2} x_1^3 x_2^5 dx_1 dx_2 = \frac{13}{258048}.$$

ÜBUNG 22. Unter der Annahme, dass I rational ist, soll ein Algorithmus erstellt werden, der mit rationaler Rekonstruktion und/oder dem chinesischen Restsatz arbeitet, um I zu berechnen. Man implementiere diesen Algorithmus in Sage und wende ihn an auf den Fall

$$[n_1, \ldots, n_{31}] = [9, 7, 8, 11, 6, 3, 7, 6, 6, 4, 3, 4, 1, 2, 2, 1, 1, 1, 2, 0, 0, 0, 3, 0, 0, 0, 0, 1, 0, 0, 0].$$

7. Polynome

Dieses Kapitel ist den Polynomen mit einer Unbestimmten und verwandten Objekten gewidmet, im wesentlichen den gebrochen rationalen Ausdrücken und den formalen Potenzreihen. Zu Anfang werden wir sehen, wie mit Sage zum Beispiel die euklidsche Polynomdivision ausgeführt wird, die Zerlegung in irreduzible Polynome, das Ziehen von Wurzeln oder die Zerlegung von gebrochen rationalen Ausdrücken in einfache Elemente. Dies erfolgt unter Beachtung des Ringes oder des Körpers, zu dem die Koeffizienten der betrachteten Polynome gehören: Sage erlaubt uns das Rechnen auf den Polynomringen $A[x]$, deren Quotienten $A[x]/ < (P(x) >$, den Körpern der gebrochen rationalen Terme $K(x)$ oder auch den Ringen der formalen Potenzreihen $A[[x]]$ für eine ganze Palette von Basisringen.

Die Operationen auf den Polynomen haben aber auch überraschende Anwendungen. Wie errät man automatisch den nächsten Term der Folge

$$1, 1, 2, 3, 8, 11, 39, \ldots ?$$

Beispielsweise mit der Padé-Approximation von gebrochen rationalen Funktionen, was in Unterabschnitt 7.4.3 gezeigt wird! Wie werden die Lösungen der Gleichung $e^{xf(x)} = f(x)$ mühelos in eine Reihe entwickelt? Die Antwort findet man im Unterabschnitt 7.5.3.

Generell stellen wir uns vor, dass der Leser mit rationalen und gebrochen rationalen Ausdrücke auf dem Niveau einer universitären Veranstaltung für das erste Studienjahr umgehen kann. Darüber hinaus bringen wir auch einige anspruchsvollere Themen zur Sprache. (Wie beweist man, dass die Lösungen der Gleichung $x^5 - x - 1 = 0$ sich nicht mit Wurzeln ausdrücken lassen? Es reicht hin, seine Galois-Gruppe zu berechnen wie in Unterabschnitt 7.3.4 erklärt.) Die entsprechenden Passagen werden später im Buch nicht gebraucht, und der Leser mag sie schadlos überspringen. Schließlich enthält das Kapitel etliche Beispiele für die Manipulation algebraischer und p-adischer Zahlen.

Die Polynome mit mehreren Unbestimmten sind ihrerseits Gegenstand des 9. Kapitels.

7.1. Polynomringe

7.1.1. Einführung

In Kapitel 2 haben wir gesehen. wie Rechnungen mit *symbolischen Ausdrücken*, den Elementen des symbolischen Ringes `SR` ausgeführt werden. Einige der auf diese Ausdrücke anwendbaren Methoden, zum Beispiel `degree` gelten für Polynome:

```
sage: x=var('x'); p = (2*x+1)*(x+2)*(x^4-1)
sage: print(p, 'ist vom Grad', p.degree(x))
(x+2)*(2*x+1)*(x^4-1) ist vom Grad 6
```

In manchen CAS wie Maple oder Maxima ist die Darstellung der Polynome als symbolische Ausdrücke die normale Art für ihre Bearbeitung. Nach dem Vorbild von Axiom, Magma oder

MuPAD[1] erlaubt Sage auch, Polynome mehr algebraisch zu behandeln und kann auf Ringen wie $\mathbb{Q}[x]$ oder $\mathbb{Z}/4\mathbb{Z}[x, y, z]$ rechnen.

So weisen wir der Python-Variablen `x` die *Unbestimmte des Polynomringes in x mit rationalen Koeffizienten* zu, die durch `polygen(QQ, 'x')` statt der durch `var('x')` zurückgegebenen[2] *symbolischen Variablen x* zu, um das vorstehende Beispiel dadurch zu reproduzieren, dass wir auf einem wohldefinierten Polynomring arbeiten.

```
sage: x = polygen(QQ, 'x'); p = (2*x+1)*(x+2)*(x^4-1)
sage: print p, 'ist vom Grad', p.degree()
2*x^6 + 5*x^5 + 2*x^4 - 2*x^2 - 5*x - 2 ist vom Grad 6
```

Zu beachten ist, dass das Polynom automatisch entwickelt wird. Die „algebraischen“ Polynome werden immer in Normalform dargestellt. Das ist ein wesentlicher Unterschied im Vergleich zu Polynomem aus `SR`. Insbesondere ist die Darstellung zweier mathematisch gleicher Polynome im Rechner die gleiche, und ein Koeffizientenvergleich genügt für die Prüfung auf Gleichheit.

Die Möglichkeiten des Rechnens mit algebraischen Polynomen sind vielfältiger als die mit polynomialen Ausdrücken.

7.1.2. Erzeugung von Polynomringen

In Sage haben Polynome wie viele andere algebraische Objekte im allgemeinen Koeffizienten aus einem kommutativen Ring. Diesen Standpunkt nehmen wir ein, doch die meisten unserer Beispiele betreffen Polynome auf Körpern. Im ganzen Kapitel bezeichnen die Buchstaben A und K einen kommutativen Ring bzw. einen beliebigen Körper.

Die erste Etappe zur Durchführung einer Rechnung in einer algebraischen Struktur R ist oft die Erzeugung von R selbst. Wir erzeugen $\mathbb{Q}[x]$ mit

```
sage: R = PolynomialRing(QQ, 'x')
sage: x = R.gen()
```

Manipulation von Polynomringen $R = A[x]$	
Erzeugung (voll besetzt)	`R.<x> = A[]` oder `R.<x> = PolynomialRing(A, 'x')`
$\mathbb{Z}[x]$, $\mathbb{Q}[x]$, $\mathbb{R}[x]$, $\mathbb{Z}/n\mathbb{Z}[x]$	`ZZ['x']`, `QQ['x']`, `RR['x']`, `Integers(n)['x']`
Erzeugung (dünn besetzt)	`R.<x> = PolynomialRing(A, 'x', sparse=True)`
Zugriff auf den Basisring A	`R.base_ring()`
Zugriff auf die Variable x	`R.gen()` oder `R.0`
Prüfungen (ganzzahlig, noethersch, ...)	`R.is_integral_domain()`, `R.is_noetherian()`, ...

Tab. 7.1 - Polynomringe

Das `'x'` in der ersten Zeile ist eine Zeichenkette, der Name der Unbestimmten oder der *Generator* des Ringes. Das `x` der zweiten Zeile ist eine Python-Variable, in der wir den Generator wiederfinden; die Verwendung des gleichen Namens verbessert die Lesbarkeit des Codes. Das so in der Variablen `x` gespeicherte Objekt repräsentiert das Polynom $x \in \mathbb{Q}[x]$. Als Vorfahr (in Sage ist der *Vorfahr* eines Objektes die algebraische Struktur „aus der es stammt“, siehe Abschnitt 5.1) hat es den Ring `QQ['x']`:

[1] MuPAD ist inzwischen an Mathworks Inc. („MATLAB“) verkauft worden.

[2] Ein kleiner Unterschied: wenn `var('x')` die gleiche Wirkung hat wie `x = var('x')` bei der interaktiven Verwendung, dann ändert `polygen(QQ, 'x')` ohne Zuweisung den Wert der Python-Variablen `x` nicht.

```
sage: x.parent()
Univariate Polynomial Ring in x over Rational Field
```

Polynome mit polynomialen Koeffizienten

In Sage können wir Ringe von Polynomen definieren, deren Koeffizienten wiederum Polynome aus einem beliebigen kommutativen Ring sind - einschließlich einem anderen Polynomring. Doch Vorsicht, Ringe des nach diesem Verfahren gebildeten Typs $A[x][y]$ sind von den echten Polynomringen mit mehreren Variablen wie $A[x, y]$ verschieden. Die letzteren, die in Kapitel 9 vorgestellt werden, sind für die gängigen Rechnungen besser geeignet. Tatsächlich gleicht die Arbeit mit $A[x][y] \ldots$ oft einem Spiel mit Unbestimmten sehr unsymmetrischen Verhaltens.
Dennoch möchten wir gelegentlich eine Variable bevorzugen und die anderen als Parameter behandeln. Die Methode `polynomial` der multivariaten Polynome erlaubt die Isolation einer Variablen, etwa wie die Methode `collect` der Ausdrücke. Hier als Beispiel die Berechnung des reziproken Polynoms eines hinsichtlich einer seiner Unbestimmten gegebenen Polynoms:

```
sage: R.<x,y,z,t> = QQ[]; p = (x+y+z*t)^2
sage: p.polynomial(t).reverse()
(x^2 + 2*x*y + y^2)*t^2 + (2*x*z + 2*y*z)*t + z^2
```

Hier erzeugt `p.polynomial(t)` ein Polynom mit der einzigen Unbestimmten `t` und Koeffizienten aus `QQ[x,y,z]`, auf welches wir dann die Methode `reverse` anwenden. Die übrigen Konversionen zwischen $A[x, y, \ldots]$ und $A[x][y] \ldots$ funktionieren wie erwartet:

```
sage: x = polygen(QQ); y = polygen(QQ[x], 'y')
sage: p = x^3 + x*y + y + y^2; p
y^2 + (x + 1)*y + x^3
sage: q = QQ['x,y'](p); q
x^3 + x*y + y^2 + y
sage: QQ['x']['y'](q)
y^2 + (x + 1)*y + x^3
```

Das Polynom $x \in \mathbb{Q}[x]$ wird zugleich als verschieden angesehen von den Polynomen der Identität $x \in A[x]$, vom Basisring $A \neq \mathbb{Q}$ und von denen, deren Unbestimmte einen anderen Namen hat wie $t \in \mathbb{Q}[t]$.

```
sage: x = PolynomialRing(QQ, 'x').gen()
```

Bei dieser Gelegenheit sei darauf hingewiesen, dass wir aus mehreren Darstellungsweisen im Speicher wählen können, wenn wir einen Polynomring erzeugen. Die Unterschiede zwischen den Darstellungsweisen werden in Abschnitt 7.6 diskutiert werden.

Übung 23 (Variable und Unbestimmte).

1. Wie sind `x` und `y` zu definieren, um folgende Ergebnisse zu beobachten?

```
sage: x^2 + 1
y^2 + 1
sage: (y^2 + 1).parent()
Univariate Polynomial Ring in x over Rational Field
```

2. Welchen Wert besitzt `p` nach den Anweisungen

```
sage: Q.<x> = QQ[]; p = x + 1; x = 2; p = p + x
```

7.1.3. Polynome

Erzeugung und einfache Arithmetik. Nach der Anweisung `R.<x> = QQ[]` sind die mit `x` und rationalen Konstanten und mit $+$ und $*$ gebildeten Ausdrücke Elemente von $\mathbb{Q}[x]$. Beispielsweise legt Sage bei `p = x + 2` automatisch fest, dass der Wert der Variablen `x` und die Zahl 2 beide als Elemente von $\mathbb{Q}[x]$ interpretiert werden können. Daher wird die Additionsroutine von $\mathbb{Q}[x]$ aufgerufen; sie liefert das Polynom $x + 2 \in \mathbb{Q}[x]$.

Eine andere Möglichkeit der Erzeugung eines Polynoms besteht aus der Auflistung seiner Koeffizienten:

```
sage: def turm_polynomial(n, var='x'):
....:     return ZZ[var]([binomial(n, k)^2*factorial(k) for k in (0..n)])
```

Diese Funktion erzeugt Polynome, bei denen der Koeffizient von x^k als Anzahl der Möglichkeiten interpretiert wird, k Türme auf einem Schachbrett so aufzustellen, dass sie sich nicht bedrohen - daher der Name. Die Klammern nach `ZZ[var]` dienen der Umwandlung eines gegebenen Objektes in ein Element dieses Ringes. Die Konversion einer Liste $[a_0, a_1, \ldots]$ in ein Element von `ZZ['x']` gibt das Polynom $a_0 + a_1 x + \ldots \in \mathbb{Z}[x]$ zurück.

Zugriff auf Daten, syntaktische Operationen	
Unbestimmte x	`p.variables(), p.variable_name()`
Koeffizient von x^k	`p[k]`
führender Koeffizient	`p.leading_coefficient()`
Grad	`p.degree()`
Liste der Koeffizienten	`p.coeffs()`
Liste der von null verschiedenen Koeffizienten	`p.coefficients()`
Diktionär Grad $\mapsto$ Koeffizient	`p.dict()`
Prüfungen (unitär, konstant, ...)	`p.is_monic(), p,is_constant, ...`
einfache Arithmetik	
Operationen $p+q$, $p-q$, $p \times q$, p^k	`p + q, p - q, p*q, p^k`
Substitution $x := a$	`p(a)` oder `p.subs(a)`
Ableitung	`p.derivative()` oder `diff(p)`
Transformationen	
Transformation der Koeffizienten	`p.map_coefficients(f)`
Wechsel des Basisrings $A[x] \mapsto B[x]$	`p.change_ring(B)` oder `B['x'](p)`
reziprokes Polynom	`p.reverse()`

Tab. 7.2 - einfache Operationen mit den Polynomen $p, q \in A[x]$.

Übersicht über die Menge der Operationen auf Polynomen. Die Elemente eines Polynomringes werden als Python-Objekte der Klasse `Polynomial` oder abgeleiteter Klassen dargestellt. Die wichtigsten verfügbaren Operationen[3] auf diesen Objekten sind in den Tabellen

[3] Davon gibt es noch viel mehr. Die Tabellen führen keine zu sehr zugespitzte Funktionalitäten auf, keine stärker spezialisierten Varianten der erwähnten Methoden und auch nicht viele der Methoden, die allen „Elementen von Ringen“ gemeinsam sind, auch keine Sage-Objekte, die für Polynome nicht von besonderem Interesse sind. Wir weisen jedoch darauf hin, dass die speziellen Methoden (zum Beispiel `p.rescale(a)`, das zu `p(a*x)` äquivalent ist), oftmals effizienter sind als die mehr generellen Methoden, die sie ersetzen können.

7.2 bis 7.5 zusammengestellt. So findet man den Grad eines Polynoms durch Aufruf seiner Methode `degree`. Ebenso ergibt `p.subs(a)` oder einfach `p(a)` den Wert von p im Punkt a, dient aber auch zur Berechnung der Komposition $p \circ a$, wenn a selbst ein Polynom ist, und, noch allgemeiner, der Auswertung eines Polynoms aus $A[x]$ in einem Element einer A-Algebra:

```
sage: R.<x> = QQ[]
sage: p = R.random_element(degree=4)  # ein zufälliges Polynom
sage: p
1/2*x^4 - x^3 + 1/4*x^2 - 5/4*x - 1/2
sage: p.subs(x^2)
1/2*x^8 - x^6 + 1/4*x^4 - 5/4*x^2 - 1/2 sage: p.subs(matrix([[1,2],[3,4]]))
[125/2    91]
[273/2   199]
```

Auf den Inhalt der beiden letzten Tabellen kommen wir in Unterabschnitt 7.2.1 und Abschnitt 7.3 zurück.

Wechsel des Ringes. Die exakte Liste der verfügbaren Operationen, ihr Effekt und ihre Effizienz hängen stark vom Basisring ab. Beispielsweise besitzen die Polynome aus `ZZ['x']` eine Methode `content`, die ihren Inhalt zurückgibt, d.h. den ggT ihrer Koeffizienten; die von `QQ['x']` aber nicht, weil die Operation trivial ist. Die Methode `factor` existiert für alle Polynome, wirft aber eine Exzeption `NotImplementedError` bei einem Polynom mit Koeffizienten aus `SR` oder aus $\mathbb{Z}/4\mathbb{Z}$. Diese Exzeption bedeutet, dass diese Operation für diesen Objekttyp in Sage nicht zur Verfügung steht, obwohl sie einen mathematischen Sinn hat.

Daher ist es sehr angenehm, mit den verschiedenen Ringen von Koeffizienten jonglieren zu können, auf denen wir „dasselbe“ Polynom betrachten können. Angewendet auf ein Polynom aus $A[x]$ gibt die Methode `change_ring` sein Bild in $B[x]$ zurück, wenn eine Methode zur Konversion der Koeffizienten existiert. Oft geschieht die Konversion durch einen kanonischen Morphismus von A nach B: insbesondere dient `change_ring` zur Erweiterung des Basisrings, um zusätzliche algebraische Eigenschaften anzubringen. In diesem Beispiel ist das Polynom irreduzibel auf den ganzen Zahlen, wird aber auf $\mathbb{R}$ faktorisiert:

```
sage: x = polygen(QQ)
sage: p = x^2 - 16*x + 3
sage: p.factor()
x^2 - 16*x + 3
sage: p.change_ring(RDF).factor()
(x - 15.810249675906654) * (x - 0.18975032409334563)
```

RDF ist die Menge der „Maschinenzahlen“, die in Kapitel 11 vorgestellt werden. Die erhaltene Zerlegung ist nur eine Näherung, mit ihr kann das Ausgangspolynom nicht exakt rekonstruiert werden. Zu einer solchen Darstellung der reellen Wurzeln von Polynomen mit ganzzahligen Koeffizienten, die exakte Rechnungen erlaubt, verwenden wir die Menge `AA` der algebraischen reellen Zahlen. In den folgenden Abschnitten werden wir einige Beispiel sehen.

Die Methode `change_ring` ermöglicht auch die Reduktion eines Polynoms aus $\mathbb{Z}[x]$ modulo einer Primzahl:

```
sage: p.change_ring(GF(3))
x^2 + 2*x
```

Wenn umgekehrt $B \subset A$ ist und die Koeffizienten von p in B liegen, dann ist es wieder `change_ring`, womit wir p nach $B[x]$ zurückholen.

Iteration. Zuweilen müssen wir eine Transformation auf alle Koeffizienten eines Polynoms anwenden. Dazu ist die Methode `map_coefficients` da. Wird sie auf ein Polynom $A[x]$ mit einer Funktion f als Parameter angewendet, gibt sie das eingegebene Polynom zurück, nachdem die Funktion f auf jeden von null verschiedenen Koeffizienten von p angewendet worden ist. Meistens ist f eine mit der Konstrultion `lambda` eingeführte anonyme Funktion (siehe Unterabschnitt 3.3.2). Hier ein Beispiel, wie die Konjugierte eines Polynoms mit komplexen Koeffizienten zu berechnen ist:

```
sage: QQi.<myI> = QQ[I]  # myI ist das i von QQi, I dasjenige von SR
sage: R.<x> = QQi[]; p = (x + 2*myI)^3; p
x^3 + 6*I*x^2 - 12*x - 8*I
sage: p.map_coefficients(lambda z: z.conjugate())
x^3 - 6*I*x^2 - 12*x + 8*I
```

Im vorliegenden Fall können wir auch schreiben `p.map_coefficients(conjugate)`, denn für $z \in \mathbb{Q}$ hat `conjugate(z)` die gleiche Wirkung wie `z.conjugate`. Der explizite Aufruf einer Methodes des Objektes `z` ist sicherer: der Code funktioniert so mit allen einer Methode `conjugate()` übergebenen Objekten und nur dort.

Teilbarkeit und euklidsche Division	
Prüfung auf Teilbarkeit $p \mid q$	`p.divides(q)`
Multiplizität eines Teilers $q^k \mid p$	`k = p.valuation(q)`
euklidsche Division $p = qd + r$	`q, r = p.quo_rem(d)` oder `q = p//d, r = p%d`
Pseudodivision $a^k p = qd + r$	`q, r, k = p.pseudo_divrem(d)`
größter gemeinsamer Teiler	`p.gcd(q)` oder `gcd([p1, p2, p3])`
kleinstes gemeinsames Vielfaches	`p.lcm(q)` oder `lcm([p1, p2, p3])`
erweiterter ggT $g = up + vq$	`g, u, v = p.xgcd(q)` oder `xgcd(p,q)`
„chinesische Reste“ $c \equiv a \bmod p$ $c \equiv p \bmod q$	`c = crt(a, b, p, q)`
Verschiedenes	
Interpolation $p(x_i) = y_i$	`p = R.lagrange_polynomial([(x1,y1), ...])`
Inhalt von $p \in \mathbb{Z}[x]$	`p.content()`

Tab. 7.3 - Arithmetik der Polynome

7.2. Euklidsche Arithmetik

Nach Summe und Produkt sind hier die elementarsten Operationen die euklidsche Division und die Berechnung des größten gemeinsamen Teilers (ggT). Die entsprechenden Operatoren und Methoden (Tab. 7.3) ähneln denjenigen der ganzen Zahlen. Doch oft sind diese Operationen unter einer Decke zusätzlicher mathematischer Abstraktion verborgen: Quotienten von Ringen (Unterabschnitt 7.2.2), wo jede arithmetische Operation eine implizite euklidsche Division enthält, oder gebrochen rationale Ausdrücke, zu deren Normierung der ggT berechnet wird usw.

7.2.1. Teilbarkeit

Die euklidsche Division funktioniert auf einem Körper, und allgemeiner auf einem kommutativen Ring. Wenn der führende Koeffizient des Divisors invertierbar ist, dann ist dieser Koeffizient das einzige Element des Basisringes, durch den bei der Berechnung zu dividieren ist:

Operationen auf Polynomringen

Die Vorfahren polynomialer Objekte, die Ringe $A[x]$, sind ihrerseits vollwertige Sage-Objekte. Schauen wir kurz, wozu sie dienen können.
Eine erste Familie von Methoden erlaubt die Bildung von interessanten Polynomen, ihre zufällige Auswahl oder die Auflistung ganzer Familien von Polynomen; dies hier sind alle Polynome zweiten Grades auf $\mathbb{F}_2$:

```
sage: list(GF(2)['x'].polynomials(of_degree=2))
[x^2, x^2 + 1, x^2 + x, x^2 + x + 1]
```

Einige dieser Methoden werden wir in den folgenden Beispielen aufrufen, um Objekte zu erzeugen, mit denen wir arbeiten. Das 15. Kapitel erklärt in größerer Allgemeinheit, wie Elemente endlicher Mengen in Sage aufgezählt werden.
Zweitens „kennt“ das System einige Fakten zu jedem Polynomring. Wir können testen, ob ein gegebenes Objekt ein Ring ist, ob es noethersch ist:

```
sage: A = QQ['x']
sage: A.is_ring() and A.is_noetherian()
True
```

oder auch, ob $\mathbb{Z}$ ein Unterring von $\mathbb{Q}[x]$ ist, und für welche Werte von n der Ring $\mathbb{Z}/n\mathbb{Z}$ integral ist.

```
sage: ZZ.is_subring(A)
True
sage: [n for n in range(20)
....:    if Integers(n)['x'].is_integral_domain()]
[0, 2, 3, 5, 7, 11, 13, 17, 19]
```

Diese Möglichkeiten beruhen weitgehend auf dem System von *Kategorien* von Sage (siehe auch Unterabschnitt 5.2.3). Die Polynomringe sind Glieder einer bestimmten Zahl von Kategorien, wie der Kategorie der Mengen, der Kategorie der euklidschen Ringe und einiger anderer:

```
sage: A.categories()
[Join of Category of euclidean domains and Category of commutative
...
Category of sets with partial maps, Category of objects]
```

Das spiegelt wider, dass jeder Polynomring auch eine Menge ist, ein euklidscher Ring und so fort. Das System kann die allgemeinen Eigenschaften der Objekte dieser verschiedenen Kategorien auf die Polynomringe automatisch übertragen.

```
sage: R.<t> = Integers(42)[]; (t^20-1) % (t^5+8*t+7)
22*t^4 + 14*t^3 + 14*t + 6
```

Da der führende Koeffizient nicht invertierbar ist, können wir eine *euklidsche Pseudodivision* definieren: seien A ein kommutativer Ring, $p, d \in A[x]$ und a der führende Koeffizient von

d. Dann existieren zwei Polynome $q, r \in A[x]$ mit $\deg(r) < \deg(d)$ und eine ganze Zahl $k \leq \deg(p) - \deg(d) + 1$, sodass gilt

$$a^k p = qd + r.$$

Die euklidsche Pseudodivision ist gegeben durch `pseudo_divrem`.

Um eine exakte Division auszuführen, verwendet man ebenfalls den euklidschen Divisionsoperator //. Tatsächlich gibt die Division durch ein nicht konstantes Polynom mit / ein Resultat gebrochen-rationalen Typs zurück, was scheitert, weil es keinen Sinn ergibt:

```
sage: ((t^2+t)//t).parent()
Univariate Polynomial Ring in t over Ring of integers modulo 42
sage: (t^2+t)/t
Traceback (click to the left of this block for traceback)
...
TypeError: self must be an integral domain.
```

Übung 24. Normalerweise werden Polynome aus $\mathbb{Q}[x]$ in Sage mit der monomialen Basis $(x^n)_{n \in N}$ dargestellt. Die durch $T_n(\cos\theta) = \cos(n\theta)$ definierten Tschebyschow-Polynome bilden eine Familie von orthogonalen Polynomen und somit eine Basis von $\mathbb{Q}[x]$. Die ersten Tschebyschow-Polynome sind

```
sage: x = polygen(QQ); [chebyshev_T(n, x) for n in (0..4)]
[1, x, 2*x^2 - 1, 4*x^3 - 3*x, 8*x^4 - 8*x^2 + 1]
```

Schreiben Sie eine Funktion, der als Argument ein Element aus $\mathbb{Q}[x]$ übergeben wird und welche die Koeffizienten der Zerlegung zur Basis $(T_n)_{n \in \mathbb{N}}$ zurückgibt.

Übung 25. (Division nach wachsenden Potenzen). Seien $n \in \mathbb{N}$ und $u, v \in A[x]$ mit invertierbarem $v[0]$. Dann existiert eindeutig ein Paar (q, r) von Polynomen aus $A[x]$ mit $\deg(q) \leq n$, sodass gilt $u = qv + x^{n+1}r$. Schreiben Sie eine Funktion, die q und r mit einem Analogon zum Algorithmus der euklidschen Division berechnet. Wie ist diese Rechnung mit existierenden Funktionen am einfachsten zu machen?

ggT. Sage kann den ggT von Polynomen auf einem Körper dank der euklidschen Struktur von $K[x]$ berechnen, aber auch auf bestimmten anderen Ringen wie den ganzen Zahlen:

```
sage: S.<x> = ZZ[]; p = 2*(x^10-1)*(x^8-1)
sage: p.gcd(p.derivative())
2*x^2 - 2
```

Wir können den mehr symmetrischen Ausdruck `gcd(p,q)` vorziehen, der das gleiche Resultat liefert wie `p.gcd(q)`. Er ist in Sage allerdings nicht ganz so natürlich, denn das ist kein allgemeiner Mechanismus: die von `gcd(p,q)` aufgerufene Routine ist eine Funktion mit zwei Argumenten, die im Quellcode von Sage manuell definiert ist und ihrerseits `p.gcd` aufruft. Nur ein paar der üblichen Methoden besitzen eine solche zugeordnete Funktion. Der *erweiterte ggT* (engl. *extended greatest common divisor*), d.h. die Berechnung einer Bézout-Identität

$$g = \mathrm{ggT}(p, q) = ap + bq, \qquad g, p, q, a, b \in K[x]$$

wird durch `p.xgcd(q)` geliefert:

```
sage: R.<x> = QQ[]; p = x^5-1; q = x^3-1
sage: print 'Der ggT ist %s = (%s)*p + (%s)*q.' % p.xgcd(q)
Der ggT ist x - 1 = (-x)*p + (x^3 + 1)*q.
```

Die Methode `xgcd` existiert auch für Polynome aus `ZZ['x']`, aber Vorsicht: da der Ring $\mathbb{Z}[x]$ kein Hauptidealring ist, ist das Ergebnis im allgemeinen keine Bézout-Identität!

7.2.2. Ideale und Quotientenringe

Ideale aus $A[x]$. Die Ideale der Polynomringe und die Quotientenringe dieser Ideale werden durch Sage-Objekte dargestellt, die mit `ideal` und `quo` aus Polynomringen gebildet werden. Das Produkt eines Tupels von Polynomen mit einem Polynomring wird als Ideal interpretiert:

```
sage: R.<x> = QQ[]
sage: J1 = (x^2 - 2*x + 1, 2*x^2 + x - 3)*R; J1
Principal ideal (x - 1) of Univariate Polynomial Ring in x over Rational Field
```

Wir können die Ideale multiplizieren und ein Polynom modulo eines Ideals reduzieren:

```
sage: J2 = R.ideal(x^5 + 2)
sage: ((3*x+5)*J1*J2).reduce(x^10)
421/81*x^6 - 502/81*x^5 + 842/81*x - 680/81
```

In diesem Fall bleibt das reduzierte Polynom ein Element von `QQ['x']`. Eine andere Möglichkeit besteht darin, den Quotientenring duch ein Ideal zu bilden und die Elemente darauf zu projizieren. Dessen Vorfahr ist dann der Quotientenring. Die Methode `lift` der Elemente des Quotientenrings dient dann dazu, sie wieder in den Ausgangsring zurückzuführen.

```
sage: B = R.quo((3*x+5)*J1*J2)  # quo heißt automatisch 'xbar'
sage: B(x^10)                   # der Generator des Bildes B von x
421/81*xbar^6 - 502/81*xbar^5 + 842/81*xbar - 680/81
sage: B(x^10).lift()
421/81*x^6 - 502/81*x^5 + 842/81*x - 680/81
```

Ist K ein Körper, ist $K[x]$ ein Hauptidealring: Ideale werden bei den Rechnungen durch einen Generator dargestellt, und all das ist wohl kaum eine besonders algebraische Sprache für die in Unterabschnitt 7.2.1 gezeigten Operationen. Ihr Hauptverdienst ist, dass die Quotientenringe in neuen Konstrktionen mühelos verwendet werden können, hier die von $\left(F_5[t]/(t^2+3)\right)[x]$:

```
sage: R.<t> = GF(5)[]; R.quo(t^2+3)['x'].random_element()
(2*tbar + 1)*x^2 + (2*tbar + 1)*x + 2*tbar
```

Sage erlaubt auch die Bildung von Idealen, die keine Hauptidealringe sind wie $\mathbb{Z}[x]$, doch sind die verfügbaren Funktionen begrenzt - außer im Fall von Polynomen mit mehreren Unbestimmten auf einem Körper. Das ist Gegenstand von Kapitel 9.

Übung 26. Wir definieren die Folge $(u_n)_{n\in\mathbb{N}}$ durch die Anfangsbedingungen $u_n = n+7$ für $0 \le n < 1000$ und die Relation der linearen Rekursion

$$u_{n+100} = 23u_{n+729} - 5u_{n+2} + 12u_{n+1} + 7u_n \qquad (n \ge 0).$$

Berechnen Sie die letzten fünf Ziffern von $u_{10^{10000}}$. *Hinweis:* wir können uns durch den Algorithmus in Unterabschnitt 3.2.4 inspirieren lassen. Der braucht aber zu lange, wenn die Ordnung der Rekursion hoch ist. Führen Sie einen sinnvollen Quotientenring ein, um dieses Problem zu vermeiden.

Bildung von Idealen und Quotientenringen $Q = R/J$	
Ideal $< u, v, w >$	`R.ideal(u, v, w)` oder `(u, v, w)*R`
Reduktion von p modulo J	`J.reduce(p)` oder `p.mod(J)`
Quotientenring $R/J, R/(p)$	`R.quo(J), R.quo(p)`
Q zu einem Quotientenring gemacht	`Q.cover_ring()`
Körper der isomorphen Zahlen	`Q.number_field()`

Elemente von $K[x]/p$	
erhoben (Schnitt von $R \to R/J$)	`u.lift()`
minimales Polynom	`u.minpoly()`
charakteristisches Polynom	`u.charpoly()`
Matrix	`u.matrix()`
Spur	`u.trace()`

Tab. 7.4 - Ideale und Quotientenringe

Algebraische Erweiterungen. Ein wichtiger Sonderfall ist der Quotient von $K[x]$ durch ein irreduzibles Polynom, um eine algebraische Erweiterung von K zu realisieren. Die Körper der Zahlen, endliche Erweiterungen von $\mathbb{Q}$, werden durch `NumberField`-Objekte dargestellt und sind von Quotienten von `QQ['x']` verschieden. Wenn das Sinn macht, gibt die Methode `number_field` eines Quotientenringes von Polynomen den Körper der entsprechenden Zahlen zurück. Die Schnittstelle des Körpers der Zahlen, der umfangreicher ist als der der Quotientenringe, sprengt den Rahmen dieses Buches. Die als algebraische Erweiterungen der primen endlichen Körper $\mathbb{F}_p$ realisierten zusammengesetzten endlichen Körper $\mathbb{F}_{p^k}$ sind im Abschnitt 6.1 beschrieben.

7.3. Faktorisierung und Wurzeln

Eine dritte Ebene nach den elementaren Operationen und der euklidschen Arithmetik betrifft die Zerlegung von Polynomen in Produkte irreduzibler Faktoren oder die Faktorisierung. Das ist vielleicht das Gebiet, wo sich das symbolische Rechnen am nützlichsten erweist.

7.3.1. Faktorisierung

Prüfung auf Irreduzierbarkeit. Auf der algebraischen Ebene ist die einfachste Frage, die die Faktorisierung eines Polynoms betrifft, ob es als Produkt zweier nichttrivialer Faktoren geschrieben werden kann, oder, im Gegenteil, irreduzibel ist. Natürlich hängt die Antwort vom Basisring ab. Die Methode `is_irreducible` zeigt, ob ein Polynom auf seinem Vorfahrring irreduzibel ist. Beispielsweise ist das Polynom $3x^2 - 6$ auf $\mathbb{Q}$ reduzibel, auf $\mathbb{Z}$ aber nicht (warum?).

```
sage: R.<x> = QQ[]; p = 3*x^2 - 6
sage: p.is_irreducible(), p.change_ring(ZZ).is_irreducible()
(True, False)
```

Faktorisierung. Die Primfaktorzerlegung einer ganzen Zahl mit hunderten oder tausenden von Stellen ist eine sehr schwierige Aufgabe. Dagegen braucht die Faktorisierung eines Polynoms 1000. Grades auf $\mathbb{Q}$ oder $\mathbb{F}_p$ nur ein paar Sekunden Prozessorzeit[4]:

```
sage: p = QQ['x'].random_element(degree=1000)
sage: timeit('p.factor()')
5 loops, best of 3: 5.23 s per loop
```

Daher endet hier die algorithmische Ähnlichkeit von Polynomen und ganzen Zahlen, die wir in den vorhergehenden Abschnitten beobachten konnten. Genau wie der Test auf Irreduzibilität findet die Faktorisierung auf dem Basisring des Polynoms statt. Beispielsweise setzt sich die Faktorisierung eines Polynoms auf $\mathbb{Z}$ aus einem konstanten Teil, der in Primzahlen zerlegt ist, und einem Produkt von primitiven Polynomen zusammen, die untereinander teilerfremd sind:

```
sage: x = polygen(ZZ); p = 54*x^4 + 36*x^3 - 102*x^2 - 72*x - 12
sage: p.factor()
2 * 3 * (3*x + 1)^2 * (x^2 - 2)
```

Sage erlaubt die Zerlegung auf verschiedenen Ringen - rationalen, komplexen (näherungsweise), endlichen Körpern und besonders Zahlenkörpern:

```
sage: for A in [QQ, ComplexField(16), GF(5), QQ[sqrt(2)]]:
....:     print A, ':'; print A['x'](p).factor()
Rational Field :
(54) * (x + 1/3)^2 * (x^2 - 2)
Complex Field with 16 bits of precision :
(54.00) * (x - 1.414) * (x + 0.3333)^2 * (x + 1.414)
Finite Field of size 5 :
(4) * (x + 2)^2 * (x^2 + 3)
Number Field in sqrt2 with defining polynomial x^2 - 2 :
(54) * (x - sqrt2) * (x + sqrt2) * (x + 1/3)^2
```

Faktorisierung	
Test auf Irreduzibilität	`p.is_irreducible`
Faktorisierungt	`p.factor()`
quadratfreie Faktorisierungt	`p.squarefree_decomposition()`
quadratfreiet Teil $p/ggT(p,p')$t	`p.radical()`

Wurzeln	
Wurzeln in A, in D	`p.roots(), p.roots(D)`
reelle Wurzeln	`p.roots(RR), p.real_roots()`
komplexe Wurzeln	`p.roots(CC), p.complex_roots()`
Isolation der reellen Wurzeln	`p.roots(RIF)`
Isolation der komplexen Wurzeln	`p.roots(CIF)`
Resultante	`p.resultant(q)`
Diskriminante	`p.discriminant()`
Galois-Gruppe (p irreduzibel)	`p.galois_group()`

Tab. 7.5 - Faktorisierung und Wurzeln

[4] Von einem theoretischen Standpunkt können wir in $\mathbb{Q}[x]$ in polynomialer Zeit faktorisieren und in $\mathbb{F}_p[x]$ in probabilistisch polynomialer Zeit, währen wir nicht wissen, ob es möglich ist, ganze Zahlen in polynomialer Zeit zu faktorisieren.

Das Ergebnis einer Zerlegung in irreduzible Faktoren ist kein Polynom (weil Polynome immer in Normalform sind, d.h. ausmultipliziert!), sondern ein Objekt `f` des Typs `Factorization`. Wir können den i-ten Faktor mit `f[i]` finden, und wir gewinnen das Polynom mit `f.expand()` zurück. Die Objekte von `Factorization` verfügen auch über Methoden wie `gcd` oder `lcm`, welche die gleiche Bedeutung haben wie für Polynome, aber auf faktorisierten Formen arbeiten.

Quadratfreie Zerlegung. Trotz seiner guten theoretischen und praktischen Komplexität ist die vollständige Faktorisierung eines Polynoms eine komplexe Operation. Die quadratfreie Zerlegung bildet eine schwächere Form der Zerlegung, viel leichter zu erhalten - einige Berechnungen von ggT genügen - und die schon viel an Information bringt.

Sei $p = \prod_{i=1}^{r} p_i^{m_i} \in K[x]$ ein Polynom, das in irreduzible Faktoren auf einem Körper K zerlegt ist. Wir sagen, p ist quadratfrei (*squarefree* auf englisch), wenn alle Faktoren p_i die Multiplizität $m_i = 1$ haben, d.h. wenn die Wurzeln von p in einem abgeschlossenen Gebiet von K einfach sind. Eine *quadratfreie Zerlegung* ist eine Faktorisierung in ein Produkt von quadratfreien Faktoren, die wechseitig teilerfremd sind:

$$p = f_1 f_2^2 \cdots f_s^s \qquad \text{wobei } f_m = \prod_{m_i = m} p_i.$$

Die quadratfreie Zerlegung ordnet daher die irreduziblen Faktoren von p nach Multiplizitäten. Der *quadratfreie Teil* $f_1 \cdots f_s = p_1 \cdots p_r$ von p ist das Polynom aus den einfachen Wurzeln, das abgesehen von den Multiplizitäten dieselben Wurzeln hat wie p.

7.3.2. Wurzelsuche

Die Berechnung der Wurzeln eines Polynoms lässt viele Varianten zu, je nachdem, ob wir reelle Wurzeln suchen, komplexe oder andere Wurzeln, exakte oder genäherte, mit oder ohne Multiplizitäten, auf garantierte oder auf heuristische Weise... Die Methode `roots` eines Polynoms gibt in der Voreinstellung die Wurzeln in seinem Basisring in Form einer Liste von Paaren (Wurzel, Multiplizität) zurück:

```
sage: R.<x> = ZZ[]; p = (2*x^2 - 5*x + 2)*(x^4 - 7); p.roots()
[(2, 2)]
```

Mit Argument gibt `roots(D)` die Wurzeln im Definitionsbereich D zurück, hier erst die rationalen Wurzeln, dann Näherungen der l-adischen Wurzeln für $l = 19$:

```
sage: p.roots(QQ)
[(2, 2), (1/2, 2)]
sage: p.roots(Zp(19, print_max_terms=3))
[(7 + 16*19 + 17*19^2 + ... + O(19^20), 1),
(12 + 2*19 + 19^2 + ... + O(19^20), 1),
(10 + 9*19 + 9*19^2 + ... + O(19^20), 2),
(2 + O(19^20), 2)]
```

Das funktioniert mit einem breiten Spektrum von Definitionsbereichen und mit unterschiedlicher Effizienz.

Insbesondere erlaubt die Wahl des Körpers der algebraischen Zahlen `QQbar` für D oder der algeraischen reellen `AA` die exakte Berechnung der komplexen oder reelen Wurzeln eines Polynoms mit rationalen Koeffizienten:

```
sage: wurzeln = p.roots(AA); wurzeln
[(-1.626576561697786?, 1),
 (0.500000000000000?, 2),
 (1.626576561697786?, 1),
 (2.000000000000000?, 2)]
```

Sage jongliert auf für den Anwender transparente Weise mit verschiedenen Darstellungsweisen der algebraischen Zahlen. Eine besteht beispielsweise darin, jedes $\alpha \in \bar{\mathbb{Q}}$ über sein an eine hinreichend genaue Einschachtelung gekoppeltes minimales Polynom zu kodieren, um α von anderen Wurzeln zu unterscheiden. So sind die zurückgegebenen Wurzeln dem Augenschein zum Trotz nicht bloße Näherungswerte. Sie können in exakten Rechnungen weiterverwendet werden:

```
sage: a = wurzeln[0][0]^4; a.simplify(); a
7
```

Hier haben wir die erste gefundene Wurzel zur vierten Potenz erhoben und Sage dann gezwungen, das Resultat ausreichend zu vereinfachen, um zu verdeutlichen, dass es sich um die ganze Zahl 7 handelt.

Eine Variante der exakten Lösung besteht darin, die Wurzeln einfach zu *isolieren*, d.h. Intervalle zu berechnen, die jeweils genau eine Wurzel enthalten, wozu als Definitionsbereich D derjenige der reellen Intervalle `RIF` oder der komplexen `CIF` übergeben wird. Von den anderen Definitionsmengen im Fall eines Polynoms mit rationalen Koeffizienten erwähnen wir `RR, CC, RDF` und `CDF`, die alle zahlenmäßig genäherten Wurzeln entsprechen wie der Körper der Zahlen `QQ[alpha]`. Die spezifischen Methoden `real_roots, complex_roots` und (für bestimmte Basisringe) `real_root_intervals` bieten zusätzliche Optionen oder liefern vom Aufruf von `roots` leicht abweichende Ergebnisse. Die Suche nach und die Isolation von numerischen Wurzeln wird genauer in Abschnitt 12.2 behandelt

7.3.3. Resultante

Auf jedem Faktorring wird die Existenz eines nicht konstanten, zwei Polynomen gemeinamen Faktors durch die Annullierung ihrer *Resultanten* $\mathrm{Res}(p, q)$ charakterisiert, welche ein Polynom in deren Koeffizienten ist. Ein größeres Interess an der Resultante bezüglich des ggT beruht darauf, dass sie sich auf Morphismen von Ringen *spezialiert*. Zum Beispiel sind die Polynome $x-12$ und $x-20$ in $\mathbb{Z}[x]$ teilerfremd, doch zeigt die Annulation modulo n ihrer Resultanten

```
sage: x = polygen(ZZ); (x-12).resultant(x-20)
-8
```

dass sie in $\mathbb{Z}/nZ$ genau dann eine gemeinsame Wurzel haben, wenn 8 von n geteilt wird.

Seien $p = \sum_{i=0}^{m} p_i x^i$ und $q = \sum_{i=0}^{n} q_i x^i$ zwei nicht konstante Polynome aus $A[x]$ mit $p_m, q_n \neq 0$. Die Resultante von p und q ist definiert durch

$$\mathrm{Res}(p,q) = \begin{vmatrix} p_m & \cdots & \cdots & p_0 & & \\ & \ddots & & & \ddots & \\ & & p_m & \cdots & \cdots & p_0 \\ q_n & \cdots & q_0 & & & \\ & \ddots & & \ddots & & \\ & & \ddots & & \ddots & \\ & & & q_n & \cdots & q_0 \end{vmatrix}. \tag{7.1}$$

Das ist die Determinante in passenden Basen der linearen Abbildung

$$\begin{aligned} A_{n-1}[x] \times A_{m-1}[x] &\to A_{m+n-1}[x] \\ u, v &\mapsto up + vq \end{aligned}$$

wobei $A_k[x] \subset A[x]$ den Untermodul des Polynoms höchstens vom Grad k bezeichnet. Sind p und q zerlegt, dann wird ihre Resultante auch als Funktion der Differenzen ihrer Wurzeln geschrieben:

$$\mathrm{Res}(p,q) = p_m^n q_n^m \prod_{i,j} (\alpha_i - \beta_j), \quad \begin{cases} p = p_m(x-\alpha_1)\cdots(x-\alpha_m) \\ q = q_n(x-\beta_1)\cdots(x-\beta_n). \end{cases}$$

Die Eigenschaft der weiter oben erwähnten Spezialisierung leitet sich aus der Definition (7.1) her: wenn $\varphi : A \to A'$ ein Ringmorphismus ist, dessen Anwendung auf p und q deren Grade nicht sinken lässt, oder anders gesagt, sodass $\varphi(p_m) \neq 0$ und $\varphi(q_n) \neq 0$ ist, dann haben wir

$$\mathrm{Res}(\varphi(p), \varphi(q)) = \varphi(\mathrm{Res}(p,q)).$$

Somit verschwindet $\varphi(\mathrm{Res}(p,q))$, wenn $\varphi(p)$ und $\varphi(q)$ einen gemeinsamen Faktor haben. Wir haben ein wenig weiter oben ein Beispiel dieser Erscheinung mit der kanonischen Projektion von $\mathbb{Z}$ nach $\mathbb{Z}/n\mathbb{Z}$ für φ gesehen.

Doch die häufigste Anwendung einer Resultante betrifft den Fall, wo der Basisring selbst ein Polynomring ist: $p, q, \in A[x]$ mit $A = K[a_1, \ldots, a_k]$. Seien insbesondere $\alpha_1, \ldots, \alpha_k \in K$ und betrachten wir die Spezialisierung

$$\begin{array}{rccc} \varphi: & B[a_1, \ldots, a_k] & \to & K \\ & q(a_1, \ldots, a_k) & \mapsto & q(\alpha_1, \ldots, \alpha_k) \end{array}$$

Man sieht dann, dass sich die Resultante genau dann in $q(\alpha_1, \ldots, \alpha_k)$ aufhebt, wenn die Spezialisierungen $\varphi(p), \varphi(q) \in K[x]$ einen Faktor gemeinsam haben *unter dem Vorbehalt, dass* einer der führenden Terme von p oder q in $(\alpha_1, \ldots, \alpha_k)$ von null verschieden ist.

Beispielsweise ist die Diskriminante von p definiert durch

$$\mathrm{disk}(p) = (-1)^{m(m-1)/2} \mathrm{Res}(p, p')/p_m.$$

Diese Definition verallgemeinert die klassischen Diskriminanten der Polynome 2. und 3. Grades:

```
sage: R.<a,b,c,d> = QQ[]; x = polygen(R); p = a*x^2 + b*x + c
sage: p.resultant(p.derivative())
```

```
-a*b^2 + 4*a^2*c
sage: p.discriminant()
b^2 - 4*a*c
sage: (a*x^3 + b*x^2 + c*x + d).discriminant()
b^2*c^2 - 4*a*c^3 - 4*b^3*d + 18*a*b*c*d - 27*a^2*d^2
```

Wie die Diskriminante von p verschwindet die Resultante von p und seiner Ableitung genau dann, wenn p eine mehrfache Wurzel hat.

7.3.4. Galoisgruppen

Die Galoisgruppe eines irreduziblen Polynoms $p \in \mathbb{Q}[x]$ ist ein algebraisches Objekt, das bestimmte „Symmetrien“ der Wurzeln von p beschreibt. Es handelt sich um ein zentrales Objekt der Theorie der algebraischen Gleichungen. Insbesondere ist die Gleichung $p(x) = 0$ durch Radikale lösbar, d.h. die Lösungen werden ausgehend von den Koeffizienten des Polynoms genau dann mit vier Operationen und dem Ziehen der n-ten Wurzel ausgedrückt, wenn die Galoisgruppe von p *lösbar* ist.

Sage gestattet die Berechnung der Galoisgruppen von Polynomen nicht zu hohen Grades mit rationalen Koeffizienten und alle Arten von Operationen auf den erhaltenen Gruppen auszuführen. Sowohl die Galoistheorie als auch Sages Funktionalitäten für Gruppen überschreiten den Rahmen dieses Buches. Beschränken wir uns darauf, den Satz von Galois ohne weitere Erläuterungen auf die Lösbarkeit mit Radikalen anzuwenden. Die folgende Rechnung erfordert eine Liste endlicher Gruppen, die in der Basisinstallation von Sage nicht enthalten ist, die man aber mit dem Befehl `./sage -i database_gap` aus dem Netz herunterladen und automatisch installieren kann. [5] Sie zeigt, dass sich die Wurzeln von $x^5 - x - 1$ nicht durch Radikale ausdrücken lassen.

```
sage: x = polygen(QQ); G = (x^5 - x - 1).galois_group(); G
Transitive group number 5 of degree 5
sage: G.is_solvable()
False
```

Es handelt sich um eines der einfachsten Beispiele für diesen Fall, da die Polynome 4. oder kleineren Grades immer mit Radikalen lösbar sind, genau so offensichtlich wie $x^5 - a$. Bei der Prüfung der Generatoren von G als Gruppe von Permutationen erkennen wir, dass $G \simeq \mathfrak{S}_5$ gilt, was wir leicht verfizieren:

```
sage: G.gens()
[(1,2,3,4,5), (1,2)]
sage: G.is_isomorphic(SymmetricGroup(5))
True
```

[5] Vorsicht: erstens dauert die Installation recht lange (eine Stunde oder mehr) und es kann passieren, dass danach in Sage nicht mehr alle bisherigen Funktionalitäten verfügbar sind.

7.4. Gebrochen rationale Ausdrücke

7.4.1. Erzeugung und elementare Eigenschaften

Die Division zweier Polynome (auf einem Integritätsring) ergibt einen gebrochen rationalen Ausdruck. Dessen Vorfahr ist der Körper der Brüche des Polynomrings, der mit `Frac(R)` erhalten wird:

```
sage: x = polygen(RR); r = (1 + x)/(1-x^2); r.parent()
Fraction Field of Univariate Polynomial Ring in x over Real
Field with 53 bits of precision
sage: r
(x + 1.00000000000000)/(-x^2 + 1.00000000000000)
```

Wir beobachten, dass die Vereinfachung nicht automatisch erfolgt. Dies, weil `RR` kein exakter Ring ist, d.h. seine Elemente werden als Näherungen mathematischer Objekte interpretiert. Die Methode `reduce` setzt den Bruch in die gekürzte Form. Sie gibt kein neues Objekt zurück, sondern modifiziert den vorhandenen gebrochen rationalen Ausdruck:

```
sage: r.reduce(); r
1.00000000000000/(-x + 1.00000000000000)
```

Umgekehrt werden die gebrochen rationalen Ausdrücke auf einem exakten Ring automatisch gekürzt.

Die Operationen auf gebrochen rationalen Ausdrücken sind analog zu denen auf Polynomen. Diejenigen, die in beiden Fällen sinnvoll sind (Substitution, Ableitung, Faktorisierung…) werden auf gleiche Weise verwendet. Die Tabelle 7.6 listet weitere nützliche Methoden auf. Die Zerlegung in einfache Elemente und vor allem die rationale Rekonstruktion verdienen einige Erläuterungen.

Gebrochen-rationale Ausdrücke	
Körper der Brüche $K(x)$	`Frac(K['x'])`
Zähler	`r.numerator()`
Nenner	`r.denominator()`
Vereinfachung (modifiziert `r`)	`r.reduce()`
Zerlegung in einfache Elemente	`r.partial_fraction_decomposition()`
rationale Rekonstruktion von s mod m	`s.rational_reconstruct(m)`

Abgeschnittene Reihen	
Ring $A[[t]]$	`PowerSeriesRing(A, 'x', default_prec=n)`
Ring $A((t))$	`LaurentSeriesRing(A, 'x', default_prec=n)`
Koeffizient $[x^k]f(x)$	`f[k]`
Abschätzung des Wertes	`x + O(x^n)`
Genauigkeit	`f.prec()`
Ableitung, Integral	`f.derivative(), f.integral()`
häufige Operationen $\sqrt{f}$, exp, …	`f.sqrt(), f.exp(), ...`
Reziprok-Wert ($f \circ g = g \circ f = x$)	`g = f.reversion()`
Lösung von $y' = ay + b$	`a.solve_linear_de(precision, b)`

Tab. 7.6 – Mit Polynomen gebildete Objekte.

7.4.2. Zerlegung in einfache Elemente

Sage berechnet die Zerlegung eines gebrochen rationalen Ausdrucks a/b aus `Frac(K['x'])` in einfache Elemente, indem es mit b aus `K['x']` beginnt. Daher handelt es sich um die Zerlegung in einfache Elemente auf K. Das Ergebnis besteht aus einem Polynom und einer Liste von gebrochen rationalen Ausdrücken, deren Nenner Potenzen der irreduziblen Faktoren von b sind:

```
sage: R.<x> = QQ[]; r = x^10 / ((x^2-1)^2 * (x^2+3))
sage: poly, parts = r.partial_fraction_decomposition()
sage: poly
x^4 - x^2 + 6
sage: for part in parts: part.factor()
(17/32) * (x - 1)^-1
(1/16) * (x - 1)^-2
(-17/32) * (x + 1)^-1
(1/16) * (x + 1)^-2
(-243/16) * (x^2 + 3)^-1
```

Somit haben wir die Zerlegung in einfache Elemente auf rationalen Zahlen erhalten

$$r = \frac{x^{10}}{(x^2-1)^2(x^3+3)} = x^4 - x^2 + 6 + \frac{\frac{17}{32}x - \frac{15}{32}}{(x-1)^2} + \frac{-\frac{17}{32}x - \frac{15}{32}}{(x+1)^2} + \frac{-\frac{243}{16}}{x^2+3}.$$

Es is unschwer zu sehen, dass dies auch die Zerlegung von r auf den reellen Zahlen ist.

Auf den komplexen Zahlen hingegen ist der Nenner des letzten Terms nicht irreduzibel, und deshalb kann der gebrochen rationale Ausdruck weiter zerlegt werden. Wir können die Zerlegung in einfache Elemente auf den reellen oder komplexen Zahlen numerisch berechnen:

```
sage: C = ComplexField(15)
sage: Frac(C['x'])(r).partial_fraction_decomposition()
(x^4 - x^2 + 6.000,
(x^4 - x^2 + 6.000,
[0.5312/(x - 1.000),
 0.06250/(x^2 - 2.000*x + 1.000),
 4.385*I/(x - 1.732*I),
 (-4.385*I)/(x + 1.732*I),
 (-0.5312)/(x + 1.000),
 0.06250/(x^2 + 2.000*x + 1.000)])
```

Die exakte Zerlegung auf $\mathbb{C}$ erhalten wir auf die gleiche Weise, indem wir nämlich `C` durch `QQbar` ersetzen. Bei der Berechnung auf `AA` hätten wir die Zerlegung auf den reellen Zahlen, selbst wenn nicht alle reellen Wurzeln des Nenners rational sind.

7.4.3. Rationale Rekonstruktion

Ein Analogon zu der in Unterabschnitt 6.1.3 vorgestellten rationalen Rekonstruktion existiert auch für Polynome in $A = \mathbb{Z}/n\mathbb{Z}$. Sind $m, s \in A[x]$ gegeben, berechnet der Befehl

```
sage: s.rational_reconstruct(m, dp, dq)
```

falls möglich die Polynone $p, q \in A[x]$, sodass

$$qs \equiv p \bmod m, \qquad \deg p \leq d_p, \qquad \deg q \leq d_q.$$

Beschränken wir uns auf die Vereinfachung für den Fall, dass n eine Primzahl ist. Eine solche Beziehung zwischen den teilerfremden q und m zieht $p/q = s$ in $A[x]/ < m >$ nach sich, daher die Bezeichnung *rationale Rekonstruktion*.

Das Problem der rationale Rekonstruktion wird in ein lineares Gleichungssystem mit Koeffizienten von p und q übersetzt, und ein einfaches Dimensionsargument zeigt, dass es eine nicht-triviale Lösung gibt, sobald $d_p + d_q \geq \deg m - 1$ ist. Nicht immer gibt es eine Lösung mit teilerfremden p und m (z.B. sind die Lösungen von $p \equiv qx \bmod x^2$ mit $\deg p \leq 0$, $\deg q \leq 1$ die multiplen Konstanten von $(p, q) = (0, x)$), doch `rational_reconstruct` sucht bevorzugt Lösungen für teilerfremde q und m.

Padé-Approximation. Der Fall $m = x^n$ heißt Padé-Approximation. Eine Padé-Näherung des Typs $(k, n-k)$ einer symbolischen Reihe $f \in K[[x]]$ ist ein gebrochen rationaler Ausdruck $p/q \in K(x)$, sodass $\deg p \leq k-1$, $\deg q \leq n-k$, $q(0) = 1$ und $p/q = f + \mathcal{O}(x^n)$ ist. Wir haben dann $p/q \equiv f \bmod x^n$.

Beginnen wir mit einem rein symbolischen Beispiel. Die folgenden Befehle berechnen eine Padé-Näherung der Reihe $f = \sum_{i=0}^{\infty}(i+1)^2 x^i$ mit Koeffizienten in $\mathbb{Z}/10\mathbb{Z}$:

```
sage: A = Integers(101); R.<x> = A[]
sage: f6 = sum( (i+1)^2 * x^i for i in (0..5) ); f6
36*x^5 + 25*x^4 + 16*x^3 + 9*x^2 + 4*x + 1
sage: num, den = f6.rational_reconstruct(x^6, 1, 3); num/den
(100*x + 100)/(x^3 + 98*x^2 + 3*x + 100)
```

Bei erneuter Reihenentwicklung des gefundenen gebrochen rationalen Ausdrucks beobachten wir, das die Entwicklungen nicht nur bis zur Ordnung 6 zusammenfallen, sondern der folgende Term sogar „richtig“ ist.

```
sage: S = PowerSeriesRing(A, 'x', 7); S(num)/S(den)
1 + 4*x + 9*x^2 + 16*x^3 + 25*x^4 + 36*x^5 + 49*x^6 + O(x^7)
```

In der Tat ist f selbst gebrochen rational: wir haben $f = (1 + x/(1-x)^3)$. Die abbrechende Entwicklung `f6` mit begrenzten Graden von Zähler und Nenner genügt für die eindeutige Darstellung. So gesehen, ist die Berechnung von Padé-Näherungen die Umkehrung der Reihenentwicklung von gebrochen rationalen Ausdrücken: sie erlaubt, von dieser alternativen Darstellung zur üblichen Darstellung als Quotient zweier Polynome zurückzukehren.

Ein analytisches Beispiel. Historisch sind die Padé-Approximationen nicht bei dieser Art von symbolischen Betrachtungen entstanden, sondern im Zusammenhang mit der Theorie der Approximation analytischer Funktionen. Tatsächlich nähern die Padé-Approximationen einer Reihenentwicklung die Funktion oft besser an als abbrechende Reihen. Wenn der Grad des Nenners groß genug ist, können sie sogar außerhalb des Konvergenzradius der Reihe gute Näherungen liefern. Wir sagen manchmal, dass sie „die Pole verschlucken“. Die Abbildung 7.1, welche die Konvergenz der Näherungen der Tangensfunktion des Typs $(2k, k)$ in der Umgebung von 0 zeigt, illustriert dieses Phänomen.

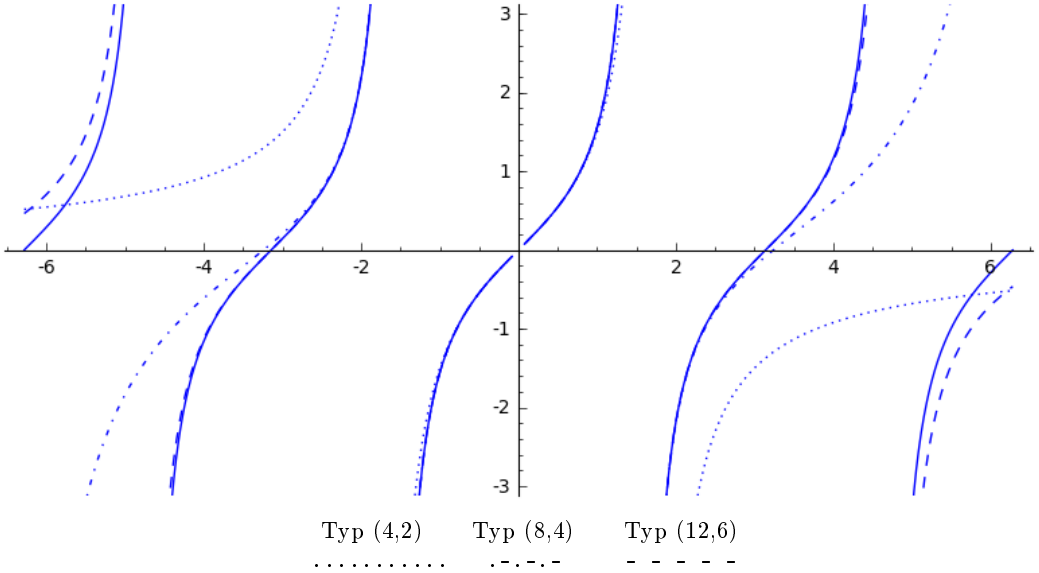


Abb. 7.1 - Die Tangensfunktion und einige Padé-Approximationen auf $[-2\pi, 2\pi]$.

Obwohl `rational_reconstruct` auf Polynome in $\mathbb{Z}/n\mathbb{Z}$ beschränkt ist, ist es damit möglich, Padé-Approximationen mit rationalen Koeffizienten zu berechnen und diese Abbildung zu erzielen. Am einfachsten beginnt man damit, die rationale Rekonstruktion modulo einer recht großen Primzahl auszuführen:

```
sage: x = var('x'); s = tan(x).taylor(x, 0, 20)
sage: p = previous_prime(2^30); ZpZx = Integers(p)['x']
sage: Qx = QQ['x']

sage: num, den = ZpZx(s).rational_reconstruct(ZpZx(x)^10,4,5)
sage: num/den
(1073741779*x^3 + 105*x)/(x^4 + 1073741744*x^2 + 105)
```

um dann die gefundene Lösung anzuheben. Die folgende Funktion hebt ein Element a aus $\mathbb{Z}/p\mathbb{Z}$ zu einer ganzen Zahl mit dem Absolutwert von höchstens $p/2$.

```
sage: def lift_sym(a):
....:     m = a.parent().defining_ideal().gen()
....:     n = a.lift()
....:     if n <= m // 2: return n
....:     else: return n - m
```

Wir erhalten:

```
sage: Qx(map(lift_sym, num))/Qx(map(lift_sym, den))
(-10*x^3 + 105*x)/(x^4 - 45*x^2 + 105)
```

Wenn die gesuchten Koeffizienten für dieses Verfahren zu groß sind, können wir die Rechnung modulo mehrerer Primzahlen ausführen und den chinesischen Restsatz anwenden,um eine Lösung mit ganzzahligen Koeffizienten zu finden wie in Unterabschnitt 6.1.4 erklärt.

Eine andere Möglichkeit ist die Berechnung einer Rekursionsbeziehung mit konstanen Koeffizienten, die den Koeffizienten der Reihe genügt. Diese Rechnung ist zur Berechnung einer Padé-Approximation nahezu äquivalent (siehe Übung 27), doch erlaubt Sages Funktion `berlekamp_massey` die Ausführung auf beliebigen Körpern.

Systematisieren wir die vorstehende Rechnung, indem wir eine Funktion schreiben, welche die Näherung mit rationalen Koeffizienten unter hinreichend förderlichen Annahmen direkt berechnet:

```
sage: def mypade(pol, n, k):
....:     x = ZpZx.gen();
....:     n,d = ZpZx(pol).rational_reconstruct(x^n, k-1, n-k)
....:     return Qx(map(lift_sym, n))/Qx(map(lift_sym, d))
```

Wir müssen auf die Ausgaben dieser Funktion nur noch `plot` anwenden, um die Grafik in Abbildung 7.1 zu erhalten (Die Ausgaben werden in Elemente von `SR` umgewandelt, denn `plot` kann den Graphen einer „algebraischen“ rationalen Funktion nicht unmittelbar zeichnen.)

```
sage: add(
....:     plot(expr, -2*pi, 2*pi, ymin=-3, ymax=3,
....:             linestyle=sty, detect_poles=True, aspect_ratio=1)
....:     for (expr, sty) in [
....:         (tan(x), '-'),
....:         (SR(mypade(s, 4, 2)), ':' ),
....:         (SR(mypade(s, 8, 4)), '-.'),
....:         (SR(mypade(s, 12, 6)), '--') ])
```

Die folgenden Übungen stellen weitere klassische Anwendungen der rationalen Rekonstruktion dar.

Übung 27.

1. Zeigen Sie: wenn $(u_n)_{n\in\mathbb{N}}$ einer linearen Rekursion mit konstanten Koeffizienten genügt, dann ist die symbolische Reihe $\sum_{n\in\mathbb{N}} u_n z^n$ ein gebrochen rationaler Ausdruck. Wie werden Zähler und Nenner repräsentiert?

2. Erraten Sie die nächsten Terme der Folge

$$1, 1, 2, 3, 8, 11, 34, 39, 148, 127, 662, 339, 3056, 371, 14602, -4257, \ldots,$$

   indem Sie `rational_reconstruct` verwenden. Finden Sie das Ergebnis auch mit der Funktion `berlekamp_massey`.

Übung 28 *(Interpolation von Cauchy).* Finden Sie einen gebrochen rationalen Ausdruck $r = p/q \in \mathbb{F}_{17}(x)$, sodass $r(0) = -1$, $r(1) = 0$, $r(2) = 7$ und $r(3) = 5$ wird, wobei p von möglichst kleinem Grad ist.

7.5. Formale Potenzreihen

Eine formale Potenzreihe ist eine Potentreihe, die als eine einfache Folge von Koeffizienten erscheint, ohne dass ihre Konvergenz untersucht wird. Genauer, wenn A ein kommutativer Ring ist, sind formale Potenzreihen (engl. *formal power series*) einer Unbestimmten x mit Koeffizienten aus A die symbolischen Summen $\sum_{n=0}^{\infty} a_n x^n$, wobei (a_n) eine Folge beliebiger Elemente aus A ist. Ausgestattet mit den Operationen der üblichen Addition und Multiplikation

$$\sum_{n=0}^{\infty} a_n x^n + \sum_{n=0}^{\infty} b_n x^n = \sum_{n=0}^{\infty} (a_n + b_n) x^n,$$
$$\left(\sum_{n=0}^{\infty} a_n x^n\right)\left(\sum_{n=0}^{\infty} b_n x^n\right) = \sum_{n=0}^{\infty}\left(\sum_{i+j=n} a_i b_j\right) x^n$$

bilden die formalen Potenzreihen einen mit $A[[x]]$ bezeichneten Ring.

In einem System zum symbolischen Rechnen werden mit solchen Reihen analytische Funktionen dargestellt, die nicht in exakter Schreibweise vorliegen. Wie immer führt der Computer Rechnungen aus, es ist aber Sache des Anwenders, ihnen einen mathematischen Sinn zu geben. Er muss sicherstellen, dass die Reihen, mit denen er arbeitet, konvergent sind.

Formale Potenzreihen begegnen auch ausgiebig in der Kombinatorik, insofern es erzeugende Reihen sind. In Unterabschnitt 15.1.2 werden wir ein Beispiel dafür sehen.

7.5.1. Operationen auf abbrechenden Reihen

Den Ring der formalen Potenzreihen $\mathbb{Q}[[x]]$erhalten wir mit

```
sage: R.<x> = PowerSeriesRing(QQ)
```

oder dessen Abkürzung `R.<x> = QQ[[]]`[6] mit `QQ['x'].completion('x')`. Die Elemente von `A[['x']]` sind die abbrechenden Reihen, d.h. Objekte der Form

$$f = f_0 + f_1 x + \ldots + f_{n-1} x^{n-1} + \mathcal{O}(x^n).$$

Sie spielen die Rolle von Approximationen der „mathematischen“ unendlichen Reihen, genau wie die Elemente von `RR` die Näherungen der reellen Zahlen darstellen. Der Ring `A[[x]]` ist daher kein exakter Ring.

Jede Reihe besitzt ihre eigene Abbruchordnung[7] und die Genauigkeit ergibt sich im Verlauf der Rechnungen automatisch:

```
sage: R.<x> = QQ[[]]
sage: f = 1 + x + O(x^2); g = x + 2*x^2 + O(x^4)
sage: f + g
1 + 2*x + O(x^2)
sage: f * g
x + 3*x^2 + O(x^3)
```

[6] Oder aus $\mathbb{Q}[x]$.

[7] In bestimmter Hinsicht ist der Hauptunterschied zwischen einem Polynom modulo x^n und einer abbrechenden Reihe der Ordnung n: die Operationen auf beiden Arten von Objekten sind analog, aber die Elemente von $A[[x]]/\langle x^n\rangle$ haben alle dieselbe „Genauigkeit“.

Es gibt Reihen unendlicher Genauigkeit, die den Polynomen genau entsprechen:

```
sage: (1 + x^3).prec()
+Infinity
```

Eine voreingestellte Genauigkeit kommt zur Anwendung, wenn es erforderlich ist, ein exaktes Resultat abzuschneiden. Sie wird mit der Erzeugung des Ringes festgelegt oder danach mit der Methode `set_default_prec`:

```
sage: R.<x> = PowerSeriesRing(Reals(24), default_prec=4)
sage: 1/(1 + RR.pi() * x)^2
1.00000 - 6.28319*x + 29.6088*x^2 - 124.025*x^3 + O(x^4)
```

Das hat zur Folge, dass ein Test auf mathematische Gleichheit zweier Reihen nicht möglich ist. Dies ist ein wichtiger begrifflicher Unterschied zwischen dieser und anderen Klassen von Objekten, die in diesem Kapitel vorkommen. Sage sieht deshalb zwei Elemente aus `A[['x']]` als gleich an, sobald sie bis auf die schwächste ihrer Genauigkeiten übereinstimmen:

```
sage: R.<x> = QQ[[]]
sage: 1 + x + O(x^2) == 1 + x + x^2 + O(x^3)
True
```

Achtung: das impliziert beispielsweise, dass der Test `O(x^2) == 0` wahr zurückgibt, weil die Nullserie eine unendliche Genauigkeit hat.

Die arithmischen Basisoperationen funktionieren wie bei Polynomen. Wir verfügen auch über einige der üblichen Funktionen, beispielsweise `f.exp()` mit $f(0) = 0$, sowie über Ableitung und Integration. So ist eine asymptotische Entwicklung für $x \to 0$ von

$$\frac{1}{x^2} \exp\left(\int_0^x \sqrt{\frac{1}{1+t}} \mathrm{d}t \right)$$

gegeben durch

```
sage: (1/(1+x)).sqrt().integral().exp() / x^2 + O(x^4)
x^-2 + x^-1 + 1/4 + 1/24*x - 1/192*x^2 + 11/1920*x^3 + O(x^4)
```

Hier wird jede Operation, selbst dann, wenn nur vier Terme im Resultat erscheinen, mit der voreingestellten Genauigkeit von 20 ausgeführt, was weithin ausreicht für ein Endergebnis in $\mathcal{O}(x^4)$. Um mehr als zwanzig Terme zu erhalten, müsste die Genauigkeit der Zwischenrechnungen erhöht werden.

Dieses Beispiel zeigt auch, dass wenn $f, g \in \mathbb{K}[[x]]$ ist und $g(0) = 0$, der Quotient f/g ein Objekt formale *Laurent-Reihe* zurückgibt. Anders als bei Laurent-Reihen der komplexen Analysis der Form $\sum_{n=-\infty}^{\infty} a_n x^n$ sind die formalen Laurent-Reihen Summen des Typs $\sum_{n=-N}^{\infty} a_n x^n$ mit einer endlichen Anzahl von Termen mit negativen Exponenten. Diese Beschränkung ist nötig, damit das Produkt zweier formaler Potenzreihen sinnvoll gebildet wird: ohne sie würde jeder Koeffizient des Produktes als Summe einer unendlichen Reihe ausgedrückt werden.

7.5.2. Reihenentwicklung von Gleichungslösungen

Angesichts einer Differentialgleichung, deren exakte Lösungen zu kompliziert sind, um sie zu berechnen oder, wenn berechnet, zu kompliziert sind für die Auswertung, besteht ein häufiger Ausweg darin, eine Lösung in Form einer Potenzreihe zu suchen. Üblicherweise beginnen wir mit der Bestimmung der Lösungen der Gleichung im Raum der formalen Potenzreihen, und wenn nötig schließen wir dann mit einem Konvergenznachweis, dass die gebildeten formalen Lösungen einen analytischhen Sinn haben. Für die erste Etappe kann Sage eine wertvolle Hilfe sein.

Betrachten wir beispielsweise die Differentialgleichung

$$y'(x) = \sqrt{1+x^2}y(x) + exp(x), \qquad y(0) = 1.$$

Diese Gleichung lässt als eindeutige Lösung eine formale Potenzreihe zu, deren erste Terme berechnet werden können mit

```
sage: R.<x> = PowerSeriesRing(QQ, default_prec=10)
sage: (1+x^2).sqrt().solve_linear_de(prec=6, b=x.exp())
1 + 2*x + 3/2*x^2 + 5/6*x^3 + 1/2*x^4 + 7/30*x^5 + O(x^6)
```

Des weiteren sichert der Satz von Cauchy zur Existenz von Lösungen linearer Differentialgleichungen mit analytischen Koeffizienten, dass diese Potenzreihe für $|x| < 1$ konvergiert: die Summe bildet daher eine analytische Lösung auf dem komplexen Enheitskreis.

Dieser Ansatz ist nicht auf Differentialgleichungen beschränkt. Die Funktionsgleichung $e^{xf(x)} = f(x)$ ist komplexer und das nur, weil sie nicht linear ist. Es ist aber eine Fixpunktgleichung, wir können versuchen, eine (symbolische) Lösung iterativ zu verfeinern:

```
sage: S.<x> = PowerSeriesRing(QQ, default_prec=5)
sage: f = S(1)
sage: for i in range(5):
....:     f = (x*f).exp()
....:     print f
1 + x + 1/2*x^2 + 1/6*x^3 + 1/24*x^4 + O(x^5)
1 + x + 3/2*x^2 + 5/3*x^3 + 41/24*x^4 + O(x^5)
1 + x + 3/2*x^2 + 8/3*x^3 + 101/24*x^4 + O(x^5)
1 + x + 3/2*x^2 + 8/3*x^3 + 125/24*x^4 + O(x^5)
1 + x + 3/2*x^2 + 8/3*x^3 + 125/24*x^4 + O(x^5)
```

Was geht hier vor? Die Lösungen von $e^{xf(x)} = f(x)$ in $\mathbb{Q}[[x]]$ sind Fixpunkte der Transformation $\Phi : f \mapsto e^{xf}$. Wenn eine Folge von Iterierten der Form $\Phi^n(a)$ konvergiert, ist ihr Grenzwert notwendigerweise Lösung der Gleichung. Setzen wir umgekehrt $f(x) = \sum_{n=0}^{\infty} f_n x^n$ und entwickeln wir eine Reihe mit zwei Gliedern: es kommt

$$\begin{aligned}\sum_{n=0}^{\infty} f_n x^n &= \sum_{k=0}^{\infty} \frac{1}{k!} \left(x \sum_{j=0}^{\infty} f_j x^j \right)^k \\ &= \sum_{n=0}^{\infty} \left(\sum_{k=0}^{\infty} \frac{1}{k!} \sum_{\substack{j_1,\ldots j_k \in \mathbb{N} \\ j_1+\ldots+j_k=n-k}} f_{j_1} f_{j_2} \ldots f_{j_k} \right) x^n. \end{aligned} \tag{7.2}$$

Weniger wichtig sind die Details der Formel; wesentlich ist, dass f_n als Funktion der vorangehenden Koeffizienten $f_0, \ldots, f_{n-1}$ berechnet wird, wie man bei der Identifizierung der Koeffizienten der beiden Glieder sieht. Jede Iteration von Φ liefert somit einen neuen korrekten Term.

Übung 29. Berechnen Sie die auf die Ordnung 15 beschränkte Entwicklung von $\tan x$ in der Umgebung von null aus der Differentialgleichung $tan' = 1 + tan^2$.

7.5.3. Faule Reihen

Das Phänomen des Fixpunktes motiviert die Einführung einer weiteren Art formaler Potenzreihen, die faulen Reihen (engl. *lazy*). Das sind keine abbrechenden Reihen sondern unendliche Reihen; das Adjektiv faul besagt, dass die Koeffizienten nur dann berechnet werden, wenn es ausdrücklich verlangt wird. Andererseits können wir nur Reihen darstellen, deren Koeffizienten wir berechnen können: im wesentlichen die Kombinationen von einfachen Reihen mit bestimmten Gleichungslösungen, für die Beziehungen wie (7.2) existieren. Beispielsweise ist die faule Reihe `lazy_exp`, die durch

```
sage: L.<x> = LazyPowerSeriesRing(QQ)
sage: lazy_exp = x.exponential(); lazy_exp
L.<x> = LazyPowerSeriesRing(QQ)
O(1)
```

definiert ist, ein Objekt, das in seiner internen Darstellung alle notwendigen Informationen enthält, um die Reihenentwicklung von $\exp x$ beliebiger Ordnung zu berechnen. Zu Beginn wird sie nur als `O(1)` angezeigt, denn noch ist kein Koeffizient berechnet. Der Versuch, auf den Koeffizienten von x^5 zuzugreifen, löst die Berechnung aus und die berechneten Koeffizienten werden dann gespeichert:

```
sage: lazy_exp[5]
1/120 sage: lazy_exp
1 + x + 1/2*x^2 + 1/6*x^3 + 1/24*x^4 + 1/120*x^5 + O(x^6)
```

Greifen wir das Beispiel von $e^{xf(x)} = f(x)$ wieder auf, um zu sehen, wie es mit faulen Reihen behandelt wird. Wir können zunächst versuchen, die weiter oben im Ring `QQ[['x']]` ausgeführte Rechnung zu reproduzieren:

```
sage: f = L(1)   # die konstante faule Reihe 1
sage: for i in range(5):
....:     f = (x*f).exponential()
....:     f.compute_coefficients(5)  # erzwingt die Berechnung
....:     print f                    # der ersten Koeffizienten
1 + x + 1/2*x^2 + 1/6*x^3 + 1/24*x^4 + 1/120*x^5 + O(x^6)
1 + x + 3/2*x^2 + 5/3*x^3 + 41/24*x^4 + 49/30*x^5 + O(x^6)
1 + x + 3/2*x^2 + 8/3*x^3 + 101/24*x^4 + 63/10*x^5 + O(x^6)
1 + x + 3/2*x^2 + 8/3*x^3 + 125/24*x^4 + 49/5*x^5 + O(x^6)
1 + x + 3/2*x^2 + 8/3*x^3 + 125/24*x^4 + 54/5*x^5 + O(x^6)
```

Die erhaltenen Entwicklungen sind selbstverständlich dieselben wie vorher[8]. Doch ist nun der Wert von f bei jeder Iteration eine unendliche Reihe, deren Koeffizienten wir auf Anforderung berechnen können. Alle diese Reihen werden als Zwischenergebnisse gespeichert. Die Berechnung einer jeden von ihnen mit der gewünschten Genauigkeit wird in der Weise automatisch angestoßen, dass zum Beispiel der Koeffizient von x^7 in der letzten Iteration berechnet wird, sobald wir versuchen, darauf zuzugreifen:

```
sage: f[7]
28673/630
```

Mit dem Code aus Unterabschnitt 7.5.2 hätte der Zugriff auf `f[7]` zu einem Fehler geführt, weil der Index 7 außerhalb der Abbruchordnung der Reihe `f` liegt.

Allerdings ist der von `f[7]` zurückgegebene Wert der Koeffizient von x^7 nur in der Iterierten $\Phi^5(1)$, nicht aber in der Lösung. Die Kraft der faulen Reihen liegt in der Möglichkeit, direkt zur Grenze zu gehen, indem sie selbst als faule Reihe kodiert wird:

```
sage: from sage.combinat.species.series import LazyPowerSeries
sage: f = LazyPowerSeries(L, name='f')
sage: f.define((x*f).exponential())
sage: f.coefficients(8)
[1, 1, 3/2, 8/3, 125/24, 54/5, 16807/720, 16384/315]
```

Was die iterative Berechnung funktionieren lässt, ist die Gleichung (7.2). Hinter den Kulissen leitet Sage aus der rekursiven Definition `f.define((x*f).exponential())` eine Formel derselben Art her, die eine rekursive Berechnung der Koeffizienten erlaubt.

7.6. Rechnerinterne Darstellung von Polynomen

Ein und dasselbe mathematische Objekt - das Polynom p mit Koeffizienten aus A - kann im Rechner auf drei verschiedene Arten codiert werden. Auch wenn das mathematische Ergebnis einer Operation auf p selbstverständlich unabhängig von der Darstellung ist, hängt das Verhalten der entsprechenden Sage-Objekte doch damit zusamen. Die Wahl der Darstellung beeinflusst die möglichen Operationen, die exakte Form ihrer Ergebnisse und besonders die Effizienz der Berechnungen.

Voll und dünn besetzte Darstellung. Für die Darstellung von Polynomen gibt es vor allem zwei Möglichkeiten. Bei der *voll besetzten* Darstellung werden die Koeffizienten von $p = \sum_{i=0}^{n} p_i x^i$ in einer Liste $[p_0, \ldots, p_n]$ mit den Exponenten als Indizes gespeichert. Bei der *dünn besetzten* Speicherung werden nur die von null verschiedenen Koeffizienten gespeichert: das Polynom wird durch eine Menge von Exponent-Koeffizient-Paaren (i, p_i) kodiert, in einer Liste zusammengefasst oder besser in einer von den Exponenten indizierten Raffung (siehe Unterabschnitt 3.3.9).

Für Polynome, die praktisch voll besetzt sind, d.h. deren Koeffizienten meistens von null verschieden sind, verbraucht die volle Darstellung weniger Speicher und erlaubt schnellere

[8] Wir stellen allerdings fest, dass Sage gelegentlich inkohärente Konventionen verwendet: die Methode `exp` für die abbrechenden Reihen heißt hier `exponential`, und `compute_coefficients(5)` berechnet die Koeffizienten bis zur Ordnung 5 einschließlich, wohingegen `default_prec=5` Reihen ergibt, die nach dem Koeffizienten von x^4 abbrechen.

Rechnungen. Sie erspart die Speicherung der Exponenten und der internen Datenstrukturen der Diktionäre: es bleibt nur das absolut Notwendige, die Koeffizienten. Mehr noch, der Zugriff auf ein Element oder die Iteration über die Elemente sind bei einer Liste schneller als bei einem Diktionär. Umgekehrt erlaubt die dünn besetzte Darstellung effiziente Rechnungen auf Polynomen, die bei voll besetzter Darstellung gar nicht in den Speicher passen würden:

```
sage: R = PolynomialRing(ZZ, 'x', sparse=True)
sage: p = R.cyclotomic_polynomial(2^50); p, p.derivative()
(x^562949953421312 + 1, 562949953421312*x^562949953421311)
```

Wie dieses Beispiel illustriert, ist die Darstellung eine Charakteristik des Polynomrings, die wir bei seiner Bildung festlegen. Das „voll besetzte“ Polynom $p \in \mathbb{Q}[x]$ und das „dünn besetzte“ Polynom $p \in \mathbb{Q}[]$ haben deshalb nicht denselben[9] Vorfahren. Die voreingestellte Darstellung von Polynomen mit einer Unbestimmten ist voll besetzt. Die Option `sparse=True` von `PolynomialRing` dient zu Bildung eines dünn besetzten Polynomrings.

Ein wenig Theorie

Um die beste Lösung für schnelle Operation auf Polynomen zu finden, ist es gut, eine Vorstellung von ihrer algorithmischen Komplexität zu haben. Hier nun eine kurze Übersicht für den in der Algorithmik etwas erfahrenen Leser. Wir beschränken uns auf den Fall voll besetzter Polynome.

Additionen, Subtraktionen und andere direkte Manipulationen der Koeffizienten werden in Abhängigkeit vom Grad der betreffenden Polynome einfach in linearer Zeit bewältigt. Ihre Geschwindigkeit hängt daher in der Praxis wesentlich von der Möglichkeit des schnellen Zugriffs auf die Koeffizienten ab und damit von der Datenstruktur.

Die kritische Operation ist die Multiplikation. Die ist tatsächlich nicht nur eine arithmetische Grundoperation, sondern auch andere Operationen arbeiten mit Algorithmen, deren Komplexität entscheidend von der Komplexität der Multiplikation abhängt. Wir können zum Beispiel die euklidsche Division zweier Polynome höchstens n-ten Grades mit Kosten von $\mathcal{O}(1)$ Multiplikationen berechnen, oder auch ihren ggT mit $\mathcal{O}(\log n)$ Multiplikationen.

Die gute Nachricht: wir können Polynome in beinahe linearer Zeit multiplizieren. Genauer gesagt ist die beste bekannte Komplexität auf einem beliebigen Ring $\mathcal{O}(n \log n \log\log n)$ Operationen auf dem Basisring. Sie beruht auf Verallgemeinerungen des berühmten Algorithmus von Schönhage-Strassen, welcher die gleiche Komplexität bei der Multiplikation ganzer Zahlen erreicht. Zum Vergleich, das Verfahren, das wir für die Multiplikation von Polynomen von Hand verwenden, erfordert eine Anzahl von Operationen in der Größenordnung n^2.

Die Algorithmen für schnelle Multiplikation sind in der Praxis bei Polynomen hinreichend hohen Grades konkurrenzfähig genauso wie die daraus abgeleiteten Divisionsverfahren. Die Bibliothen, auf die sich Sage bei bestimmten Typen von Koeffizienten stützt, arbeiten mit dieser Art von ausgefeilten Algorithmen. Das ist der Grund, weshalb Sage mit Polynomen astronomischen Grades auf bestimmten Koeffizientenringen so effizient arbeiten kann.

[9] Trotzdem gibt `QQ['x'] == PolynomialRing(QQ, 'x', sparse=True)` wahr zurück: die beiden Vorfahren sind *gleich*, denn sie repräsentieren dasselbe mathematische Objekt. Natürlich gibt der entsprechende Test mit `is` falsch zurück.

Bestimmte Details der Darstellung variieren überdies je nach der Natur der Koeffizienten des Polynoms. Es kommt bei der Ausführung der grundlegenden Operationen auch auf den verwendeten Code an. Tatsächlich bietet Sage außer einer *generischen* Implementierung der Polynome, die auf jedem kommutativen Ring funktioniert, mehrere optimierte Varianten für einen speziellen Typ von Koeffizienten. Diese verfügen über zusätzliche Funktionalitäten und sind vor allem beträchtlich effizienter als die generische Version. Sie greifen dafür auf spezialisierte externe Bibliotheken zurück, im Fall von $\mathbb{Z}[x]$ beispielsweise auf FLINT oder NTL.

Bei der Ausführung vieler umfangreicher Rechnungen kommt es entscheidend darauf an, soweit wie irgend möglich auf solchen Polynomringen zu arbeiten, die effiziente Implementierungen bieten. Die für ein Polynom `p` mit `p?` angezeigte Hilfeseite zeigt, welche Implementierung vorliegt. Die Wahl der Implementierung ergibt sich meistens aus den Implementierungen von Basisring und Darstellung. Die Option `implementation` von `PolynomialRing` erlaubt die Präzisierung einer Implementierung, wenn mehrere Möglichkeiten zur Auswahl stehen.

Symbolische Ausdrücke. Die in den Kapiteln 1 und 2 beschriebenen symbolischen Ausdrücke (d.h. die Elemente von `SR`) liefern eine dritte Darstellung von Polynomen. Sie stellen eine natürliche Wahl dar, wenn eine Rechnung Polynome und komplexere Ausdrücke mischt, wie das in der Analysis häufig der Fall ist. Die Flexibilität der Darstellung aber, die sie bieten, ist zuweilen sogar in einem eher algebraischen Kontext von Nutzen. Das Polynom $(x+1)^{10^{10}}$ beispielsweise ist entwickelt voll besetzt, es ist aber nicht nötig (auch nicht wünschbar), es auszumultiplizieren, um es abzuleiten oder numerisch auszuwerten.

Doch Vorsicht: anders als algebraische Polynome sind symbolische Polynome (aus `SR`) nicht an einen bestimmten Koeffizientenring gebunden und werden nicht in kanonischer Form behandelt. Dasselbe Polynom hat eine Vielzahl verschiedener Schreibweisen, wobei es Sache des Anwenders ist, die nötigen Umwandlungen zischen ihnen anzugeben. Es entspricht dieser Denkweise, dass in der Definitionsmenge `SR` alle symbolischen Ausdrücke zusammengefasst werden, ohne einen Unterschied zwischen den Polynomen und den anderen zu machen, wir können aber mit `f.is_polynomial(x)` explizit testen, ob ein symbolischer Ausdruck `f` ein Polynom in der Variablen `x` ist.

8. Lineare Algebra

> Mathematik ist die Kunst, jedes Problem auf lineare Algebra zu reduzieren.
>
> William Stein

Dieses Kapitel behandelt die exakte und symbolische lineare Algebra, d.h. das symbolische Rechnen auf solchen Ringen wie $\mathbb{Z}$, endlichen Körpern, Polynomringen... Was die numerische lineare Algebra betrifft, so wird sie in Kapitel 13 behandelt. Die Konstruktionen auf Matrizen und ihren Räumen stellen wir so dar, dass die Basisoperationen (Abschnitt 8.1), dann die verschiedenen, auf Matrizen möglichen, Rechnungen, in zwei Gruppen zusammengefasst werden: die eine, die mit der Gauß-Methode und mit Äquivalenz-Umformungen von links zu tun hat, die andere, die mit Ähnlichkeits-Transformationen (Unterabschnitt 8.2.3) verbunden ist. In den Büchern von Gantmacher [Gan90], von zur Gathen und Gerhard [vzGG03] sowie von Abdeljaoued und Lombardi [AL03] kann man eine gründliche Behandlung der in diesem Kapitel angerissenen Themen finden.

8.1. Erstellung und elementare Operationen

8.1.1. Vektorräume, Matrix-Vektorräume

Ebenso wie Polynome werden Vektoren und Matrizen als algebraische Objekte behandelt, die zu einem Vektorraum gehören. Wenn die Koeffizienten Elemente eines Körpers $\mathbb{K}$ sind, ist das ein Vektorraum auf $\mathbb{K}$; gehören sie zu einem Ring, ist es ein freies $\mathbb{K}$-Modul.

So konstruiert man den Vektorraum $\mathcal{M}_{2,3}$ und den Vektorrarum $(\mathbb{F}_3^2)^3$ durch

```
sage: MS = MatrixSpace(ZZ,3,3); MS
Full MatrixSpace of 2 by 3 dense matrices over Integer Ring
sage: VS = VectorSpace(GF(3^2,'x'),3); VS
Vector Space of dimension 3 over Finite Field in x of size 3^2
```

Ein System von Erzeugenden für diese Vektorräume ist durch die kanonische Basis gegeben; man kann sie mit `MS.gens()` als Tupel oder mit `MS.basis()` als Liste bekommen.

```
sage: MatrixSpace(ZZ,2,3).basis()
```

$$\left[\begin{pmatrix}1&0&0\\0&0&0\end{pmatrix},\begin{pmatrix}0&1&0\\0&0&0\end{pmatrix},\begin{pmatrix}0&0&1\\0&0&0\end{pmatrix},\begin{pmatrix}0&0&0\\1&0&0\end{pmatrix},\begin{pmatrix}0&0&0\\0&1&0\end{pmatrix},\begin{pmatrix}0&0&0\\0&0&1\end{pmatrix}\right]$$

Matrixgruppen. Wir können außerdem Untergruppen des gesamten Matrix-Vektorraums definieren. So gibt der Konstruktor `MatrixGroup([A,B,...])` die Gruppe zurück, die von den als Argument übergebenen Matrizen erzeugt wird. Sie müssen invertierbar sein.

```
sage: A = matrix(GF(11), 2, 2, [1,0,0,2])
sage: B = matrix(GF(11), 2, 2, [0,1,1,0])
sage: MG = MatrixGroup([A,B])
```

Matrix-Vektorräume	
Konstruktion	`MS = MatrixSpace(K, nrows, ncols)`
von dünn besetzten M.	`MS = MatrixSpace(K, nrows, ncols), sparse=True`
Ring mit Basis $\mathbb{K}$	`MS.base_ring()`
Ringerweiterung	`MS.base_extend(B)`
Ringänderung	`MS.change_ring(B)`
aufgespannte Gruppe	`MatrixGroup([A,B])`
Basis des Vektorraums	`MS.basis()` *oder* `MS.gens()`

Erzeugung von Matrizen	
Nullmatrix	`MS()` *oder* `MS.zero()` *oder* `matrix(K,nrows,ncols)`
Matrix mit Elementen	`MS([1,2,3,4]` *oder* `matrix(K,2,2,[1,2,3,4])` *oder* `matrix(K,[[1,2] , [3,4]])`
Identitätsmatrix	`MS.one()` *oder* `MS.identity_matrix()` *oder* `identity_matrix(K,n)`
Zufallsmatrix	`MS.random_element()` *oder* `random_matrix(K,nrows,ncols)`
Jordan-Block	`jordan_block(x,n)`
Matrix durch Blöcke	`block_matrix([A,1,B,0])` *oder* `block_diagonal_matrix`

Manipulations de base	
Zugriff auf ein Element	`A[2,3]`
Zugriff auf Zeile oder Spalte	`A[-1,:], A[:,2]`
Zugriff auf Spaltenpaare	`A[:,0:8:2]`

Untermatrizen	`A[3:4,2:5], A[:,2:5], A[:4,2:5]`
	`A.matrix_from_rows( [1,3]),`
	`A.matrix_from_columns([2,5]),`
	`A.matrix_from_rows_and_columns([1,3], [2,5])`
	`A.submatrix(i,j,nrows,ncols)`

Verkettung durch Zeilen	`A.stack(B)`
Verkettung durch Spalten	`A.augment(B)`

Tabelle 8.1 - Konstruktoren von Matrizen und ihren Vektorräumen.

```
sage: MG.cardinality()
200
sage: identity_matrix(GF(11),2) in MG
True
```

Die allgemeine lineare Gruppe vom Grad n auf einem Körper $\mathbb{K}$, geschrieben $\mathrm{GL}_n(\mathbb{K})$, ist die von den invertierbaren $n \times n$-Matrizen gebildete Gruppe. In Sage konstruiert man sie natürlicherweise mit dem Befehl `GL(n,K)`. Die spezielle lineare Gruppe $\mathrm{SL}_n(\mathbb{K})$ mit Elementen aus $\mathrm{GL}_n(\mathbb{K})$ mit der Determinante 1 konstruiert man mit dem Befehl `SL(n,K)`.

8.1.2. Erzeugung von Matrizen und Vektoren

Matrizen und Vektoren können selbstverständlich als Elemente eines Vektorraums durch Übergabe der Liste ihrer Einträge als Argument erzeugt werden. Bei Matrizen werden sie *als Zeile* eingelesen:

```
MS = MatrixSpace(ZZ,2,3); A = MS([1,2,3,4,5,6]); A
```

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

Der leere Konstruktor `MS()` gibt eine Nullmatrix zurück, genau wie die Methode `MS.zero()`. Mehrere spezielle Konstruktoren ermöglichen die Erzeugung gebräuchlicher Matrizen wie `random_matrix`, `identity_matrix`, `jordan_bloc` (siehe Tabelle 8.1). Insbesondere können wir Matrizen und Vektoren mit den Konstruktoren `matrix` und `vector` erzeugen, ohne zuvor den zugehörigen Vektorraum bilden zu müssen. In der Voreinstellung wird eine Matrix auf dem Ring der ganzen Zahlen $\mathbb{Z}$ konstruiert und hat als Dimensionen 0×0.

```
sage: a = matrix(); a.parent()
Full MatrixSpace of 0 by 0 dense matrices over Integer Ring
```

Natürlich kann man für die Einträge eine Definitionsmenge wie auch die Dimensionen angeben, wenn eine Nullmatrix erzeugt wird oder sonst auch eine Liste aller Einträge als Argument.

```
sage: a = matrix(GF(8,'x'),3,4); a.parent()
Full MatrixSpace of 3 by 4 dense matrices over Finite Field in x of size 2^3
```

Der Konstruktor `matrix` akzeptiert auch Objekte als Argument, die eine natürliche Transformation in eine Matrix zulassen. Er kann so verwendet werden, um die Adjazenzmatrix eines Graphen in Gestalt einer Koeffizientenmatrix in $\mathbb{Z}$ zu bekommen.

```
sage: g = graphs.PetersenGraph()
sage: m = matrix(g); m; m.parent()

[0 1 0 0 1 1 0 0 0 0]
[1 0 1 0 0 0 1 0 0 0]
[0 1 0 1 0 0 0 1 0 0]
[0 0 1 0 1 0 0 0 1 0]
[1 0 0 1 0 0 0 0 0 1]
[1 0 0 0 0 0 0 1 1 0]
[0 1 0 0 0 0 0 0 1 1]
[0 0 1 0 0 1 0 0 0 1]
[0 0 0 1 0 1 1 0 0 0]
[0 0 0 0 1 0 1 1 0 0]
Full MatrixSpace of 10 by 10 dense matrices over Integer Ring
```

Blockmatrizen. Zur Konstruktion einer Blockmatrix aus Untermatrizen können wir die Funktion `block_matrix` nutzen.

```
sage: A = matrix([[1,2],[3,4]])
sage: block_matrix([[A,-A],[2*A,A^2]])
```

$$\left(\begin{array}{cc|cc} 1 & 2 & -1 & -2 \\ 3 & 4 & -3 & -4 \\ \hline 2 & 4 & 7 & 10 \\ 6 & 8 & 15 & 22 \end{array}\right)$$

Voreingestellt ist eine quadratische Struktur, doch kann die Anzahl der Zeilenblöcke oder der Spaltenblöcke durch die optionalen Argumente `ncols` und `nrows` festgelegt werden. Nachdem das möglich ist, kann ein Eintrag wie 0 oder 1 als Diagonalblock mit passenden Dimensionen interpretiert werden (hier als Nullmatrix und Identitätsmatrix)

```
sage: A = matrix([[1,2],[3,4]])
sage: block_matrix([1,A,0,0,-A,2], ncols=3)
```

$$\left(\begin{array}{cc|ccc|cc} 1 & 0 & 1 & 2 & 3 & 0 & 0 \\ 0 & 1 & 4 & 5 & 6 & 0 & 0 \\ \hline 0 & 0 & -1 & -2 & -3 & 2 & 0 \\ 0 & 0 & -4 & -5 & -6 & 0 & 2 \end{array}\right)$$

Im Spezialfall diagonaler Blockmatrizen übergeben wir dem Konstruktor `block_diagonal_matrix` einfach die Liste der diagonalen Blöcke.

```
sage: A = matrix([[1,2,3],[0,1,0]])
sage: block_diagonal_matrix(A, A.transpose())
```

$$\left(\begin{array}{ccc|cc} 1 & 2 & 3 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 2 & 1 \\ 0 & 0 & 0 & 3 & 0 \end{array}\right)$$

Die Blockstruktur ist nur eine Abart der Eingabe, und Sage behandelt die Matrix wie jede andere auch. Wir können diese Eingabe auch deaktivieren, indem wir dem Konstruktor das Argument `subdivide=False` hinzufügen.

8.1.3. Grundlegende Operationen und Matrizen-Arithmetik

Indizes und der Zugriff auf Einträge. Der Zugriff auf Einträge sowie auf Untermatrizen geschieht mit eckigen Klammern `A[i,j]`, wobei die Regeln von Python gelten. Die Zeilenindizes i und die Spaltenindizes j müssen für den Zugriff auf Einträge ganze Zahlen sein (wir erinnern uns, dass die Indizierung in Python mit 0 beginnt und die Intervalle schließen die untere Grenze ein, die obere aber aus). Das Intervall `" : "` ohne Grenzen entspricht der Gesamtheit der möglichen Indizes in der betrachteten Dimension. Die Schreibweise `a:b:k` ermöglicht den Zugriff auf Indizes zwischen a und $b-1$ mit der Schrittweite k. Negative Indizes sind ebenfalls gültig und ermöglichen, die Indizes vom Ende her zu durchlaufen. So entspricht `A[-2,:]` der vorletzten Zeile. Der Zugriff auf Untermatrizen ist sowohl lesend wie schreibend möglich. Eine gegebene Spalte können wir beispielsweise auf folgende Weise verändern:

```
sage: A = matrix(3,3,range(9))
sage: A[:,1] = vector([1,1,1]); A
```

$$\begin{pmatrix} 0 & 1 & 2 \\ 3 & 1 & 5 \\ 6 & 1 & 8 \end{pmatrix}$$

Der Schrittparameter k kann auch negativ sein und bezeichnet dann einen Durchlauf in absteigender Richtung.

```
sage: A[::-1], A[:,::-1], A[::2,-1]
```

$$\left(\begin{pmatrix} 6 & 1 & 8 \\ 3 & 1 & 5 \\ 0 & 1 & 2 \end{pmatrix}, \quad \begin{pmatrix} 2 & 1 & 0 \\ 5 & 1 & 3 \\ 8 & 1 & 6 \end{pmatrix}, \quad \begin{pmatrix} 2 \\ 8 \end{pmatrix} \right)$$

Isolation von Untermatrizen. Um eine Matrix unter Angabe einer Liste von Zeilen- oder Spaltenindizes zu isolieren, die nicht unbedingt benachbart sein müssen, benutzten wir die Methode `A.matrix_from_rows_and_columns`.

```
sage: A = matrix(ZZ,4,4,range(16)); A
```

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{pmatrix}$$

```
sage: A.matrix_from_rows_and_columns([0,2,3],[1,2])
```

$$\begin{pmatrix} 1 & 2 \\ 9 & 10 \\ 13 & 14 \end{pmatrix}$$

Um eine Untermatrix mit benachbarten Zeilen und Spalten zu extrahieren, können wir uns auch der Methode `submatrix(i,j,m,n)` bedienen, die eine $m \times n$-Matrix bildet, deren oberer linker Eintrag sich an Position (i, j) befindet.

Einfügen und Erweitern. Möchte man einen Matrix-Vektorraum in einen solchen gleicher Dimension, aber mit Einträgen aus einer Erweiterung des Basisringes einfügen, kann man mit der Methode `base_extend` des Matrix-Vektorraumes arbeiten. Diese Operation ist jedoch nur gültig für die Erweiterung eines Körpers oder eines Ringes. Um die Änderung eines Basisringes zu bewirken, die einem Morphismus folgt (sofern der existiert), verwendet man besser die Methode `change_ring`.

```
sage: MS = MatrixSpace(GF(3),2,3)
sage: MS.base_extend(GF(9,'x'))
Full MatrixSpace of 2 by 3 dense matrices over Finite Field in x of size 3^2
sage: MS = MatrixSpace(ZZ,2,3)
sage: MS.change_ring(GF(3))
Full MatrixSpace of 2 by 3 dense matrices over Finite Field of size 3
```

Mutabilität und Zwischenspeicherung. Die Objekte, die Matrizen repräsentieren, sind nach Voreinstellung mutabel, was heißt, dass ihre Einträge nach ihrer Erzeugung noch beliebig modifiziert werden können. Will man eine Matzrix vor Veränderung schützen, benutzt man die Funktion `A.set_immutable()`. Es ist dann weiterhin möglich, mit der Funktion `copy(A)` mutable Kopien herzustellen. Anzumerken ist, dass der Mechanismus der Zwischenspeicherung der Ergebnisse von Berechnungen (wie Rang, Determinante usw.) funktionsfähig bleibt, ob nun mutabel oder nicht.

8.1.4. Grundlegende Operationen mit Matrizen

Die arithmetischen Operationen mit Matrizen geschehen mit den üblichen Operatoren $+,-,*$ und `^`. Die Inverse einer Matrix kann sowohl `A^-1` als auch `~A` geschrieben werden. Wenn a ein Skalar ist und A eine Matrix, entspricht die Operation `a*A` der äußeren Multiplikation des Matrix-Vektorraumes. Bei den anderen Operationen, bei denen ein Skalar a anstelle einer Matrix eingegeben wird (zum Beispiel die Operation `a+A`), wird a als skalare Matrix angesehen, die aI_n entspricht, wenn $a \neq 0$ ist und die Dimensionen das erlauben. Die elementweise Multiplikation zweier Matrizen wird mit der Operation `elementwise_product` angestoßen.

8.2. Rechnungen mit Matrizen

In der linearen Algebra können Matrizen zur Darstellung von Vektorfamilien verwendet werden, von linearen Gleichungssystemen, von linearen Anwendungen und von Untervektorräumen. So wird die Berechnung einer Eigenschaft wie der Rang einer Familie, der Lösung eines Gleichungssystems, von Eigenräumen einer linearen Anwendung oder der Dimension eines Untervektorraumes zurückgeführt auf Transformationen von Matrizen, die diese Eigenschaft aufweisen.

Grundlegende Operationen	
Transponierte, Konjugierte	`A.transpose(), A.conjugate()`
äußeres Produkt	`a*A`
Summe, Produkt, k-te Potenz, Inverse	`A + B, A*B, A^k, A^-1` oder `-A`

Tab. 8.2 - Grundlegende Operationen und Arithmetik mit Matrizen

Diese Transformationen entsprechen Basiswechseln, die auf der Ebene der Matrizen als Äquivalenzumformungen erscheinen: $B = PAQ^{-1}$, wobei P und Q invertierbare Matrizen sind. Zwei Matrizen heißen äquivalent, wenn eine solche Umformung existiert, die eine in die andere umwandelt. Für diese Relation können wir auch Äquivalenzklassen bilden, und wir definieren Normalformen, um jede Äquivalenzklasse auf eindeutige Weise zu charakterisieren. Im Folgenden stellen wir die wichtigsten Rechnungen mit Matrizen vor, die in Sage verfügbar sind. Dabei betrachten wir besonders zwei Fälle dieser Transformationen:

- Die Äquivalenzumformungen von links der Form $V = UA$, die solche charakteritischen Eigenschaften für die Vektorfamilien zeigen, wie den Rang (Anzahl der linear unabhängigen Vektoren), die Determinante(Volumen des Parallelepipeds, das durch die Vektorfamilie beschrieben wird), das Rangprofil (erste Teilmenge von Vektoren, die eine Basis bilden),... Der Gauß-Algorithmus ist das zentrale Werkzeug für diese Umformungen und die reduzierte Treppennormalform (die Gauß-Jordan-Form in einem Körper oder die Hermite-Form in $\mathbb{Z}$) ist die normale Form. Darüberhinaus dienen diese Transformationen zur Lösung linearer Gleichungssysteme.

- Die Ähnlichkeitstransformationen der Form $B = UAU^{-1}$, welche die charakteritischen Eigenschaften von Matrizen zeigen, die Endomorphismen darstellen, wie Eigenwerte, Eigenräume, minimale und charakteristische Polynome,... Die Jordan-Form oder die Frobenius-Form werden je nach Definitionsmenge der Rechnung die Normalformen für diese Transformationen sein.

Die Form von Gram-Schmidt ist eine weitere Zerlegung, die auf Äquivalenzumformungen von links basiert und die eine Matrix in eine Menge von orthogonalen Vektoren umwandelt.

8.2.1. Gauß-Algorithmus und Treppennormalform

Gauß-Algorithmus und Äquivalenz von links. Der Gauß-Algorithmus ist eines der grundlegenden Verfahren in der linearen Algebra; denn er erlaubt den Zugriff auf eine Matrixdarstellung, die an das Rechnen angepasst ist wie die Lösung von Gleichungssystemen und gleichzeitig einige ihrer fundamentalen Eigenschaften zeigt wie den Rang, die Determinante, das Rangprofil usw. Die wesentlichen Operationen des Verfahrens sind die elementaren Zeilenoperationen:

- Vertauschen zweier Zeilen: $L_i \leftrightarrow L_j$,
- Addition des Vielfachen einer Zeile zu einer anderen: $L_i \leftarrow L_i + sL_j$.

Auf der Matrixebene entsprechen diese Transformationen der Multiplikation von links mit der Transpositionsmatrix $T_{i,j}$ bzw. mit der Transvektionsmatrix $C_{i,j,s}$, die gegeben sind durch

$$T_{i,j} = \begin{array}{c} \begin{matrix} & & i & & j & & \end{matrix} \\ \begin{bmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & 0 & & 1 & & \\ & & & \ddots & & & \\ & & 1 & & 0 & & \\ & & & & & \ddots & \\ & & & & & & 1 \end{bmatrix} \end{array}, \quad C_{i,j,s} = \begin{array}{c} \begin{bmatrix} 1 & & & & \\ & \ddots & & & \\ & & 1 & & s \\ & & & \ddots & \\ & & & & 1 \\ & & & & & \ddots \end{bmatrix} \\ \begin{matrix} \\ \\ i \\ \\ j \end{matrix} \end{array}.$$

Diese Matrizen haben alle eine Determinante 1 oder -1. Somit kann jede Multiplikation von links mit einem Produkt dieser Matrizen als Basiswechsel gesehen werden, der das Volumen konstant lässt und daher insbesondere die Determinante (bis auf das Vorzeichen). In Sage erfolgt die Transvektion mit der Methode `add_multiple_of_row(i,j,s)` und die Transposition mit der Methode `swap_rows(i,j)`.

Gauß-Algorithmus und Anwendungen	
Zeilen-Transvektion	`add_multiple_of_row(i,j,s)`
Spalten-Transvektion	`add_multiple_of_column(i,j,s)`
Zeilen-, Spalten-Transposition	`swap_rows(i1,i2)`, `swap_columns(j1,j2)`
Gauß-Jordan-Form, unveränderlich	`echelon_form`
Gauß-Jordan-Form, in place	`echelonize`
invariante Faktoren	`elementay_divisors`
Normalform von Smith	`smith_form`
Determinante, Rang	`det`, `rank`
Minoren der Ordnung k	`minors(k)`
Rangprofil der Spalte, der Zeile	`pivots`, `pivot_rows`
Lösung des Systems von links (${}^txA = b$)	`b/A` oder `A.solve_left(b)`
Lösung des Systems von rechts ($Ax = b$)	`A\b` oder `A.solve_right(b)`
Bild des Vektorraumes	`image`
Kern von links	`kernel` oder `left_kernel`
Kern von rechts	`right_kernel`
Kern im Basis-Ring	`integer_kernel`

Spektralzerlegung	
minimales Polynom	`minimal_polynomial` oder `minpoly`
charakteristisches Polynom	`charakteristic_polynomial` oder `charpoly`
Krylow-Iteration von links	`maxspin(v)`
Eigenwerte	`eigenvalues`
Eigenvektoren von links, von rechts	`eigenvectors_left`, `eigenvectors_right`
Eigenräume von links, von rechts	`eigenspaces_left`, `eigenspaces_right`
Diagonalisierung	`eigenmatrix_left`, `eigenmatrix_right`
Jordanblock $J_{a,k}$	`jordan_block(a,k)`

Tab. 8.3 - Rechnungen mit Matrizen

Zu einem gegebenen Spaltenvektor $x = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}$, dessen k-ter Eintrag x_k invertierbar ist, definieren wir die Gauß-Transformation als Komposition der Transvektion C_{i,k,l_i} für $i = k+1, \ldots m$ mit $l_i = -\frac{x_i}{x_k}$ (ungeachtet der Reihenfolge, in der sie wechseln). Die entsprechende Matrix ist die folgende:

$$G_{z,k} = C_{k+1,k,l_{k+1}} \times \ldots \times C_{m,k,l_m} \begin{bmatrix} 1 & & & & & \\ & \ddots & & & & \\ & & 1 & & & \\ & & l_{k+1} & \ddots & & \\ & & l_{k+2} & & \ddots & \\ & & \vdots & & & \ddots \\ & & l_m & & & 1 \end{bmatrix}^k .$$

Eine Gauß-Transformation $G_{x,k}$ bewirkt, dass die Einträge unterhalb des Pivotelements x_k verschwinden:

$$G_{x,k} = \begin{bmatrix} x_1 \\ \vdots \\ x_k \\ x_{k+1} \\ \vdots \\ x_m \end{bmatrix} = \begin{bmatrix} x_1 \\ \vdots \\ x_k \\ 0 \\ \vdots \\ 0 \end{bmatrix} .$$

Für eine Matrix $A = [a_{i,j}]$ der Dimension $m \times n$ schreitet der Pivot-Algorithmus von Gauß iterativ voran, von der linken zur rechten Spalte. Unter der Voraussetzung, dass die ersten $k-1$ Spalten bereits bearbeitet sind, was $p \leq k-1$ Pivots geliefert hat, wird die k-te Spalte wie folgt behandelt:

- Finde den ersten invertierbaren Eintrag $a_{i,k}$ der Spalte C_k auf einer Zeile $i > p$. Wir nennen ihn die Pivotstelle.
- Wird keine Pivotstelle gefunden, gehe weiter zur nächsten Spalte.
- Wende die Transposition $T_{i,p+1}$ auf die Zeilen an, um die Pivotstelle auf die Position $(p+1, k)$ zu setzen.
- Wende die Gauß-Transformation $G_{x,p+1}$ an mit x als der neuen Spalte C_k.

Dieser Algorithmus transformiert die Matrix A in eine obere Dreiecksmatrix. Genauer gesagt, wird sie in Treppennormalform sein, d.h. dass der erste von 0 verschiedene Eintrag jeder Zeile sich rechts vom ersten solchen Eintrag der Zeile darüber befindet; außerdem finden sich alle Nullzeilen unten in der Matrix. Hier ein Beispiel für den Ablauf dieses Algorithmus:

```
sage: a = matrix(GF(7),4,3,[6,2,2,5,4,4,6,4,5,5,1,3]); a
```

$$\begin{pmatrix} 6 & 2 & 2 \\ 5 & 4 & 4 \\ 5 & 1 & 3 \end{pmatrix}$$

```
sage: u = copy(identity_matrix(GF(7),4)); u[1:,0] = -a[1:,0]/a[0,0]
sage: u, u*a
```

$$\left(\begin{pmatrix} 1 & 0 & 0 & 0 \\ 5 & 1 & 0 & 0 \\ 6 & 0 & 1 & 0 \\ 5 & 0 & 0 & 1 \end{pmatrix}, \begin{pmatrix} 6 & 2 & 2 \\ 0 & 0 & 0 \\ 0 & 2 & 3 \\ 0 & 4 & 6 \end{pmatrix} \right)$$

```
sage: v = copy(identity_matrix(GF(7),4)); v.swap_rows(1,2)
sage: b = v*u*a; v, b
```

$$\left(\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \begin{pmatrix} 6 & 2 & 2 \\ 0 & 2 & 3 \\ 0 & 0 & 0 \\ 0 & 4 & 6 \end{pmatrix} \right)$$

```
sage: w = copy(identity_matrix(GF(7),4))
sage: w[2:,1] = -b[2:,1]/b[1,1]; w, w*b
```

$$\left(\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 5 & 0 & 1 \end{pmatrix}, \begin{pmatrix} 6 & 2 & 2 \\ 0 & 2 & 3 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \right)$$

Der Gauß-Jordan-Algorithmus. Die Transformation von Gauß-Jordan ähnelt dem Gauß-Algorithmus, es wird $G_{x,k}$ die Transvektion hinzugefügt, die der Zeile mit dem Index $i < k$ entspricht. Das läuft darauf hinaus, dass die Einträge einer Spalte unterhalb und oberhalb des Pivots eliminiert werden. Teilt man außerdem jede Zeile durch ihren Pivot, erhält man eine Treppennormalform, die *reduziert* oder Gauß-Jordan-Form genannt wird. Für jede Äquivalenzklasse von Matrizen existiert eine eindeutige Matrix in dieser Gestalt; es handelt sich dabei also um eine Normalform.

Definition. Eine Matrix befindet sich in reduzierter Treppennormalform, wenn:

- sich alle Nullzeilen unten in der Matrix befinden,
- der am weitesten links befindliche von 0 verschiedene Eintrag, der Pivot, eine 1 ist und rechts des Pivots der Zeile darüber liegt,
- die Pivots die einzigen von 0 verschiedenen Einträge innerhalb ihrer Spalte sind.

Satz. Für jede Matrix A der Dimension $m \times n$ mit Einträgen aus einem Körper existiert eine eindeutige Matrix R der Dimension $m \times n$ in reduzierter Treppennormalform und eine invertierbare Matrix U der Dimension $m \times n$, sodass $UA = R$ gilt. Es handelt sich hierbei um die Gauß-Jordan-Zerlegung.

In Sage erhält man die reduzierte Treppennormalform mit den Methoden `echelonize` und `echelon_form`. Die erstere ersetzt die ursprüngliche Matrix durch ihre reduzierte Treppennormalform, währen die zweite eine unveränderliche Matrix zurückgibt ohne die ursprüngliche zu verändern.

```
sage: A = matrix(GF(7),4,5,[4,4,0,2,4,5,1,6,5,4,1,1,0,1,0,5,1,6,6,2])
sage: A, A.echelon_form()
```

$$\left(\begin{pmatrix} 4 & 4 & 0 & 2 & 4 \\ 5 & 1 & 6 & 5 & 4 \\ 1 & 1 & 0 & 1 & 0 \\ 5 & 1 & 6 & 6 & 2 \end{pmatrix}, \begin{pmatrix} 1 & 0 & 5 & 0 & 3 \\ 0 & 1 & 2 & 0 & 6 \\ 0 & 0 & 0 & 1 & 5 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}\right)$$

Mehrere Varianten des Gauß-Algorithmus erscheinen als verschiedene Formen der Matrizenzerlegung, die bei Rechnungen zuweilen nützlich sind: Die Zerlegung $A = LU$ für generische Matrizen, $A = LUP$ für reguläre Matrizen, $A = LQUP$ oder $A = PLUQ$ für Matrizen beliebigen Ranges. Die Matrizen L sind untere Dreiecksmatrizen (mit Nullen oberhalb der Hauptdiagonalen, auf englisch *Lower triangular*), U obere Dreiecksmatrix (*Upper triangular*), und die Matrizen P, Q sind Permutationen. Wenn diese Varianten algorithmisch auch weniger aufwendig sind als die reduzierte Treppennormalform, bieten sie doch nicht den Vorteil, eine Normalform zu liefern.

Treppennormalform auf euklidschen Ringen. Auf einem euklidschen Ring sind die von 0 verschiedenen Einträge nicht unbedint invertierbar, und der Gauß-Algorithmus beruht daher auf der Auswahl des ersten invertierbaren Eintrags in der aktuellen Spalte als Pivot. So können bestimmte Spalten, obwohl ein von 0 verschiedener Eintrag vorhanden ist, keinen Pivot enthalten, und das Verfahren ist nicht durchführbar.

Es ist demgegenüber immer möglich, eine unimodulare Transformation zu definieren, die mit Hilfe des erweiterten euklischen Algorithmus den Eintrag am Anfang einer Zeile mit demjenigen einer anderen eliminiert.

Sei $A = \begin{bmatrix} a & * \\ b & * \end{bmatrix}$ und sei $g = \operatorname{pgcd}(a, b)$. Seien u und v die vom erweiterten, auf a und b angewandten euklidschen Algorithmus ermittelten Bézout-Koeffizienten (sodass $g = ua + vb$ ist), und $s = -b/g$, $t = a/g$ sodass

$$\begin{bmatrix} u & v \\ s & t \end{bmatrix} \begin{bmatrix} a & * \\ b & * \end{bmatrix} = \begin{bmatrix} g & * \\ 0 & * \end{bmatrix}.$$

Diese Transformation ist unimodular, denn es gilt $\det\left(\begin{bmatrix} v & v \\ s & t \end{bmatrix}\right) = 1$.

Außerdem kann man wie bei Gauß-Jordan den Zeilen darüber immer Vielfache der Pivotzeile hinzufügen, um ihre Einträge in derselben Spalte modulo des Pivotelements g zu reduzieren. Wird diese Operation nach und nach auf alle Spalten der Matrix angewendet, ergibt das die hermitesche Normalform.

Definition. Eine Matrix heißt hermitesch, wenn

- sich die Nullzeilen unten befinden,
- die am weitesten links stehenden und von 0 verschiedenen Einträge jeder Zeile, die Pivotelemente heißen, sich rechts vom Pivotelement in der Zeile darüber befinden,
- alle Einträge oberhalb des Pivots reduziert sind modulo des Pivots.

Satz. Zu jeder Matrix A der Dimension $m \times n$ mit Koeffizienten aus einem euklidschen Ring existieren eine eindeutige hermitesche Matrix H der Dimension $m \times n$ und eine unimodulare Matrix U der Dimension $m \times m$ sodass $UA = H$ gilt.

Bei einem Körpers entspricht die hermitesche Form der Zeilentreppenform oder der Form von Gauß-Jordan. In diesem Fall sind alle Pivotelemente tatsächlich invertierbar, jede Zeile kann durch ihr Pivotelement dividiert werden, und die Einträge darüber können wieder durch eine Gauß-Transformation eliminiert werden, wodurch eine reduzierte Treppennormalforn entsteht. In Sage gibt es dafür eine einzige Methode: `echelon_form`, die entweder die hermitesche Form oder die reduzierte Treppennormalform zurückgibt, je nachdem, ob die Einträge der Matrix aus einem Ring oder einem Körper kommen.

Für eine Matrix mit Koeffizienten aus $\mathbb{Z}$ beispielsweise erhalten wir zwei verschiedene Treppennormalformen, je nachdem ob die Basismenge $\mathbb{Z}$ oder $\mathbb{Q}$ ist:

```
sage: a = matrix(ZZ, 4, 6, [2,1,2,2,2,-1,1,2,-1,2,1,-1,2,1,-1,\
-1,2,2,2,1,1,-1,-1,-1]); a.echelon_form()
```

$$\begin{pmatrix} 1 & 2 & 0 & 5 & 4 & -1 \\ 0 & 3 & 0 & 2 & -6 & -7 \\ 0 & 0 & 1 & 3 & 3 & 0 \\ 0 & 0 & 0 & 6 & 9 & 3 \end{pmatrix}$$

```
sage: a.base_extend(QQ).echelon_form()
```

$$\begin{pmatrix} 1 & 0 & 0 & 0 & \frac{5}{2} & \frac{11}{6} \\ 0 & 1 & 0 & 0 & -3 & -\frac{8}{3} \\ 0 & 0 & 1 & 0 & -\frac{3}{2} & -\frac{3}{2} \\ 0 & 0 & 0 & 1 & \frac{3}{2} & \frac{1}{2} \end{pmatrix}$$

Für Matrizen auf $\mathbb{Z}$ kann die hermitesche Normalform auch mit `hermite_form` berechnet werden. Um die Übergangsmatrix U zu erhalten, sodass $UA = H$ wird, können wir die Option `transformation=True` wählen.

```
sage: A = matrix(ZZ,4,5,[4,4,0,2,4,5,1,6,5,4,1,1,0,1,0,5,1,6,6,2])
sage: H, U = A.echelon_form(transformation=True); H, U
```

$$\left(\begin{pmatrix} 1 & 1 & 0 & 0 & 2 \\ 0 & 4 & -6 & 0 & -4 \\ 0 & 0 & 0 & 1 & -2 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 1 & 1 & -1 \\ 0 & -1 & 5 & 0 \\ 0 & -1 & 0 & 1 \\ 1 & -2 & -4 & 2 \end{pmatrix}\right)$$

Invariante Faktoren und die Normalform von Smith. Wenn wir die hermitesche Form mit weiteren unimodularen Transformationen von rechts (d.h. mit den Spalten) umformen, erhalten wir eine kanonische Diagonalmatrix, die Normalform von Smith. Ihre Einträge auf der Diagonalen sind die *invarianten Faktoren* (engl. *elementary divisors*) der Matrix. Sie sind nach Teilbarkeit total geordnet (d.h. $s_i \mid s_{i+1}$).

Satz. Zu jeder Matrix A der Dimension $m \times n$ mit Einträgen aus einem Hauptidealring existieren unimodulare Matrizen U und V der Dimension $m \times m$ bzw. $n \times n$ und eine $m \times n$-Diagonalmatrix S, sodass $S = UAV$ ist. Die Einträge $s_i = S_{i,i}$ erfüllen für $i \in \{1, \ldots, \min(m, n)\}$ die Beziehung $s_i \mid s_{i+1}$ und heißen invariante Faktoren von A.

In Sage gibt die Methode `elementary_divisors` die Liste der invarianten Faktoren zurück.Wir können außerdem die Normaform von Smith berechnen sowie mit dem Befehl `smith_form` die Übergangsmatrizen U und V.

```
sage: A = matrix(ZZ, 4, 5, [-1,-1,-1,-2,-2,-2,1,1,-1,2,2,2,2,2,-1,2,2,2,2,2])
sage: S,U,V = A.smith_form(); S,U,V
```

$$\left(\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 6 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -2 & 1 & 0 & 0 \\ 0 & 0 & -2 & -1 \end{pmatrix}, \begin{pmatrix} 0 & -2 & -1 & -5 & 0 \\ 1 & 0 & 1 & -1 & -1 \\ 0 & 0 & 0 & 0 & 1 \\ -1 & 2 & 0 & 5 & 0 \\ 0 & -1 & 0 & -2 & 0 \end{pmatrix}\right)$$

```
sage: A.elementary_divisors()
[1, 1, 3, 6]
sage: S == U*A*V
True
```

Rang, Rangprofil und Pivotelemente. Der Gauß-Algorithmus offenbart zahlreiche Invarianten der Matrix wie ihren Rang und ihre Determinante (die als Produkt der Pivotstellen gelesen werden kann). Sie sind mit den Funktionen `det` und `rank` zugänglich. Diese Werte werden intern gespeichert und werden deshalb bei einem erneuten Aufruf nicht nochmals berechnet.

Allgemeiner ist der Begriff des Rangprofils sehr nützlich, wenn die Matrix als Folge von Vektoren betrachtet wird.

Definition. Das Rangprofil nach Spalten einer $m \times n$-Matrix mit dem Rang r ist die Folge der r lexikographisch minimalen Indizes, sodass die entsprechenden Spalten von A linear unabhängig sind.

Das Rangprofil wird an der reduzierten Treppennormalform als Folge der Indizes der Pivotstellen direkt abgelesen. Es wird von der Funktion `pivots` berechnet. Wenn die reduzierte Treppennormalform bereits berechnet ist, ist auch das Rangprofil schon gespeichert und kann ohne zusätzliche Rechnung erhalten werden.

Das Rangprofil nach Zeilen ist auf ähnliche Weise definiert, indem die Matrix als Folge von m Zeilenvektoren angesehen wird. Wir bekommen es mit dem Befehl `pivot_rows` oder als Unterprodukt der reduzierten Treppennormalform der Transponierten Matrix.

```
sage: B = matrix(GF(7),5,4,[4,5,1,5,4,1,1,1,0,6,0,6,2,5,1,6,4,4,0,2])
sage: B.transpose().echelon_form()
```

$$\begin{pmatrix} 1 & 0 & 5 & 0 & 3 \\ 0 & 1 & 2 & 0 & 6 \\ 0 & 0 & 0 & 1 & 5 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

```
sage: B.pivot_rows()
(0, 1, 3)
sage: B.transpose().pivots() == B.pivot_rows()
True
```

8.2.2. Lösung von Gleichungssystemen; Bild und Basis des Kerns

Lösung von Gleichungssystemen. Ein lineares Gleichungssystem kann durch eine Matrix A dargestellt werden und einen Vektor b, sei es von rechts: $Ax = b$, sei es von links: ${}^t xA = b$. Die Funktionen `solve_right` und `solve_left` bewirken die Lösung. Auch kann man, gleichwertig dazu, die Operatoren $A\backslash b$ und b/A benutzen. Sobald das System durch eine Koeffizientenmatrix auf einem Ring gegeben ist, wird die Lösung systematisch auf dem Körper der Brüche dieses Ringes berechnet (z.B. $\mathbb{Q}$ für $\mathbb{Z}$ oder $K(X)$) für $K[X]$). im weiteren wird man sehen, wie die Lösung im Ring selbst berechnet wird. Das rechte Glied in der Gleichung des Systems kann sowohl ein Vektor sein als auch eine Matrix (was der gleichzeitigen Lösung mehrerer linearer Systeme mit gleicher Matrix entspricht).

Die Matrizen der Systeme können rechteckig sein, und die System können keine, genau eine oder unendlich viele Lösungen besitzen. In diesem letzteren Fall gibt die Funktion `solve` eine dieser Lösungen zurück, indem sie die Bestandteile, die linear abhängigen Spalten des Systems entsprechen, zu null setzt.

```
sage: R.<x> = PolynomialRing(GF(5),'x')
sage: A = random_matrix(2,3); A
```

$$\begin{pmatrix} 4x^2+1 & 2x^2+4x+1 & 4x^2+3 \\ 3x^2+3x & 3x+4 & 2x^2+x+1 \end{pmatrix}$$

```
sage: b = random_matrix(R,2,1); b
```

$$\begin{pmatrix} 2x+1 \\ 2x^2+x \end{pmatrix}$$

```
sage: A.solve_right(b)
```

$$\begin{pmatrix} \frac{4x^2+3x+3}{x^2+3x+3} \\ \frac{2x^2+x}{x^2+3x+3} \\ 0 \end{pmatrix}$$

```
sage: A.solve_right(b) == A\b
True
```

Bild und Kern. Als lineare Abbildung Φ interpretiert, definiert eine $m \times n$-Matrix A zwei Untervektorräume K^m und K^n bzw. Bild und Kern von Φ.

Das Bild ist die Menge der Vektoren von K^m, die durch Linearkombination der Spalten von A erhalten wird, Man bekommt es mit der Funktion `image`, die einen Vektorraum mit der Basis in Treppennormalform zurückgibt.

Der Kern ist der Untervektorraum K^n von Vektoren x, sodass $Ax = 0$ ist. Die Berechnung einer Basis dieses Untervektorraums dient insbesondere zur Beschreibung der Lösungsmenge eines linearen Systems, wenn es unendlich viele Lösungen besitzt: wenn $\bar{x}$ eine Lösung von $Ax = b$ ist und V der Kern von A, dann wird die Lösungsmenge einfach $\bar{x} + V$ geschrieben. Wir erhalten ihn mit der Funktion `right_kernel`, die eine Beschreibung des Vektorraums zusammen mit einer Basis in Gestalt der reduzierten Treppennormalform zurückgibt. Natürlich können wir auch den Kern von links definieren (die Menge der x in K^m mit ${}^t xA = 0$), welcher dem Kern von rechts der Transponierten von A entpricht (d.h. der Adjungierten von Φ). Den erhalten wir mit der Funktion `left_kernel`. Es ist festgelegt, dass die Funktion `kernel` den Kern von links zurückgibt. Außerdem werden die Basen in beiden Fällen als Matrizen von Zeilenvektoren zurückgegeben.

```
sage: a = matrix(QQ,3,5,[2,2,-1,-2,-1,2,-1,1,2,-1/2,2,-2,-1,2,-1/2])
sage: a.image()
Vector space of degree 5 and dimension 3 over Rational Field
Basis matrix:
[     1      0      0    1/4 -11/32]
[     0      1      0     -1   -1/8]
[     0      0      1    1/2   1/16]
sage: a.right_kernel()
Vector space of degree 5 and dimension 2 over Rational Field
Basis matrix:
[    1     0     0  -1/3   8/3]
[    0     1  -1/2 11/12   2/3]
```

Der Begriff des Kerns wird für den Fall verallgemeinert, dass die Koeffizienten keine Körperelemente mehr sind; dann handelt es sich um ein freies Modul. Ist eine Matrix insbesondere über einem Körper von Brüchen definiert, werden wir den Kern mit dem Befehl `integer_kernel` in einem Basisring erhalten. Zu einer Matrix mit Koeffizienten aus $\mathbb{Z}$ beispielsweise, die in den Vektorraum der Matrizen mit Koeffizienten aus $\mathbb{Q}$ eingebettet ist, können wir den Kern sehr wohl auch als Untervektorraum von $\mathbb{Q}^m$ oder als freien Modul von $\mathbb{Z}^m$ berechnen.

```
sage: a = matrix(ZZ,5,3,[1,1,122,-1,-2,1,-188,2,1,1,-10,1,-1,-1,-1])
sage: a.kernel()
Free module of degree 5 and rank 2 over Integer Ring
```

```
Echelon basis matrix:
[   1  979  -11 -279  811]
[   0 2079  -22 -569 1488]
sage: b = a.base_extend(QQ)
sage: b.kernel()
Vector space of degree 5 and dimension 2 over Rational Field
Basis matrix:
[        1         0  -121/189 -2090/189   6949/63]
[        0         1    -2/189 -569/2079   496/693]
sage: b.integer_kernel()
Free module of degree 5 and rank 2 over Integer Ring
Echelon basis matrix:
[   1  979  -11 -279  811]
[   0 2079  -22 -569 1488]
```

8.2.3. Eigenwerte, Jordanform und Ähnlichkeitstransformationen

Wenn wir eine quadratische Matrix als linearen Operator interpretieren (ein Endomorphismus), ist sie nur die Darstellung zu einer gegebenen Basis. Jeder Basiswechsel entspricht einer Ähnlichkeitstransformation $B = U^{-1}AU$ der Matrix. Die beiden Matrizen A und B heißen deshalb *ähnlich*. So werden die Eigenschaften des linearen Operators, die von der Basis unabhängig sind, durch die Untersuchung der Ähnlichkeitsinvarianten der Matrix gefunden.

Unter diesen Invarianten sind die einfachsten der Rang und die Determinante. Tatsächlich ist der Rang von $U^{-1}AU$, wobei U^{-1} und U invertierbare Matrizen sind, gleich dem Rang von A. Außerdem ist $\det(U^{-1}AU) = \det(U^{-1})\det(A)\det(U) = \det(U^{-1}U)\det(A) = \det(A)$. Ebenso ist das charakteristische Polynom der Matrix A, das durch $\chi_A(x) = \det(x\mathrm{Id} - A)$ definiert ist, bei Ähnlichkeitstransformation invariant:

$$\det(x\mathrm{Id} - U^{-1}AU) = \det(U^{-1}(x\mathrm{Id} - A)U) = \det(x\mathrm{Id} - A).$$

Als Konsequenz sind die charakteristischen Werte einer Matrix, die als Wurzeln des charakteristischen Polynoms in ihrem Zerlegungskörper definiert sind, ebenfalls Ähnlichkeitsinvarianten. Definitionsgemäß ist ein Skalar λ ein Eigenwert einer Matrix A, wenn ein Vektor u, der kein Nullvektor ist, existiert, sodass $Au = \lambda u$ ist. Der einem Eigenwert λ zugeordnete Eigenraum ist die Menge der Vektoren u, sodass $Au = \lambda u$ ist. Das ist ein durch $E_\lambda = \mathrm{Ker}(\lambda\mathrm{Id} - A)$ definierter Untervektorraum.

Die Eigenwerte fallen mit den charakteristischen Werten zusammen:

$$\det(\lambda\mathrm{Id} - A) = 0 \Leftrightarrow \dim(\mathrm{Ker}(\lambda\mathrm{Id} - A)) \geq 1 \Leftrightarrow \exists u \neq 0,\ \lambda u - Au = 0.$$

Diese beiden Sichtweisen entsprechen jeweils dem algebraischen bzw. dem geometrischen Ansatz der Eigenwerte. Bei der geometrischen Sichtweise interessieren wir uns für die Wirkung des linearen Operators A auf die Vektoren des Vektorraums mit größerer Genauigkeit als bei der algebraischen Sichtweise. Insbesondere unterscheiden wir die Begriffe der algebraischen Vielfachheit, die der Ordnung der Wurzeln des charakteristischen Polynoms entspricht, von der geometrischen Vielfachheit, die der Dimension des dem Eigenwert zugeordneten Eigenuntervektorraums entspricht. Bei diagonalisierbaren Matrizen sind beide Begriffe äquivalent. Andernfalls ist die geometrische Vielfachheit immer kleiner als die algebraische.

Die geometrische Sichtweise erlaubt eine weitergehende Beschreibung der Struktur der Matrix. Außerdem gibt es viel schnellere Algorithmen für die Berechnung der Eigenwerte und -räume sowie der charakteristischen und minimalen Polynome.

Zyklische invariante Vektorräume und Frobenius-Normalform. Sei A eine $n \times n$-Matrix auf einem Körper $\mathbb{K}$ und u ein Vektor aus $\mathbb{K}^n$. Die Familie der Vektoren u, Au, A^2u, ..., A^nu heißt Krylow-Folge und ist damit verbunden (als Familie von $n+1$ Vektoren der Dimension n). Sei d eine Zahl, sodass A^du der erste Vektor der linear abhängigen Folge mit den Vorgängern $u, Au, \ldots, A^{d-1}u$ ist. Wir schreiben diese Relation der linearen Abhängigkeit dann

$$A^du = \sum_{i=0}^{d-1} \alpha_i A^i u.$$

Das Polynom $\varphi_{A,u}(x) = x^d - \sum_{i=0}^{d-1} \alpha_i x^i$, das die Gleichung $\varphi_{A,u}(A)u = 0$ erfüllt, ist somit ein unitäres Annihilator-Polynom der Krylow-Folge und von minimalem Grad. Wir nennen es das *Minimalpolynom* des Vektors u (bezüglich der Matrix A). Die Menge der Annihilator-Polynome von u bildet ein von $\varphi_{A,u}$ erzeugtes Ideal von $\mathbb{K}[X]$

Das Minimalpolynom der Matrix A ist als unitäres Polynom $\varphi_A(x)$ kleinsten Grades definiert, das die Matrix A annulliert: $\varphi_A(A) = 0$. Wenn insbesondere $\varphi_A(A)$ auf den Vektor u angewendet wird, stellen wir fest, dass φ_A ein Annihilator der Krylow-Folge ist. Es ist daher notwendigerweise ein Vielfaches des Minimalpolynoms von u. Überdie können wir zeigen (siehe Übung 30), dass ein Vektor $\bar{u}$ existiert, sodass

$$\varphi_{A,\bar{u}} = \varphi_A. \tag{8.1}$$

Wird der Vektor u zufällig ausgewählt, ist die Wahrscheinlichkeit, dass er Gl. (8.1) genügt, umso größer, je größer der Körper ist (man kann zeigen, dass sie mindesten $1 - \frac{n}{|K|}$ beträgt).

Übung 30. Wir möchten zeigen, dass immer ein Vektor $\bar{u}$ existiert, dessen Minimalpolynom mit dem Minimalpolynom der Matrix zusammenfällt.

1. Sei $(e_1, \ldots, e_n)$ eine Basis des Vektorraums. Zeigen Sie, dass φ_A mit dem kgV der φ_{A,e_i} zusammenfällt.

2. Zeigen Sie für den Fall, dass φ_A eine Potenz eines irreduziblen Polynoms ist, dass ein Index i_0 existiert, sodass $\varphi_A = \varphi_{A,e_{i_0}}$ wird.

3. Zeigen Sie, dass wenn die Minimalpolynome $\varphi_i = \varphi_{A,e_i}$ und φ_{A,e_j} teilerfremd sind, dann $\varphi_{A,e_i+e_j} = \varphi_i\varphi_j$ gilt.

4. Zeigen Sie, dass wenn $\varphi_A = P_1P_2$ mit teilerfremden P_1 und P_2 gilt, dann Vektoren $x_1 \neq 0$ und $x_2 \neq 0$ existieren, sodass P_i das Minimalpolynom von x_i ist.

5. Folgern Sie mit Faktorisierung in irreduzible Polynome, dass $\varphi_A = \varphi_1^{m_1} \cdot \ldots \cdot \varphi_k^{m_k}$ gilt.

6. Veranschaulichung: sei $A = \begin{bmatrix} 0 & 0 & 3 & 0 & 0 \\ 1 & 0 & 6 & 0 & 0 \\ 0 & 1 & 5 & 0 & 0 \\ 0 & 0 & 0 & 0 & 5 \\ 0 & 0 & 0 & 1 & 5 \end{bmatrix}$ eine Matrix aus GF(7). Berechnen Sie die Grade der Minimalpolynome der Vektoren der kanonischen Basis $u = e_1$ und $v = e_4$

sowie von $u+v$. Wir können uns der Funktion `maxspin(u)` bedienen, die bei Anwendung auf die Transponierte von A die maximale Folge der Iterierten von Krylow eines Vektors u zurückgibt.

Sei $P = x^k + \sum_{i=0}^{k-1}$ ein unitäres Polynom k-ten Grades. Die dem Polynom P zugeordnete Begleitmatrix ist die $k \times k$-Matrix, die definiert ist durch

$$C_p = \begin{bmatrix} 0 & & & -\alpha_0 \\ 1 & & & -\alpha_1 \\ & \ddots & & \vdots \\ & & 1 & -\alpha_{k-1} \end{bmatrix}.$$

Diese Matrix hat die Eigenschaft, P als minimales und charakteristisches Polynom zu haben. Sie spielt somit eine große Rolle bei der Berechnung von minimalen und charakteristischen Polynomen.

Proposition. Sei K_u die von den ersten d Krylow-Iterierten eines Vektors u gebildete Matrix. Dann gilt

$$AK_u = K_u C_{\varphi_{A,u}}$$

Wenn $d = n$ ist, ist die Matrix K_u somit quadratisch der Ordnung n und invertierbar. Sie definiert eine Ähnlichkeitstransformation $K_u^{-1} A K_u = K_u C_{\varphi_{A,u}}$, welche die Matrix A zu einer Begleitmatrix reduziert. Nun bewahrt diese Transformation die Determinante und somit auch das charakteristische Polynom; wir können deshalb die Koeffizienten des minimalen und charakteristischen Polynoms (die hier identisch sind) an der Begleitmatrix direkt ablesen.

```
sage: A = matrix(GF(97), 4, 4, [86,1,6,68,34,24,8,35,15,36,68,42,27,1,78,26])
sage: e1 = identity_matrix(GF(97),4)[0]
sage: U = matrix(A.transpose().maxspin(e1)).transpose()
sage: F = U^-1*A*U; F
```

$$\begin{pmatrix} 0 & 0 & 0 & 83 \\ 1 & 0 & 0 & 77 \\ 0 & 1 & 0 & 20 \\ 0 & 0 & 1 & 10 \end{pmatrix}$$

```
sage: K.<x> = GF(97)[]
sage: P = x^4-sum(F[i,3]*x^i for i in range(4)); P
```

$x^4 + 87x^3 + 77x^2 + 20x + 14$

```
sage: P == A.charpoly()
True
```

Im allgemeinen Fall ($d \leq n$) bilden die iterierten Vektoren $u, \ldots, A^{d-1}u$ eine Basis eines Untervektorraums I, der gegen Einwirkung der Matrix A invariant ist (d.h. $AI \subseteq I$). Da wir jeden dieser Vektoren zyklisch durch Anwendung der Matrix A auf den vorherigen Vektor erhalten, nennen wir ihn auch zyklischen Untervektorraum. Die maximale Dimension eines solchen Untervektorraums ist der Grad des Minimalpolynoms der Matrix. Es wird von den Krylow-Iterierten des in Übung 30 gebildeten Vektors erzeugt, den wir mit u_1^* bezeichnen. Wir nennen ihn den ersten invarianten Untervektorraum. Dieser erste Vektorraum lässt einen komplementären Vektorraum V zu. Bei Berechnung modulo des ersten invarianten Vektorraums, d.h. wenn zwei Vektoren als gleich gelten, wenn ihre Differenz zum ersten invarianten Untervektorraum gehört, können wir zu den Vektoren in diesem komplementären Vektorraum

einen zweiten invarianten Untervektorraum definieren wie auch ein Minimalpolynom, das die zweite Ähnlichkeitsinvariante genannt wird. Wir erhalten dann eine Beziehung der Form:

$$A[K_{u_1^*} \;\; K_{u_2^*}] = [K_{u_1^*} \;\; K_{u_2^*}] \begin{bmatrix} C_{\varphi_1} & \\ & C_{\varphi_2} \end{bmatrix},$$

worin φ_1, φ_2 die beiden ersten Ähnlichkeitsinvarianten sind und $K_{u_1^*}, K_{u_2^*}$ die Krylow-Matrizen, die den beiden von den Vektoren u_1^* und u_2^* aufgespannten zyklischen Vektorräumen entsprechen.

Wir bilden iterativ eine quadratische, invertierbare Matrix $K = [K_{u_1^*} \;\; \dots \;\; K_{u_k^*}]$, so dass

$$K^{-1}AK = \begin{bmatrix} C_{\varphi_1} & & \\ & \ddots & \\ & & C_{\varphi_k} \end{bmatrix}. \tag{8.2}$$

Da jedes u_i^* durch die φ_j mit $j \leq i$ annulliert wird, ersehen wir daraus, dass $\varphi_i \mid \varphi_{i-1}$ für alle $2 \leq i \leq k$, anders gesagt, die Folge der φ_i ist bezüglich der Division total geordnet. Wir können beweisen, dass zu jeder Matrix eine eindeutige Folge von invarianten Polynomen $\varphi_1, \dots, \varphi_k$ existiert. So ist die diagonale Blockmatrix $\mathrm{Diag}(C_{\varphi_1}, \dots, C_{\varphi_k})$, zur Matrix A ähnlich, und diese Polynome zeigt eine Normalform, die rational kanonische Form genannt wird oder Frobenius-Normalform.

Satz (Frobenius-Normalform). Jede quadratische Matrix A auf einem Körper ist zu einer eindeutigen Matrix $F = \begin{bmatrix} C_{\varphi_1} & & \\ & \ddots & \\ & & C_{\varphi_k} \end{bmatrix}$ mit $\varphi_{i+1} \mid \varphi_i$ für jedes $i < k$ ähnlich.

Gemäß Gl. (8.2) können wir anscheinend die Basen der invarianten Untervektorräume an der Übergangsmatrix K ablesen.

Bemerkung. Der Satz von Caley-Hamilton besagt, dass das charakteristische Polynom seine Matrix annulliert: $\chi_A(A) = 0$. Das zu beweisen ist nach Einführung der Frobenius-Normalform einfach. Es gilt

$$\begin{aligned} \chi_A(x) &= \det(x\mathrm{Id} - A) = \det(K)\det(x\mathrm{Id} - F)\det(K^{-1}) \\ &= \prod_{i=1}^{k} \det(x\mathrm{Id} - C_{\varphi_i}) = \prod_{i=1}^{k} \varphi_i(x). \end{aligned}$$

Somit ist das Minimalpolynom φ_1 ein Teiler des charakteristischen Polynoms, das deshalb Annihilator der Matrix A ist.

In Sage können wir die Frobenius-Normalform in $\mathbb{Q}$ mit Koeffizienten aus $\mathbb{Z}$ mit der Methode `frobenius` berechnen[1]:

```
sage: A = matrix(ZZ,8,[[6,0,-2,4,0,0,0,-2],[14,-1,0,6,0,-1,-1,1],\
....:                  [2,2,0,1,0,0,1,0],[-12,0,5,-8,0,0,0,4],\
....:                  [0,4,0,0,0,0,4,0],[0,0,0,0,1,0,0,0],\
....:                  [-14,2,0,-6,0,2,2,-1],[-4,0,2,-4,0,0,0,4]])
```

[1] Das ist eine leichte Abweichung der aktuellen Schnittstelle des Programms: obwohl die Frobenius-Form für jede Matrix auf einem Körper definiert ist, erlaubt Sage die Berechnung nur für Matrizen auf $\mathbb{Z}$ und bewirkt implizit die Einbettung in $\mathbb{Q}$.

```
sage: A.frobenius()
```
$$\begin{pmatrix} 0 & 0 & 0 & 4 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 4 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 \end{pmatrix}$$

Außerdem können wir die Liste der invarianten Polynome erhalten, indem wir als Argument 1 übergeben. Um Informationen zu den zugeordneten invarianten Vektorräumen zu erhalten, übergeben wir das Argument 2, was die Übergangsmatrix K erzeugt. Sie liefert eine Basis des gesamten Vektorraums, der in die direkte Summe der invarianten Vektorräume zerlegt wird:

```
sage: A.frobenius(1)
```
$$\left[x^4 - x^2 - 4x - 4, x^3 - x^2 - 4, x - 2\right]$$

```
sage: F,K = A.frobenius(2)
sage: K
```
$$\begin{pmatrix} 1 & -\frac{15}{56} & \frac{17}{224} & \frac{15}{56} & -\frac{17}{896} & 0 & -\frac{15}{112} & \frac{17}{64} \\ 0 & \frac{29}{224} & -\frac{13}{224} & -\frac{23}{448} & -\frac{17}{896} & -\frac{17}{896} & \frac{29}{448} & \frac{13}{128} \\ 0 & -\frac{75}{896} & \frac{75}{896} & -\frac{47}{896} & 0 & -\frac{17}{896} & -\frac{23}{448} & \frac{11}{128} \\ 0 & \frac{17}{896} & -\frac{29}{896} & \frac{15}{896} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & -\frac{4}{21} & -\frac{4}{21} & -\frac{10}{21} & 0 & 0 & -\frac{2}{21} & 1 \end{pmatrix}$$

```
sage: K^-1*F*K == A
True
```

Diese Resultate geben zu verstehen, dass die Matrix A mit Koeffizienten aus $\mathbb{Z}$ in den Körper $\mathbb{Q}$ eingebettet wurde. Um das Verhalten der Matrix A auf dem freien Modul $\mathbb{Z}^n$ und die Zerlegung des Moduls, das sie erzeugt, zu studieren, benutzen wir die Funktion `decomposition`; allerdings sprengt deren Studium den Rahmen dieses Buches.

Invariante Faktoren und Ähnlichkeitsinvarianten. Eine wichtige Eigenschaft verbindet die Ähnlichkeitsinvarianten mit den invarianten Faktoren, die wir in Unterabschnit 8.2.1 gesehen haben.

Satz. Die Ähnlichkeitsinvarianten einer Matrix A mit Koeffizienten aus einem Körper entsprechen den invarianten Faktoren ihrer charakteristischen Matrix $x\mathrm{Id} - A$.

Der Beweis dieses Resultats sprengt den Rahmen dieses Buches, sodass wir uns mit der Veranschaulichung des vorigen Beispiels zufrieden geben.

```
sage: S.<x> = QQ[]
sage: B = x*identity_matrix(8) - A
sage: B.elementary_divisors()
```
$$\left[1, 1, 1, 1, 1, x - 2, x^3 - x^2 - 4, x^4 - x^2 - 4x - 4\right]$$

```
sage: A.frobenius(1)
```
$[x^4 - x^2 - 4x - 4, x^3 - x^2 - 4, x - 2]$

Eigenwerte, Eigenvektoren Wenn wir das Minimalpolynom in irreduzible Faktoren zerlegen, $\varphi_1 = \psi_1^{m_1} \cdot \ldots \cdot \psi_s^{m_s}$, dann werden alle invarianten Faktoren in der Form $\varphi_i = \psi_1^{m_{i,1}} \cdot \ldots \cdot \psi_s^{m_{i,s}}$ mit den Multiplizitäten $m_{i,j} \leq m_k$ geschrieben. Wir zeigen, dass dann eine Ähnlichkeitstransformation gefunden werden kann, die jeden Begleitblock C_φ der Frobenius-Form in einen Diagonalblock $\mathrm{Diag}(C_{\psi_1^{m_{i,1}}}, \ldots, C_{\psi_s^{m_{i,s}}})$ verwandelt. Diese Variante der Frobenius-Form, die wir Zwischenform nennen, wird immer von Begleitblocks gebildet, doch diesmal entspricht jeder einer Potenz eines irreduziblen Polynoms.

$$F = \begin{bmatrix} C_{\psi_1^{m_{1,1}}} & & & & \\ & \ddots & & & \\ & & C_{\psi_s^{m_{1,s}}} & & \\ & & & C_{\psi_1^{m_{2,1}}} & \\ & & & & \ddots \\ & & & & & C_{\psi_1^{m_{k,1}}} \\ & & & & & & \ddots \end{bmatrix} \tag{8.3}$$

Sobald ein irreduzibler Faktor ψ_i den Grad 1 und die Multiplizität 1 hat, ist sein Begleitblock eine 1×1-Matrix auf der Diagonalen und entspricht somit einem Eigenwert. Wenn das Minimalpolynom reduzibel und quadratfrei ist, ist die Matrix deshalb diagonalisierbar.

Die Eigenwerte erhalten wir mit der Methode `eigenvalues`. Die ihrem Eigenwert und dessen Vielfachheit zugeordnete Liste der Eigenvektoren von rechts (bzw. von links) wird durch die Methode `eigenvectors_right` (bzw. `eigenvectors_left`) angegeben. Schließlich werden die Eigenräume wie auch deren Basis von Eigenvektoren von den Methoden `eigenspaces_right` und `eigenspaces_left` geliefert.

```
sage: A = matrix(GF(7),4,[5,5,4,3,0,3,3,4,0,1,5,4,6,0,6,3])
sage: A.eigenvalues()
[4, 1, 2, 2]
sage: A.eigenvectors_right()
[(4, [
  (1, 5, 5, 1)
  ], 1), (1, [
  (0, 1, 1, 4)
  ], 1), (2, [
  (1, 3, 0, 1),
  (0, 0, 1, 1)
  ], 2)]
sage: A.eigenspaces_right()
[
(4, Vector space of degree 4 and dimension 1 over Finite Field of size 7
User basis matrix:
[1 5 5 1]),
(1, Vector space of degree 4 and dimension 1 over Finite Field of size 7
```

```
User basis matrix:
[0 1 1 4]),
(2, Vector space of degree 4 and dimension 2 over Finite Field of size 7
User basis matrix:
[1 3 0 1]
[0 0 1 1])
]
```

Kürzer gibt die Methode `eigenmatrix_right` das aus der diagonalisierten Matrix und der Matrix ihrer Eigenvektoren von rechts gebildete Paar zurück. (Die Methode `eigenmatrix_left` tut dasselbe mit den Eigenvektoren von links).

```
sage: A.eigenmatrix_right()
```

$$\left(\begin{pmatrix} 4 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 2 \end{pmatrix}, \begin{pmatrix} 1 & 0 & 1 & 0 \\ 5 & 1 & 3 & 0 \\ 5 & 1 & 0 & 1 \\ 1 & 4 & 1 & 1 \end{pmatrix}\right)$$

Jordanform. Wenn das Minimalpolynom reduzibel ist, aber Faktoren hat, deren Vielfachheit größer ist als 1, ist die Zwischenform (8.3) nicht diagonal. Wir beweisen jetzt, dass es keine Ähnlichkeitstransformation gibt, die sie diagonal macht, die anfängliche Matrix ist also nicht diagonalisierbar. Wir können sie jedoch trigonalisieren, d.h. sie zu einer oberen Dreiecksmatrix machen, sodass ihre Eigenwerte auf der Diagonalen erscheinen. Unter den verschiedenen möglichen Dreiecksmatrizen ist die am weitesten vereinfachte die Jordan-Normalform.

Ein Jordanblock $J_{\lambda,k}$, der dem Eigenwert λ und der Ordnung k zugeordnet ist, ist die $k \times k$-Matrix $J_{\lambda,k}$, die gegeben ist durch

$$J_{\lambda,k} = \begin{bmatrix} \lambda & 1 & & \\ & \ddots & \ddots & \\ & & \lambda & 1 \\ & & & \lambda \end{bmatrix}.$$

Diese Matrix spielt eine ähnliche Rolle wie die Begleitblöcke, indem sie die Multiplizität eines Eigenwertes genauer aufzeigt. Tatsächlich ist ihr charakteristisches Polynom $\chi_{J_{\lambda,k}} = (X-\lambda)^k$. Des weiteren ist auch ihr Minimalpolynom $\varphi_{J_{\lambda,k}} = (X-\lambda)^k$ und ist notwendigerweise ein Vielfaches von $P = X - \lambda$. Nun ist die Matrix

$$P(J_{\lambda,k}) = \begin{bmatrix} 0 & 1 & & \\ & \ddots & \ddots & \\ & & 0 & 1 \\ & & & 0 \end{bmatrix}$$

nilpotent der Ordnung k, woraus $\varphi_{J_{\lambda,k}} = \chi_{J_{\lambda,k}} = (X-\lambda)^k$ folgt. Die Jordan-Normalform entspricht der Zwischenform (8.3), wobei die Begleitblöcke $\psi_j^{m_{i,j}}$ durch Jordanblöcke $J_{\lambda_j,m_{i,j}}$ ersetzt sind (wir erinnern uns, dass wenn das Minimalpolynom reduzibel ist, die ψ_j in der Form $X - \lambda_j$ geschrieben werden).

Bei reduziblem Minimalpolynom ist somit jede Matrix ähnlich einer Jordanmatrix der Form

$$J = \begin{bmatrix} \boxed{\begin{matrix} J_{\lambda_1,m_{1,1}} & & \\ & \ddots & \\ & & J_{\lambda_s,m_{1,s}} \end{matrix}} & & & \\ & \boxed{\begin{matrix} J_{\lambda_1,m_{2,1}} & \\ & \ddots \end{matrix}} & & \\ & & \ddots & \\ & & & \boxed{\begin{matrix} J_{\lambda_1,m_{k,1}} & \\ & \ddots \end{matrix}} \end{bmatrix} \tag{8.4}$$

Insbesondere ist die Jordanform auf jedem algebraisch abgeschlossenen Körper wie $\mathbb{C}$ stets definiert.

In Sage produziert der Konstruktor `jordan_block(a,k)` den Jordanblock $J_{a,k}$. Die Jordan-Normalform erhalten wir mit `jordan_form`. Die Option `transformation=True` erlaubt, die Transformationsmatrix U zu erhalten, sodass $U^{-1}AU$ in Jordanform ist.

```
sage: A = matrix(ZZ,4,[3,-1,0,-1,0,2,0,-1,1,-1,2,0,1,-1,-1,3])
sage: A.jordan_form()
```

$$\left(\begin{array}{c|c|cc} 3 & 0 & 0 & 0 \\ \hline 0 & 3 & 0 & 0 \\ \hline 0 & 0 & 2 & 1 \\ 0 & 0 & 0 & 2 \end{array}\right)$$

```
sage: J,U = A.jordan_form(transformation=True)
sage: U^-1*A*U == J
True
```

Die Jordanform ist bis auf eine Permutation der Jordanblöcke eindeutig. Gemäß der Literatur kann man vorschreiben oder auch nicht, dass die Reihenfolge ihres Auftreten auf der Diagonalen wie in Gleichung (8.4) der Reihenfolge der invarianten Polynome entspricht. Wir bemerken in obigem Beispiel, dass Sage diese Ordnung nicht respektiert, weil das erste invariante Polynom (das Minimalpolynom) das Polynom $(X-3)(X-2)^2$ ist.

Primäre Normalform. Der Vollständigkeit halber müssen wir eine letzte Normalform erwähnen, welche die Jordanform für den beliebigen Fall verallgemeinert, wo das Minimalpolynom nicht reduzibel ist. Für ein irreduzibles Polynom P k-ten Grades definieren wir den Jordanblock der Multiplizität m als die Matrix $J_{P,m}$ der Dimension $km \times km$

$$J_{P,m} = \begin{bmatrix} C_P & B & & \\ & \ddots & \ddots & \\ & & C_P & N \\ & & & C_P \end{bmatrix}.$$

wobei B die $k \times k$-Matrix ist, deren einziger von null verschiedener Eintrag $B_{k,1} = 1$ ist, und C_P ist die dem Polynom P zugeordnete Begleitmatrix (Unterabschnitt 8.2.3). Wir sehen, wie wir für $P = X - \lambda$ den dem Eigenwert λ zugeordneten Jordanblock wiederfinden. Auf ähnliche

Weise zeigen wir, dass für die minimalen und die charakteristischen Polynome dieser Matrix gilt:

$$\chi_{J_{P,m}} = \varphi_{J_{P,m}} = P^m.$$

Somit zeigen wir, dass eine Ähnlichkeitstransformation existiert, die jeden Begleitblock $C_{\psi_j{}^{m_{i,j}}}$ der Zwischenform (8.3) durch einen Jordanblock $J_{\psi_{j,m_{i,j}}}$ ersetzt. Die so gebildete Form wird beschränkte Form oder auch zweite Frobeniusform genannt. Es handelt sich hierbei wieder um eine Normalform, d.h. sie ist eindeutig bis auf eine Permutation der Blöcke auf der Diagonalen. Die Einzigartigkeit dieser Normalformen ermöglicht insbesondere zu testen, ob zwei Matrizen ähnlich sind und bei dieser Gelegenheit eine Matrix für den Übergang der einen in die andere zu erzeugen.

Übung 31. Schreiben Sie ein Programm, das bestimmt, ob zwei Matrizen A und B ähnlich sind und das die Übergangsmatrix U zurückgibt, sodass $A = U^{-1}BU$ ist (es wird `None` zurückgegeben, wenn die Matrizen nicht ähnlich sind).

9. Polynomiale Systeme

Dieses Kapitel führt die beiden vorigen weiter. Sein Gegenstand sind Gleichungssysteme mit mehreren Variablen wie in Kapitel 8. Diese Gleichungen sind nun Polynome wie in Kapitel 7. Im Vergleich zu Polynomen in einer einzigen Unbestimmten präsentieren diese in mehreren Unbestimmten einen großen mathematischen Reichtum, aber auch neue Schwierigkeiten, die besonders damit zusammenhängen, dass der Ring $\mathbb{K}[x_1, \ldots, x_n]$ kein Hauptidealring ist. Die Theorie der Gröbnerbasen liefert Werkzeuge, diese Begrenzung zu umgehen. Im Endeffekt verfügen wir über machtvolle Methoden zur Untersuchung polynomialer Systeme mit unzähligen Anwendungen auf den verschiedensten Gebieten.

Ein guter Teil des Kapitels setzt nur Grundkenntnisse über Polynome in mehreren Unbestimmten voraus. Manche Passagen sind jedoch auf dem Niveau einer Vorlesung über kommutative Algebra im fortgeschrittenen Bachelor- oder im Masterstudium. Sucht der Leser eine leichter zugängliche Einführung in die mathematische Theorie der polynomialen Systeme, kommt (in französischer Sprache) der Beitrag von Faugère und Safey El Din in [FSE09] in Frage. Eine eher fortgeschrittene Behandlung findet man in dem Buch von Elkadi und Mourrain [EM07]. Schließlich, und diesmal in englischer Sprache, ist das Buch von Cox, Little und O'Shea [CLO07] sowohl leicht zugänglich als auch umfassend.

9.1. Polynome in mehreren Unbestimmten

9.1.1. Die Ringe $A[x_1, \ldots, , x_n]$

Wir interessieren uns hier für Polynome in mehreren Unbestimmten, die auch - mit dem in der symbolischen Mathematik verbreiteten Anglizismus - multivariat genannt werden.

Wie bei anderen algebraischen Strukturen, die es in Sage gibt, müssen wir vor der Bildung von Polynomen eine Familie von Unbestimmten definieren, die alle zum selben Ring gehören. Die Synntax ist praktisch die gleiche wir für eine Variable (siehe Unterabschnitt 7.1.1):

```
sage: R = PolynomialRing(QQ, 'x,y,z')
 sage: x,y,z = R.gens() # ergibt das n-Tupel der Unbestimmten
```

oder kürzer:

```
sage: R.<x,y,z> = QQ[]
```

(oder auch `R = QQ['x,y,z']`). Der Konstruktor `PolynomialRing` erlaubt auch die Erzeugung einer Familie von Unbestimmten gleichen Namens mit ganzzahligen Indizes:

```
sage: R = PolynomialRing(QQ, 'x', 10)
```

Die Unterbringung des n-Tupels, das von `gens` zurückgegeben wird, in der Variablen `x` erlaubt nun natürlich mit `x[i]` den Zugriff auf die Unbestimmte x_i:

```
sage: x = R.gens()
sage: sum(x[i] for i in xrange(5))
x0 + x1 + x2 + x3 + x4
```

Die Reihenfolge der Variablen ist wichtig. Der Vergleich von `QQ['x,y']` und `QQ['y,x']` mit `==` gibt falsch zurück und dasselbe Polynom, das als Element des einen oder des anderen gesehen wird, wird unterschiedlich ausgegeben:

```
sage: def test_poly(ring, deg=3):
 ....:     monomials = Subsets(
....:     flatten([(x,)*deg for x in (1,) + ring.gens()]),
....:     deg, submultiset=True)
....:     return add(mul(m) for m in monomials)

sage: test_poly(QQ['x,y'])
x^3 + x^2*y + x*y^2 + y^3 + x^2 + x*y + y^2 + x + y + 1
sage: test_poly(QQ['y,x'])
y^3 + y^2*x + y*x^2 + x^3 + y^2 + y*x + x^2 + y + x + 1
sage: test_poly(QQ['x,y']) == test_poly(QQ['y,x'])
True
```

Übung 32. Erläutern Sie die Arbeitsweise der soeben definierten Funktion `test_poly`.

Des weiteren erfordert das Schreiben des Polynoms in kanonischer Form die Festlegung einer Reihenfolge der Summanden. Das Ordnen nach der Größe der Exponenten drängt sich auf, wenn es nur eine Unbestimmte gibt, doch bei multivariaten Polynomen erfüllt keine Reihenfolge der Monome alle Wünsche. Sage ermöglicht deshalb mit der Option `order` von `PolynomialRing` die Wahl zwischen verschiedenen Möglichkeiten. Die Reihenfolge `deglex` zum Beispiel ordnet die Monome nach dem Totalgrad, der Summe der Exponenten und bei Gleichheit dann nach der lexikographischen Reihenfolge der Unbestimmten:

```
sage: test_poly(PolynomialRing(QQ, 'x,y', order='deglex'))
x^3 + x^2*y + x*y^2 + y^3 + x^2 + x*y + y^2 + x + y + 1
```

Konstruktion von Polynomringen	
Ring $A[x,y]$	`PolynomialRing(A, 'x,y')` oder `A['x,y']`
Ring $A[x_0,\ldots,x_{n-1}]$	`PolynomialRing(A, 'x', n)`
Ring $A[x_0,x_1,\ldots,y_0,y_1,\ldots]$	`InfinitePolynomialRing(A, ['x','y'])`
n-Tupel von Generatoren	`R.gens()`
1., 2.,... Generator	`R.0, R.1,...`
Unbestimmte von $R = A[x,y][z]\ldots$	`R.variable_names_recursive()`
Umwandlung $A[x_1,x_2,y] \to A[x_1,x_2][y]$	`p.polynomial(y)`

Zugriff auf Koeffizienten	
Exponenten, Koeffizienten	`p.exponents(), p.coefficients()`
Koeffizient eines Monoms	`p[x^2*y]` oder `p[2,1]`
Grad(e) gesamt, in x, partiell	`p.degree(), p.degree(x), p.degrees()`
Leitmonom/Leitkoeffizient/Leitterm	`p.lm(), p.lc(), p.lt()`

Grundlegende Operationen	
Transformation der Koeffizienten	`p.map_coefficients(f)`
partielle Ableitung $\partial/\partial x$	`p.dervative(x)`
Auswertung $p(x,y)_{x=a,y=b}$	`p.subs(x=a, y=b)` oder `p(x=a, y=b)`
Homogenisierung	`p.homogenize()`
Hauptnenner ($p \in \mathbb{Q}[x,y,\ldots]$)	`p.denominator`

Tab. 9.1 - Polynome in mehreren Unbestimmten

Die wichtigsten verfügbaren Ordnungen werden in Unterabschnitt 9.3.1 genau beschrieben. Wir werden sehen, dass die Wahl der Ordnung nicht nur eine Frage der Ausgabe ist, sondern auch bestimmte Rechnungen beeinflusst.

Übung 33. Definieren Sie den Ring $\mathbb{Q}[x_2, x_3, \ldots, x_{37}]$, dessen Unbestimmte durch die Primzahlen unterhalb 40 indiziert sind und mit `x2, x3,...,x37` auf die Unbestimmten zugegriffen werden kann.

Es kann sich in einigen Fällen als nützlich erweisen, Polynome in mehreren Unbestimmten in *rekursiver Darstellung* zu bearbeiten, d.h. als Elemente eines Polynomrings mit Koeffizienten, die selber Polynome sind (siehe den Rahmen auf Seite 129)

9.1.2. Polynome

Genau wie Polynome in einer Variablen Instanzen der Klasse `Polynomial` sind, sind Polynome in mehreren Variablen (in Ringen mit einer endlichen Anzahl von Unbestimmten) Instanzen der Klasse `MPolynomial`[1]. Bei normalen Basisringen (wie $\mathbb{Z}$, $\mathbb{Q}$ oder $\mathbb{F}_q$) stützen sie sich auf das Programm Singular, ein CAS, das auf schnelle Rechnungen mit Polynomen spezialisiert ist. In den übrigen Fällen weicht Sage auf eine viel langsamere generische Implementierung aus.

[1] Anders als `Polynomial` ist diese Klasse nicht direkt nach der Kommandozeile zugänglich: man muss den vollständigen Namen verwenden. Wir können beispielsweise mit `isinstance(p, sage.rings.polynomial.multi_polynomial.MPolynomial)` testen, ob ein Objekt ein multivariates Polynom ist

Die Ringe $A[(x_n, y_n \ldots)_{n\in\mathbb{N}}]$

Es kommt vor, dass man bei Beginn einer Rechnung nicht weiß, wieviele Variablen gebraucht werden. Das nimmt `PolynomialRing` jedoch ziemlich genau: man muss mit einer ersten Definitionsmenge beginnen und dann jedesmal, wenn eine neue Variable hinzukommen soll, erweitern und alle Elemente konvertieren.
Polynomringe in unendlich vielen Unbestimmten bieten eine viel flexiblere Datenstruktur. Ihre Elemente können Variable enthalten, die aus einer oder mehreren unendlichen Familien von Unbestimmten genommen werden. Jeder Generator des Ringes entpricht nicht einer einzigen Variablen, sondern einer Familie von Variablen, die durch natürliche Zahlen indiziert werden:

```
sage: R.<x,y> = InfinitePolynomialRing(ZZ, order='lex')
 sage: p = mul(x[k] - y[k] for k in range(2)); p
x_1*x_0 - x_1*y_0 - x_0*y_1 + y_1*y_0
sage: p + x[100]
x_100 + x_1*x_0 - x_1*y_0 - x_0*y_1 + y_1*y_0
```

Wir gelangen mit der Methode `polynomial` zu einem normalen Polynomring `PolynomialRing` zurück, der das Bild eines Elementes eines `InfinitePolynomialRing` aus einem Ring zurückgibt, der groß genug ist, um alle Elemente des Ringes in unendlich vielen Variablen zu enthalten, die bis dahin bearbeitet sind. Der erhaltene Ring ist im allgemeinen nicht der kleinste mit dieser Eigenschaft.
Im Gegensatz zu dieser Flexibilität sind diese Ringe weniger effizient als Ringe aus `PolynomialRing`. Außerdem werden ihre *Haupideale* bei Rechnungen mit polynomialen Systemen, dem zentralen Gegenstand dieses Kapitels, nicht durch solche der normalen Polynomringe ersetzt

Polynome in mehreren Unbestimmten werden immer in dünn besetzter Darstellung kodiert[2]. Wozu die Festlegung? Ein voll besetztes Polynom in n Variablen des gesamten Grades d zählt $\binom{n+d}{n}$ Monome: für $n = d = 10$ macht das 184756 zu speichernde Koeffizienten! Es ist deshalb sehr schwierig, grosse voll besetzte Polynome so zu behandeln, wie man das bei einer Variablen tut. Außerdem haben die Stützstellen (die Positionen der von 0 verschiedenen Monome), denen man in der Praxis begegnet, verschiedene Formen. Wenn nun beispielsweise ein bis zum gesamten Grad $d-1$ voll besetztes Polynom in n Variablen mit großem d durch ein rechteckiges Schema $d \times \ldots \times d$ dargestellt wird, ist nur etwa ein Koeffizient von $n!$ von null verschieden. Umgekehrt wird die dünn besetzte Darstellung durch ein Diktionär an die Form der Stützstellen sowie an die auf den Monomen geltende Ordnung angepasst.

9.1.3. Grundlegende Operationen

Legen wir etwas Terminologie fest. Sei $R = A[x_1, \ldots, x_n]$ ein Polynomring. Wir nennen *Monom* einen Ausdruck der Form $x_1^{\alpha_1} x_2^{\alpha_2} \ldots x_n^{\alpha_n}$, d.h. ein Produkt von Unbestimmten, das wir abkürzend $\boldsymbol{x}^\alpha$ nennen. Das n-Tupel ganzer Zahlen $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_n)$ ist der *Exponent* des Monoms $\boldsymbol{x}^\alpha$. Ein *Term* ist ein Monom multipliziert mit einem Element aus A, seinem Koeffizienten.

[2] Die rekursive Darstellung (siehe Kasten Seite 129) liefert indessen eine teilweise voll besetzte Form multivariater Polynome. Bei der Darstellung eines Polynoms aus $A[x][y]$ im Speicher belegt jeder Koeffizient von y^k (im allgemeinen) Platz, der zu seinem Grad in x proportional ist und dem Platz proportional zum Grad von y für das Polynom selbst hinzugefügt werden muss.

Da die Anordnung der Terme nicht eindeutig ist, haben die Elemente von R als mathematische Objekte keinen Leitkoeffizienten. Ist bei der Erzeugung des Ringes aber eine Ordnung festgelegt worden, ist es möglich und nützlich, das in der Ausgabe am weitesten links stehende Monom als Leitmonom festzulegen. Die Methoden `lm` (für *leading monomial*), `lc` (*leading coefficient*) und `lt` (*leading term*) eines Polynoms in mehreren Varianten geben das Leitmonom, dessen Koeffizienten bzw. den Term, den sie zusammen bilden, zurück:

```
sage: R.<x,y,z> = QQ[]
sage: p = 7*y^2*x^2 + 3*y*x^2 + 2*y*z + x^3 + 6
sage: p.lt()
7*x^2*y^2
```

Die arithmetischen Operationen +, - und * ebenso wir die Methoden `dict, coefficients` und viele andere werden wie ihre Analoga in nur einer Variablen verwendet. Zu den kleinen Unterschieden gehört, dass der Operator [] für die Extraktion eines Koeffizienten als Parameter sowohl dessen Monom als auch dessen Exponenten akzeptiert:

```
sage: p[x^2*y] == p[(2,1,0)] == p[2,1,0] == 3
True
```

Ebenso müssen bei der Auswertung Daten für alle Variablen eingegeben oder die zu substituierenden präzisiert werden:

```
sage: p(0, 3, -1)
0
sage: p.subs(x = 1, z = x^2+1)
 2*x^2*y + 7*y^2 + 5*y + 7
```

Die Methode `subs` kann auch gleichzeitig eine beliebige Anzahl von Variablen ersetzen, siehe die Dokumentation hinsichtlich anspruchsvollerer Beispiele. Der Grad kann als gesamte Summe der Exponenten (total) und für einzelne Variablen (partiell) ausgegeben werden:

```
sage: print "total={d} (nur x)={dx} partiell={ds}"\
....:   .format(d=p.degree(), dx=p.degree(x), ds=p.degrees())
total=4    (nur x)=3    partiell=(3, 2, 1)
```

Andere Konstruktionen erfahren evidente Anpassungen, beispielsweise erhält die Methode `derivative` diejenige Variable als Parameter, nach der abgeleitet werden soll.

9.1.4. Arithmetik

Außer den elementaren syntaktischen und arithmetischen Operationen sind die in Sage verfügbaren Funktionen generell auf Polynome auf einem Körper und zuweilen auf $\mathbb{Z}$ oder $\mathbb{Z}/n\mathbb{Z}$ beschränkt. In diesem Kapitel bewegen wir uns auf einem Körper, wenn anders nicht explizit erwähnt.

Die euklidsche Polynomdivision hat nur in einer Variablen Sinn. In Sage bleiben die Methode `quo_rem` und die zugehörigen Operatoren // und % für multivariate Polynome dennoch definiert. Die „Division mit Rest“, die damit berechnet wird, genügt der Beziehung

```
(p//q)*q + (p%q) == p
```

und stimmt mit der euklidschen Division überein, wenn p und q nur von einer Variablen abhängen. Das ist dann aber selbst keine euklidsche Division und sie ist nicht kanonisch. Sie erweist sich dennoch als nützlich, wenn die Division exakt ist oder wenn der Divisor ein Monom ist. In den übrigen Fällen ziehen wir die in Unterabschnitt 9.2.3 beschriebene Funktion `mod` der Funktion `quo_rem` und ihren Varianten vor, weil sie ein Polynom modulo eines Haupideals reduziert und die Festlegung der Ordnung der Monome berücksichtigt:

```
sage: R.<x,y> = QQ[]; p = x^2 + y^2; q = x + y
sage: print("({quo})*({q}) + ({rem}) == {p}".format( \
....:           quo=p//q, q=q, rem=p%q, p=p//q*q+p%q))
(-x + y)*(x + y) + (2*x^2) == x^2 + y^2
sage: p.mod(q) # ist NICHT äquivalent zu p%q
```

Die Methoden `divides, gcd, lcm` oder auch `factor` haben die gleiche Bedeutung wie bei nur einer Variablen. Wegen fehlerhafter euklidscher Division sind die ersteren bei Koeffizienten beliebigen Typs nicht einsetzbar, sie funktionieren jedoch auf diversen gebräuchlichen Körpern, beispielsweise dem Körper der Zahlen:

```
sage: R.<x,y> = QQ[exp(2*I*pi/5)][]
sage: (x^10 + y^5).gcd(x^4 - y^2)
x^2 + y
sage: (x^10 + y^5).factor()
(x^2 + y) * (x^2 + (a^3)*y) * (x^2 + (a^2)*y) * (x^2 + (a)*y) * (x^2 +
(-a^3 - a^2 - a - 1)*y)
```

9.2. Polynomiale und ideale Systeme

Nun wenden wir uns dem zentralen Gegenstand dieses Kapitels zu. Die Unterabschnitte 9.2.1 und 9.2.2 bieten ein Panorama der verschiedenen Möglichkeiten, mit Hilfe von Sage Lösungen eines Systems von polynomialen Gleichungen zu finden und zu verstehen. Der Unterabschnitt 9.2.3 ist den diesen Systemen zugeordneten Haupidealen gewidmet. Die danach folgenden Abschnitte kommen auf die Werkzeuge zur algebraischen Elimination und zur Lösung dieser Systeme vertiefend zurück.

9.2.1. Ein erstes Beispiel

Wir betrachten eine Variante des polynomialen Systems aus Abschnitt 2.2:

$$\begin{cases} x^2yz = 18 \\ xy^3z = 24 \\ xyz^4 = 6. \end{cases} \tag{9.1}$$

Operationen auf Polynomen mit mehreren Unbestimmten	
Teilbarkeit $p \mid q$	`p.divides(q)`
Faktorisierung	`p.factor()`
ggT, kgV	`p.gcd(q), p.lcm(q)`
Test ob quadratfrei	`p.is_squarefree`
Resultante $\mathrm{Res}_x(p,q)$	`p.resultant(q, x)`

Tab. 9.2 - Arithmetik

Sages Funktion `solve` hatte uns nur erlaubt, eine numerische Lösung zu finden. Wir sehen jetzt, wie es Sage gelingt, das System exakt zu lösen, und mit etwas Unterstützung durch den Anwender einfache geschlossene Formen für alle Lösungen zu finden[3].

Auflisten der Lösungen Wir beginnen damit, das Problem in mehr algebraische Form zu bringen, indem wir das Ideal von $\mathbb{Q}[x, y, z]$ mit diesen Gleichungen bilden:

```
sage: R.<x,y,z> = QQ[]
sage: J = R.ideal(x^2 * y * z - 18,
....:             x * y^3 * z - 24,
....:             x * y * z^4 - 6)
```

Wie wir in Unterabschnitt 9.2.3 sehen werden, erlaubt uns der folgende Befehl zu verifizieren, dass das Ideal J die Dimension null hat, d.h. dass das System (9.1) auf $\mathbb{C}^3$ eine endliche Zahl von Lösungen besitzt.

```
sage: J.dimension()
 0
```

Der erste Reflex ist nun, mit der Methode `variety` alle Lösungen des System zu berechnen. Ohne Parameter liefert sie alle Lösungen auf dem Basiskörper des Polynomrings:

```
sage: J.variety()
[{y: 2, z: 1, x: 3}]
```

Die bereits gefundene Lösung $(3, 2, 1)$ ist demnach die einzige Lösung des Systems.

Der folgende Schritt besteht in der Auflistung der komplexen Lösungen. Um das exakt zu tun, arbeiten wir auf dem Körper der algebraischen Zahlen. Wir bekommen 17 Lösungen:

```
sage: V = J.variety(QQbar)
 sage: len(V)
17
```

Die letzten drei Lösungen haben folgendes Aussehen:

```
sage: V[-3:]
[{z: 0.9324722294043558? - 0.3612416661871530?*I, y: -1.700434271459229?
+ 1.052864325754712?*I, x: 1.337215067329615? - 2.685489874065187?*I},
 {z: 0.9324722294043558? + 0.3612416661871530?*I, y: -1.700434271459229?
- 1.052864325754712?*I, x: 1.337215067329615? + 2.685489874065187?*I},
 {z: 1, y: 2, x: 3}]
```

Die Lösungspunkte werden als Diktionär ausgegeben, dessen Schlüssel die Generatoren von `QQbar['x,y,z']` sind (und nicht von `QQ['x,y,z']`, von wo mit einem kleinen Umweg darauf zugegriffen werden kann), und die Koordinaten des Punktes die zugehörigen Werte. Außer dem vorhin identifizierten Wert der rationalen Lösung sind die ersten Koordinaten sämtlich algebraische Zahlem 16. Grades:

[3] Weil der Zweck der Übung ist, die Werkzeuge zur Lösung von polynomialen Systemen zu erläutern, lassen wir die Möglichkeit außer acht, (9.1) durch Logarithmieren in lineare Gleichungen zu überführen!

```
sage: (xx, yy, zz) = QQbar['x,y,z'].gens()
sage: [ pt[xx].degree() for pt in V ]
[16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 1]
```

Rechnen mit den Lösungen und Ermitteln ihrer Struktur. Wir haben eine exakte Darstellung der komplexen Lösungen des Systems (9.1) bekommen, doch ist diese Darstellung nicht offen explizit. Das ist nicht schlimm: mit den Koordinaten als Elementen von `QQbar` können hier die exakten Rechnungen ausgeführt weren.

Beispielsweise ist es nicht schwierig zu sehen, dass wenn (x, y, z) eine Lösung des Systems (9.1) ist, dann auch $(|x|, |y|, |z|)$. Konstruieren wir die Menge der $(|x|, |y|, |z|)$ für die Lösung (x, y, z):

```
sage: Set(tuple(abs(pt[i]) for i in (xx,yy,zz)) for pt in V)
{(3, 2, 1)}
```

Alle Werte von x (bzw. von y) sind daher vom selben Modul. Besser noch, wir können verifizieren, dass die Substitution

$$(x, y, z) \mapsto (\omega x, \omega^9 y, \omega^6 z) \quad \text{mit } \omega = e^{2\pi i/17} \tag{9.2}$$

dass System unverändert lässt. Insbesondere sind die letzten Koordinaten der Lösungen genau die 17. Einheitswurzeln, was uns deshalb dank der Möglichkeit, auf den algebraischen Zahlen genau zu rechnen, sicher sein lässt:

```
sage: w = QQbar.zeta(17); w # Primitivwurzel von 1
0.9324722294043558? + 0.3612416661871530?*I
sage: Set(pt[zz] for pt in V) == Set(w^i for i in range(17))
True
```

Lösungen des Systems sind daher die Tripel $(3\omega, 2\omega^9, \omega^6)$ für $\omega^{17} = 1$. Und das sagt doch alles.

Übung 34. Zu suchen sind die reellen Lösungen (nicht bloß die rationalen), um unmittelbar zu verifizieren, dass es nur $(3, 2, 1)$ gibt. Finden Sie durch Berechnung mit Sage auch die Substitution (9.2), darin auch den Wert 17 für die Ordnung von ω als Wurzel der Einheit.

Zum gleichen Ergebnis hätten wir auch durch einen Blick auf die Minimalpolynome der Koordinaten der Punkte in V kommen können. Wir erkennen, dass die gleiche Koordinate für alle anderen Lösungspunkte als $(3, 2, 1)$ dasselbe Minimalpolynom hat. Das gemeinsame Minimalpolynom ihrer dritten Koordinate ist nichts anderes als das zyklotome Polynom Φ_{17}.

```
sage: set(pt[zz].minpoly() for pt in V[:-1])
{x^16 + x^15 + x^14 + x^13 + x^12 + x^11 + x^10 + x^9 +
x^8 + x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x + 1}
```

Die gemeinsamen Minimalpolynome der ersten und der zweiten Koordinate sind $3^{16} \cdot \Phi_{17}(x/3)$ bzw. $2^{16} \cdot \Phi_{17}(x/2)$.

Geschlossene Formeln. Ein expliziter Ausdruck für die Lösungen ist deshalb durch Rückgriff auf die Exponentialschreibweise der komplexen Zahlen möglich:

```
sage: def polar_form(z):
....:     rho = z.abs(); rho.simplify()
....:     theta = 2 * pi * z.rational_argument()
 ....:     return (SR(rho) * exp(I*theta))
sage: [tuple(polar_form(pt[i]) for i in [xx,yy,zz]) for pt in V[-3:]]
[(3*e^(-6/17*I*pi), 2*e^(14/17*I*pi), e^(-2/17*I*pi)),
 (3*e^(6/17*I*pi), 2*e^(-14/17*I*pi), e^(2/17*I*pi)), (3, 2, 1)]
```

Wären wir nicht auf die Idee gekommen, die Elemente von V als Potenzen zu schreiben, hätte das am Ende genügt.

Vereinfachung des Systems. Es ist auch ein anderer Ansatz möglich. Statt nach Lösungen zu suchen, versuchen wir, eine einfachere Form des Systems zu berechnen. Die grundlegenden Werkzeuge, die Sage dafür anbietet, sind die Dreieckszerlegung und die Gröbnerbasen. Wir werden im weiteren sehen, was sie exakt berechnen; versuchen wir schon einmal, sie mit diesem Beispiel zu verwenden:

```
sage: J.triangular_decomposition()
 [Ideal (z^17 - 1, y - 2*z^10, x - 3*z^3) of Multivariate
Polynomial Ring in x, y, z over Rational Field]
sage: J.transformed_basis()
 [z^17 - 1, -2*z^10 + y, -3/4*y^2 + x]
```

Auf die eine Weise wie auf die andere erhalten wir das äquivalente System

$$z^{17} = 1 \quad y = 2z^{10} \quad x = 3z^3,$$

dasselbe wie $V = \{(3\omega^3, 2\omega^{10}, \omega) \mid \omega^{17} = 1\}$. Das ist wieder eine unmittelbare Parameterdarstellung der kompakten Schreibweise der oben bereits von Hand gefundenen Lösungen.

9.2.2. Was heißt Lösen?

Ein polynomiales System, das Löungen besitzt, hat davon oft unendlich viele. Die sehr einfache Gleichung $x^2 - y = 0$ lässt auf $\mathbb{Q}^2$ unendlich viele Lösungen zu, ganz zu schweigen von $\mathbb{R}^2$ oder $\mathbb{C}^2$. Es ist dann nicht möglich sie aufzulisten. Das beste, was man tun kann, ist die Menge der Lösungen „so explizit wie möglich“ zu beschreiben, d.h. eine Darstellung zu berechnen, aus der die interessierenden Informationen leicht extrahiert werden können. Die Situation ist ähnlich wie bei linearen Systemen, für die (im homogenen Fall) eine Basis des Kerns des Systems eine gute Beschreibung des Lösungsraumes ist.

In dem Spezialfall, wo die Anzahl der Lösungen endlich ist, wird es möglich, „sie zu berechnen“. Doch versucht man auch in diesem Fall, die Lösungen in $\mathbb{Q}$ oder einem endlichen Körper $\mathbb{F}_q$ aufzulisten? Um numerische Näherungen der reellen oder komplexen Lösungen zu finden? Oder auch, wie im Beispiel aus dem vorigen Abschnitt, die letzteren mit Hilfe algebraischer Zahlen darzustellen, d.h. beispielsweise die Minimalpolynome ihrer Koordinaten zu berechnen?

Dasselbe Beispiel illustriert, dass andere Darstellungen der Lösungsmenge als eine einfache Liste vor allem dann aussagekräftiger sein könnenn, wenn die Lösungen zahlreich sind. Somit ist eine Aufzählung nicht zwangsläufig die beste Art des Vorgehens, selbst wenn das möglich ist. Kurzum, wir versuchen nicht so sehr, Lösungen zu finden, sondern mit ihnen zu rechnen, um daraus dann gemäß der Aufgabenstellung die Informationen zu gewinnen, für die wir uns eigentlich interessieren. Dieses Kapitel erkundet verschiedene Werkzeuge, dies zu erreichen.

9.2.3. Ideale und Systeme

Wenn s Polynome $p_1, \ldots, p_s \in \mathbb{K}[x]$ in einem Punkt $\boldsymbol{x}$ mit Koordinaten aus $\mathbb{K}$ oder einer Erweiterung von $\mathbb{K}$ null werden, dann verschwindet auch jedes Element des Ideals, das sie erzeugen, im Punkt $\boldsymbol{x}$. Es ist deshalb nur natürlich, dem polynomialen System

$$p_1(\boldsymbol{x}) = p_2(\boldsymbol{x}) = \ldots = p_s(\boldsymbol{x}) = 0$$

das Ideal $J = \langle p_1, \ldots, p_a \rangle \subset \mathbb{K}[\boldsymbol{x}]$ zuzuordnen. Zwei polynomiale Systeme, die dasselbe Ideal erzeugen, sind in dem Sinne äquivalent, dass sie die gleichen Lösungen haben. Ist L ein $\mathbb{K}$ enthaltender Körper, dann nennen wir die Menge

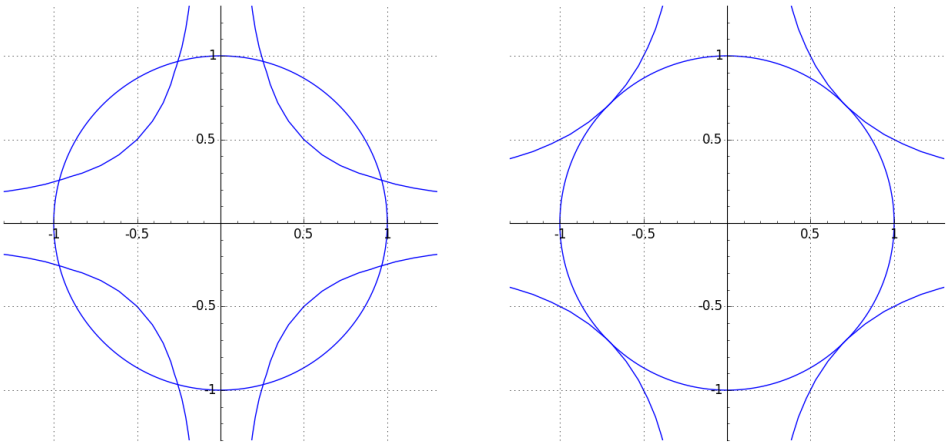
$$V_L(J) = \{\boldsymbol{x} \in L^n \mid \forall p \in J, p(\boldsymbol{x}) = 0\} = \{\boldsymbol{x} \in L^n \mid p_1(\boldsymbol{x}) = \ldots = p_s(\boldsymbol{x}) = 0\}$$

der Lösungen des Systems mit Koordinaten aus L eine *algebraische Subvarietät*. Verschiedene Ideale können dieselbe Varietät zugeordnet haben. Beispielsweise haben die Gleichungen $x = 0$ und $x^2 = 0$ in $\mathbb{C}$ dieselbe eindeutige Lösung, obwohl wir $\langle x^2 \rangle \subsetneq \langle x \rangle$ haben. Was ein von einem polynomialen System erzeugtes Ideal vor allem bringt, ist die intuitive Schreibweise der „Lösung mit Multiplizitäten“.

So drücken die beiden folgenden Systeme jedes den Schnitt des Einheitskreises mit dem Graphen der Gleichung $\alpha x^2 y^2 = 1$ aus, wobei jeweils zwei gleichseitige Hyperbeln auftreten (siehe Abb. 9.1):

$$(S_1) \begin{cases} x^2 + y^2 = 1 \\ 16x^2y^2 = 1 \end{cases} \qquad (S_2) \begin{cases} x^2 + y^2 = 1 \\ 4x^2y^2 = 1 \end{cases} \tag{9.3}$$

Das System (S_1) besitzt auf $\mathbb{C}$ acht Lösungen, alle mit reellen Koordinaten. Verzerrt man (S_2) durch Veränderung des Parameters α, haben sich die beiden Lösungen auf jedem Ast der Hyperbel soweit angenähert, dass sie verschmelzen. Das System S_2 hat nur vier Lösungen, jede in gewissem Sinne zweifach. Würde α noch weiter verringert, gäbe es keine reelle Lösung mehr, dafür aber acht komplexe.



(a) (S_1) (b) (S_2)

```
sage: R.<x,y> = QQ[]
sage: J = R.ideal(x^2 + y^2 - 1, 16*x^2*y^2 - 1)
sage: opts = {'axes':True, 'gridlines':True, 'frame':False,
....:     'aspect_ratio':1, 'axes_pad':0, 'fontsize':12,
....:     'xmin':-1.3, 'xmax':1.3, 'ymin':-1.3, 'ymax':1.3}
sage: (ideal(J.0).plot() + ideal(J.1).plot()).show(**opts)[4]
```

Abb. 9.1 - Schnitt zweier ebener Kurven, siehe die Systeme (9.3)

Rechnung modulo eines Ideals. Wie im Fall der Polynome mit nur einer Unbestimmten erlaubt Sage die Definition von Idealen[5] $J \subset \mathbb{K}[\boldsymbol{x}]$, von Quotientenringen $\mathbb{K}[x]/J$ und natürlich das Rechnen mit den Elementen dieser Quotientenringe. Das (S_1) zugeordnete Ideal J_1 wird gebildet mit

```
sage: R.<x,y> = QQ[]
sage: J = R.ideal(x^2 + y^2 - 1, 16*x^2*y^2 - 1)
```

Dann können wir den Quotienten von $\mathbb{K}[\boldsymbol{x}]$ durch J_1 ausführen, dabei Polynome auswerfen, mit Äquivalenzklassen modulo J_1 rechnen und sie als Repräsentanten „aufleben“ lassen:

```
sage: ybar2 = R.quo(J)(y^2)
sage: [ybar2^i for i in range(3)]
[1, ybar^2, ybar^2 - 1/16]
sage: ((ybar2 + 1)^2).lift()
3*y^2 + 15/16
```

Hier gibt es eine theoretische Schwierigkeit. Die Elemente von $\mathbb{K}[\boldsymbol{x}]/J$ werden in Normalform dargestellt, was erforderlich ist, um zwei Elemente auf Gleichheit testen zu können. Nun ist diese Normalform aus dem in Unterabschnitt 9.1.4 schön erwähnten Grund nicht eindeutig zu definieren: die Division des Repräsentanten einer Äquivalenzklasse $p + J$ durch einen Hauptgenerator von J, die zum Rechnen auf $\mathbb{K}[x]/J$ gebraucht wird, hat bei mehreren Variablen keine Entsprechung. Begnügen wir uns für den Moment damit, dass nichtsdestotrotz eine Normalform existiert, die von der bei der Erzeugung des Ringes festgelegten Ordnung auf den Elementen abhängt und die auf einem besonderen System von Erzeugenden beruht, die Gröbnerbasen heißen. Der Abschnitt 9.3 am Ende dieses Kapitels ist der Definition der Gröbnerbasen gewidmet und soll zeigen, wie wir sie bei den Rechnungen verwenden können. Sobald erforderlich, berechnet Sage Gröbnerbasen automatisch; diese Rechnungen sind gelegentlich jedoch sehr aufwendig, besonders wenn die Anzahl der Variablen sehr groß ist, sodass die Rechnung auf einem Quotientenring schwierig sein kann.

Kehren wir zu Sage zurück. Wenn $p \in J$ ist, schreibt der Befehl `p.lift(J)` p als lineare Kombination mit Polynomen des Generators von J als Koeffizienten:

```
sage: u = (16*y^4 - 16*y^2 + 1).lift(J); u
[16*y^2, -1]
```

[4] Dieser Code erzeugt nur Teil (a); um Teil (b) zu erhaltem muss die Definition von J geändert werden.

[5] Vorsicht: die Objekte `InfinitePolynomialRing` haben ebenfalls eine Methode `ideal`, die aber nicht dasselbe Verhalten zeigt wie bei den normalen Polynomringen. (Ein beliebiges Ideal von $k[(x_n)_{n\in\mathbb{N}}]$ hat keinen Grund, endlich erzeugt zu sein!) Diese Objekte werden im weiteren Verlauf des Kapitels nicht verwendet.

```
sage: u[0]*J.0 + u[1]*J.1
16*y^4 - 16*y^2 + 1
```

Zu einem beliebigen Polynom ergibt der Ausdruck `p.mod(J)` die Normalform von p modulo J, die als Element von $\mathbb{K}[\boldsymbol{x}]$ gesehen wird:

```
sage: (y^4).mod(J)
y^2 - 1/16
```

Achtung: wenn `J.reduce(p)` auch zu `p.mod(J)` äquivalent ist, so gibt andererseits die Variante `p.reduce([p1, p2, ...])` einen Repräsentanten von $p + J$ zurück, der nicht unbedingt in Normalform ist (siehe Unterabschnitt 9.3.2).

```
sage: (y^4).reduce([x^2 + y^2 - 1, 16*x^2*y^2 - 1])
y^4
```

Durch Kombination von `p.mod(J)` und `p.lift(J)` können wir ein Polynom p in eine Linearkombination mit polynomialen Koeffizienten des Generators von J plus einem Rest zerlegen, der genau dann verschwindet, wenn $p \in J$ ist.

Radikal eines Ideals und Lösungen. Der wesentliche Punkt bei der Entsprechung von Idealen und Varietäten ist der Hilbertsche *Nullstellensatz*. Sei $\bar{K}$ ein algebraischer Abschluss von $\mathbb{K}$.

Satz *(Nullstellensatz).* Seien $p_1, \ldots, p_s \in \mathbb{K}[\boldsymbol{x}]$ und $\bar{Z} \subset \bar{K}^n$ die Menge der gemeinsamen Nullstellen von p_i. Ein Polynom $p \in \mathbb{K}[\boldsymbol{x}]$ verschwindet auf Z identisch, genau dann, wenn eine ganze Zahl k existiert, sodass $p^k \in \langle p_1, \ldots, p_2 \rangle$ ist.

Ideale	
Ideal $\langle p_1, p_2 \rangle \subset R$	`R.ideal(p1, p2)` oder `(p1, p2)*R`
Summe, Produkt, Potenz	`I + J`, `I*J`, `I^k`
Schnittmenge $I \cap J$	`I.intersection(J)`
Quotient $I : J = \{p \mid pJ \in I\}$	`I.quotient(J)`
Radikal $\sqrt{J}$	`J.radical()`
Reduktion modulo J	`J.reduce(p)` oder `p.mod(J)`
Sektion von $R \twoheadrightarrow R/J$	`p.lift(J)`
Quotientenring R/J	`R.quo(J)`
homogenisiertes Ideal	`J.homogenize()`

Einige vordefinierte Ideale	
„irrelevantes Ideal“ $\langle x_1, \ldots, x_n \rangle$	`R.irrelevant_ideal()`
Jacobi-Ideal $\langle \partial p/\partial x_i \rangle_i$	`p.jacobian_ideal()`
„zyklische Wurzeln“ (9.11)	`sage.rings.ideal.Cyclic(R)`
Körpergleichungen $x_i^q = x_i$	`sage.rings.ideal.FieldIdeal(GF(q)['x1,x2'])`

Tab. 9.3 - Ideale

Dieses Result liefert ein algebraisches Kriterium für den Test, ob ein polynomiales System Lösungen besitzt. Das konstante Polynom 1 wird auf Z genau dann identisch 0, wenn Z leer ist. Deshalb hat das System $p_1(\boldsymbol{x}) = \ldots = p_s(\boldsymbol{x})$ genau dann Lösungen in $\bar{K}$, wenn 1 im Ideal $\langle p_1, \ldots, p_s \rangle$ nicht enthalten ist. Beispielsweise schneiden sich die Kreise mit dem Radius 1 und den Zentren in $(0, 0)$ und $4, 0$ im Komplexen:

```
sage: 1 in ideal(x^2+y^2-1, (x-4)^2+y^2-1)
False
```

Andererseits hat das System keine Lösung mehr, wenn $x = y$ hinzugefügt wird. Wir können einen trivialen Beweis angeben, um zu verifizieren, dass es schon ein Widerspruch ist, wenn wir eine Kombination von untereinander widerspruchsfreien Gleichungen zeigen, die sich auf $1 = 0$ reduzieren lässt. Die Rechnung

```
sage: R(1).lift(ideal(x^2+y^2-1, (x-4)^2+y^2-1, x-y))
[-1/28*y + 1/14, 1/28*y + 1/14, -1/7*x + 1/7*y + 4/7]
```

liefert in unserem Fall die Beziehung

$$\frac{1}{28}\left((-y+2)(x^2+y^2+1)+(y+2)\left((x-4)^2+y^2-1\right)+(-4x+4y+16)(x-y)\right)=1.$$

In Begriffen der Ideale bestätigt der *Nullstellensatz*, dass die Menge der Polynome, die auf der einem Ideal J zugeordneten Varietät $V_{\bar{K}}(J)$ identisch null sind, das *Radikal* dieses Ideals ist, das definiert ist durch

$$\sqrt{J} = \left\{p \in \mathbb{K}[\boldsymbol{x}] \mid \exists k \in \mathbb{N}.p^k \in J\right\}.$$

Wir haben

$$V(\left(\sqrt{J}\right) = V(J),$$

doch intuitiv werden beim Übergang zum Radikal die „Multiplizitäten vergessen“. Somit ist das Ideal J_1 sein eigenes Radical (man sagt, es sei radikal), wohingegen das (S_2) zugeordnete Ideal J_2 der Beziehung $J_{\subsetneq}\sqrt{J_2}$ genügt:

```
sage: J1 = (x^2 + y^2 - 1, 16*x^2*y^2 - 1)*R
sage: J2 = (x^2 + y^2 - 1, 4*x^2*y^2 - 1)*R
sage: J1.radical() == J1
True
sage: J2.radical()
Ideal (2*y^2 - 1, 2*x^2 - 1) of Multivariate Polynomial Ring in x, y
over Rational Field
sage: 2*y^2 - 1 in J2
False
```

Systeme, Ideale und die Kryptographie

Die spezifischen Module `sage.rings.polynomial.multi_polynomial_sequence` und `sage.crypto.mq` bieten Werkzeuge für den Umgang mit polynomialen Systemen, indem sie der besondere Gestalt der Gleichungen Rechnung tragen und nicht nur dem Ideal, das sie erzeugen. Das ist nützlich für die Arbeit mit großen strukturierten Systemen, wie solchen, die bei der Kryptographie vorkommen. Das Modul `sage.crypto` definiert auch mehrere polynomiale Systeme, die den klassischen kryptographischen Konstruktionen zugeordnet sind.

Operationen auf Idealen. Es ist auch möglich, mit den Idealen selbst zu rechnen. Erinnern wir uns, dass die Summe zweier Ideale definiert ist durch

$$I + J = \{p + q \mid p \in I \text{ und } q \in J\} = \langle I \cup J \rangle.$$

Sie entspricht geometrisch der Schnittmenge der Varietäten:

$$V(I + J) = V(I) \cap V(J).$$

Somit wird das (S_1) zugeordnete Ideal J_1 als Summe von $C = \langle x^2 + y^2 - 1 \rangle$ und $H = \langle 16x^2y^2 - 1 \rangle$ geschrieben, welche den Kreis bzw. die doppelte Hyperbel definieren. In Sage:

```
sage: C = ideal(x^2 + y^2 - 1); H = ideal(16*x^2*y^2 - 1)
sage: C + H == J1
True
```

Der Test auf Gleichheit stützt sich auch auf eine Berechnung der Gröbnerbasis.

Die Schnittmenge, das Produkt und der Quotient genügen ebenfalls den Gleichungen

$$\begin{array}{rl} I \cap J &= \{p \mid p \in I \text{ und } q \in J\} \\ I \cdot J &= \{p \mid p \in I \text{ und } q \in J\} \\ I : J &= \{p \mid pJ \subset I\} \end{array} \qquad \begin{array}{rl} V(I \cap J) &= V(I) \cup V(J) \\ V(I \cdot J) &= V(I) \cup V(J) \\ V(I : J) &= \overline{V(I) \backslash V(J)} \end{array}$$

und werden berechnet, wie in Tabelle 9.3 gezeigt. Die überstrichene Schreibweise $\bar{X}$ bezeichnet hier den *Abschluss von Zariski* von X, d.h. die kleinste algebraische Subvarietät, die X enthält. Beispielsweise ist die Kurve in Abb. 9.1 (a) die Menge der Nullstellen der Polynome von $C \cap H$ und der Quotient $(C \cap H) : \langle 4xy - 1 \rangle$ entspricht der Vereinigungsmenge des Kreises und einer der beiden Hyperbeln.

```
sage: CH = C.intersection(H).quotient(ideal(4*x*y-1)); CH
Ideal (4*x^3*y + 4*x*y^3 + x^2 - 4*x*y + y^2 - 1) of
Multivariate Polynomial Ring in x, y over Rational Field
sage: CH.gen(0).factor()
(4*x*y + 1) * (x^2 + y^2 - 1)
```

Dagegen ist die erhaltene Kurve keine algebraische Subvarietät, wenn eine endliche Anzahl von Punkten aus $V(H)$ entfernt wird, wie etwa:

```
sage: H.quotient(C) == H
True
```

Dimension. Jedem Ideal $J \subset \mathbb{K}[\boldsymbol{x}]$ ist zudem eine *Dimension* zugeordnet, die intuitiv der maximalen „Dimension“ der „Komponenten“ der Varietät $V(J)$ auf einem abgeschlossenen algebraischen Körper entspricht[6]. Zum Beispiel haben wir:

```
sage: [J.dimension() for J in [J1, J2, C, H, H*J2, J1+J2]]
[0, 0, 1, 1, 1, -1]
```

In der Tat werden $V(J_1)$ und $V(J_2)$ von einer endlichen Anzahl von Punkten gebildet, $V(C)$ und $V(H)$ sind Kurven, $V(H \cdot J_2)$ ist eine Vereinigung von Kurven und isolierten Punkten,

[6] Im Unterabschnitt 9.3.3 geben wir eine strengere, wenn auch weniger erhellende Definition. Wegen einer ausführlicheren Behandlung siehe die am Anfang des Kapitels erwähnten Literaturhinweise.

und $V(J_1 + J_2)$ ist leer. Systeme mit der Dimension null, das sind solche, die ein Ideal der Dimension null erzeugen, oder auch (bei Systemen mit gebrochenen Koeffizienten), die nur eine endliche Anzahl von Lösungen besitzen, haben im weiteren Verlauf des Kapitels eine besondere Bedeutung, weil sie meistens diejenigen sind, die wir explizit lösen können.

9.2.4. Elimination

Eine Variable in einem Gleichungssystem zu *eliminieren* heißt „Folgen“ zu finden oder besser „alle Folgen“ des Systems, die von dieser Variablen unabhängig sind. Anders gesagt handelt es sich darum, Gleichungen zu finden, die jeder Lösung genügen, in denen die eliminierte Variable aber nicht vorkommt, was sie oft leichter analysierbar macht.

Beispielsweise können wir aus dem folgenden Gleichungssystem

$$\begin{cases} 2x + y - 2z = 0 \\ 2x + 2y + z = 1 \end{cases} \tag{9.4}$$

x eliminieren, indem wir die erste Gleichung von der zweiten abziehen. Es kommt $y + 3z = 1$, was zeigt, dass jedes Lösungstripel (x, y, z) von (9.4) die Form $(x, 1 - 3z, z)$ hat.

Allgemeine polynomiale Systeme: Elimination, Geometrie	
Eliminationsideal $J \cap A[z,t] \subset \mathbb{K}[x,y,z,t]$	`J.elimination_ideal(x, y)`
Resultante $\mathrm{Res}_x(p,q)$	`p.resultant(q, x)`
Dimension	`J.dimension()`
Art	`J.genus()`

Dimension null (endliche Lösungsmenge)	
Lösungen in $L \supseteq K$	`J.variety(L)`
Dimension des Quotienten auf $\mathbb{K}$	`J.vector_space_dimension()`
Basis des Quotienten	`J.normal_basis()`
Dreieckszerlegung	`J.triangular_decomposition()`

Tab. 9.4 – Lösung polynomialer Gleichungssysteme.

Dann können wir verifizieren, dass jede „partielle Lösung“ $(1 - 3z, z)$ zu einer (eindeutigen) Lösung von (9.4) $\left(\frac{5z-1}{2}, 1 - 3z, z\right)$ erweitert wird. Das illustriert, dass der Gauß-Algorithmus lineare Systeme durch Elimination löst, im Gegensatz beispielsweise zu den Formeln von Cramer.

Eliminationsideale. Im Kontext der polynomialen Systeme sind die „Folgen“ der Gleichungen $p_1(\boldsymbol{x}) = \ldots = p_s(\boldsymbol{x}) = 0$ die Elemente des Ideals $\langle p_1, \ldots, p_s \rangle$. Wenn J ein Ideal von $\mathbb{K}[x_1, \ldots, x_n]$ ist, dann nennen wir die Menge

$$J_k = J \cap \mathbb{K}[x_{k+1}, \ldots, x_n] \tag{9.5}$$

der Elemente von J, wobei nur die letzten $n-k$ Variablen auftreten, das k-te *Eliminationsideal* von J.

In Sage wird der Methode `elimination_ideal` die Liste der zu eliminierenden Variablen übergeben. Achtung: sie gibt nicht $J_k \subset \mathbb{K}[x_{k+1}, \ldots, x_v]$ zurück, sondern das Ideal $\langle J_k \rangle$ von $\mathbb{K}[x_{k+1}, \ldots, x_v]$, das von diesem erzeugt wird. Für das lineare System (9.4) findet man

```
sage: R.<x,y,z> = QQ[]
sage: J = ideal(2*x+y-2*z, 2*x+2*y+z-1)
sage: J.elimination_ideal(x)
Ideal (y + 3*z - 1) of Multivariate Polynomial Ring in x, y, z
over Rational Field
sage: J.elimination_ideal([x,y])
Ideal (0) of Multivariate Polynomial Ring in x, y, z over Rational Field
```

Mathematisch interpretieren wir diese Resultate so: wir haben $J \cap \mathbb{Q}[y,z] = \langle y + 3z - 1\rangle \subset \mathbb{Q}[y,z]$ und $L \cap \mathbb{Q}[z] = \mathbb{Q}[z]$, d.h. $\mathbb{Q}[z] \subset J$. (Tatsächlich entspricht das Ideal von $\langle 0\rangle$ dem reduzierten System mit der einzigen trivialen Gleichung $0 = 0$, von der jedes Polynom eine Lösung ist.) Freilich ist das keine empfehlenswerte Vorgehensweise zur Lösung eines linearen Systems: die in Kapitel 8 diskutierten spezifischen Werkzeuge sind viel effizienter!

Als weniger triviales Beispiel greifen wir das System (S_1) aus Unterabschnitt 9.2.3 wieder auf (siehe Abb. 9.1 (a)):

```
sage: R.<x,y> = QQ[]
sage: J1 = ideal(x^2 + y^2 - 1, 16*x^2*y^2 - 1)
```

Die Eliminierung von y liefert ein Ideal von $\mathbb{Q}[x]$ - daher Haupideal -, das von einem Polynom g erzeugt wird. Dessen Wurzeln sind die Abszissen

$$\frac{\pm\sqrt{2 \pm \sqrt{3}}}{2}$$

der acht Lösungen von (S_1):

```
sage: g = J1.elimination_ideal(y).gens(); g
[16*x^4 - 16*x^2 + 1]
sage: SR(g[0]).solve(SR(x)) # gelöst durch Radikale
[x == -1/2*sqrt(sqrt(3) + 2), x == 1/2*sqrt(sqrt(3) + 2),
x == -1/2*sqrt(-sqrt(3) + 2), x == 1/2*sqrt(-sqrt(3) + 2)]
```

Durch Wiedereinsetzten jedes der zu (S_1) gefundenen x-Werte erhalten wir ein (redundantes) Gleichungssystem nur in y, welches die entsprechenden beiden Werte von y zu berechnen erlaubt.

Eliminieren = Projizieren. Das vorstehende Ergebnis illustriert, dass die Elimination von y aus einem Gleichungssystem geometrisch der *Projektion* π der Lösung der Varietät auf einer Hyperebene der Gleichung y =const. entspricht. Aber betrachten wir getrennt davon die ideale $C = \langle x^2 + y^2 - 1\rangle$ und $H = \langle 16x^2y^2 - 1\rangle$, deren J_1 die Summe ist, und eliminieren wir y erneut:

```
sage: C.elimination_ideal(y).gens()
[0]
sage: H.elimination_ideal(y).gens()
[0]
```

Was C betrifft, ist das keine Überraschung. Der Kreis $\left\{(x,y) \in \mathbb{R}^2 \mid x^2 + y^2 = 1\right\}$ wird zwar auf $[-1,1]$ projiziert, doch es ist klar, dass es nicht darauf ankommt, welcher x-Wert in die eindeutige Gleichung $x^2 + y^2 - 1 = 0$ wieder eingesetzt wird und dass die erhaltene Gleichung in y *komplexe* Lösungen hat. Was die Eliminierung von y in C überträgt, ist die Projektion

auf die erste Koordinate des komplexen Kreises $\{(x,y) \in \mathbb{C}^2 \mid x^2+y^2=1\}$, und das ist das ganze $\mathbb{C}$.

Die Situation bei H ist ein klein wenig komplizierter. Die Gleichung $16x^2y^2 = 1$ hat keine, auch keine komplexe Lösung. Diesmal haben wir

$$V_{\mathbb{C}}(H \cap \mathbb{Q}[x]) = \mathbb{C} \subsetneq \pi(V_{\mathbb{C}}(H)) = \mathbb{C}\setminus\{0\}\,.$$

Tatsächlich ist die Projektion der Hyperbel $\mathbb{C}\setminus\{0\}$ keine algebraische Subvarietät. Die Moral von der Geschicht': die Elimination entspricht wohl der Projektion (auf einen algebraisch abgeschlossenen Körper), doch berechnet sie nicht die exakte Projektion einer affinen Varietät, sondern nur deren Zariski-Abschluss.

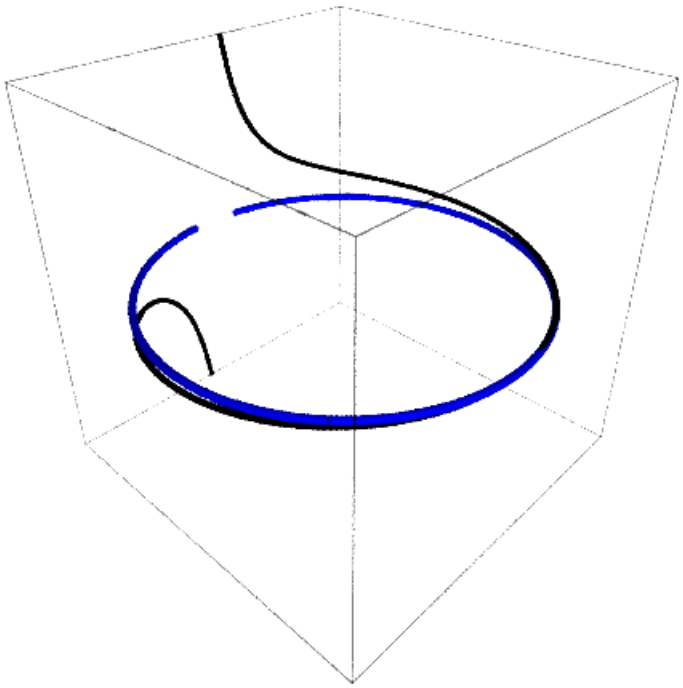


Abb. 9.2 - Ein Teil der durch (9.6) definierten Kurve in (x,y,t) und ihre Projektion auf die Ebene $t=0$.

Anwendungen: ein wenig Geometrie der Ebene. Wenn $X \subset \mathbb{C}^k$ durch eine rationale Parametrisierung gegeben ist:

$$X = \{(f_1(\boldsymbol{t}), f_2(\boldsymbol{t}), \ldots, f_k(\boldsymbol{t}))\}\,, \qquad f_1,\ldots,f_k \in \mathbb{Q}(t_1,\ldots,t_n),$$

soll eine implizite Gleichung für X gefunden werden, die den Teil von $\mathbb{C}^{k+n}$ auf den Unterraum von $(x_1,\ldots,x_k) \simeq \mathbb{C}^k$ projiziert, der durch die Gleichungen $x_i = f_i(\boldsymbol{t})$ definiert ist. Das ist eine Aufgabe für die Elimination. Betrachten wir die klassische Parametrisierung des Kreises

$$x = \frac{1-t^2}{1+t^2} \qquad y = \frac{2t}{1+t^2} \tag{9.6}$$

der dem Ausdruck von $(\sin\theta, \cos\theta)$ als Funktion von $\tan(\theta/2)$ zugeordnet ist. Sie wird durch polynomiale Beziehungen übertragen, die ein Ideal von $\mathbb{Q}[x,y,t]$ definieren:

```
sage: R.<x,y,t> = QQ[]
sage: Param = R.ideal((1-t^2)-(1+t^2)*x, 2*t-(1+t^2)*y)
```

Eliminieren wir t:

```
sage: Param.elimination_ideal(t).gens()
```

Wir erhalten eine Kreisgleichung. Wir können bemerken, dass diese Gleichung in $(x, y) = (-1, 0)$ null wird, obwohl die Parametrisierung (9.6) diesen Punkt nicht enthält, denn der Kreis, der einen Punkt nicht enthält, ist keine algebraische Subvarietät.

Ein anderes Beispiel: Zeichnen wir mit Sage (siehe Abb. 9.3) einige Kreise $(\mathcal{C}_t)$ der Gleichung

$$\mathcal{C}_t : \quad x^2 + (y - t)^2 = \frac{t^2 + 1}{2} \tag{9.7}$$

```
sage: R.<x,y,t> = QQ[]
sage: eq = x^2 + (y-t)^2 - 1/2*(t^2+1)
sage: fig = add((eq(t=k/5)*QQ[x,y]).plot() for k in (-15..15))
sage: fig.show(aspect_ratio=1, xmin=-2, xmax=2, ymin=-3, ymax=3)
```

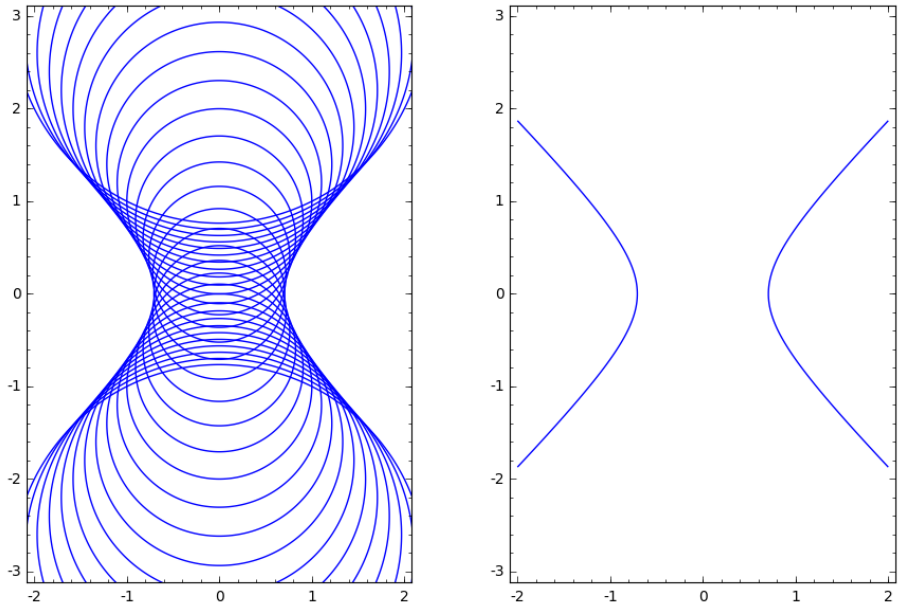


Abb. 9.3 - Eine Familie von Kreisen und ihre Evolvente

Wir sehen die *Evolvente* der Familie der Kreise zum Vorschein kommen, eine „Grenzkurve“, die alle $\mathcal{C}_t$ berührt, die man informell als Menge der „Schnittpunkte der unendlich nahe angenäherten Kreise“ der Familie beschreiben kann.

Genauer, wenn f eine differenzierbare Funktion ist und die Kurve $\mathcal{C}_t$ hat für alle t die Gleichung $f(x, y, t) = 0$, dann ist die Evolvente von $(\mathcal{C}_t)$ die Menge der (x, y), für die gilt

$$\exists t, \qquad f(x, y, t) = 0 \quad \text{und} \quad \frac{\partial f}{\partial t}(x, y, z) = 0. \tag{9.8}$$

Bei den Kreisen (9.7) ist die Funktion $f(x, y, z)$ ein Polynom. Ihre Evolvente ist die Projektion auf die x, y-Ebene der Lösungen von (9.8), weshalb wir durch Berechnung des Eliminationsideals folgende Gleichung bestimmen können:

```
sage: env = ideal(eq, eq.derivative(t)).elimination_ideal(t)
sage: env.gens()
[2*x^2 - 2*y^2 - 1]
```

Die gefundene Kurve ist nur noch zu zeichnen:

```
sage: env.change_ring(QQ[x,y]).plot((x,-2,2), (y,-3,3))
```

Resultante und Elimination. Die Operationen der Elimination in den vorstehenden Beispielen stützen sich implizit auf Gröbnerbasen, die von Sage automatisch berechnet werden. Wir sind in diesem Buch jedoch zuvor schon einem anderen Werkzeug für die Elimination begegnet: der Resultanten.

Ungleichungen

Betrachten wir ein Dreieck mit den Ecken $A = (0,0), B = (1,0)$ und $C = (x,y)$. Setzen wir die Winkel $\angle BAC = \hat{A}$ und $\angle CBA = \hat{B}$ als gleich voraus und versuchen wir durch Rechnung zu beweisen, dass das Dreieck dann gleichschenklig ist. Nach Einführung des Parameters $t = \tan\hat{A} = \tan\hat{B}$ wird die Situation durch die Gleichungen $y = tx = t(1-x)$ kodiert und dann muss gezeigt werden, dass aus ihnen folgt

$$x^2 + y^2 = (1-x)^2 + y^2.$$

Mit Sage bekommen wir

```
sage: R.<x,y,t> = QQ[]
sage: J = (y-t*x, y-t*(1-x))*R
sage: (x^2+y^2) - ((1-x)^2+y^2) in J
False
```

Und das aus gutem Grund: für $x = y = t =$ sind zwar die Voraussetzungen erfüllt, aber die Konklusion ist falsch! Geometrisch gesehen müssen wir den Fall ebener Dreiecke ausschließen, die zwei gleiche Winkel haben können, ohne gleichschenklig zu sein.

Wie ist beispielsweise die Bedingung $t \neq 0$ zu kodieren? Der Trick besteht darin, eine Hilfsvariable u einzuführen und $tu = 1$ vorzuschreiben. Die Rechnung wird dann:

```
sage: R.<x,y,t,u> = QQ[]
sage: J = (y-t*x, y-t*(1-x), t*u-1)*R
sage: (x^2+y^2) - ((1-x)^2+y^2) in J
True
```

und diesmal erhalten wir das erwartete Ergebnis. Nebenbei bemerken wir, dass wir durch eine Gleichung des Typs $t_1 t_2 \cdots t_n u = 1$ mehrere Ausdrücke gleichzeitig verarbeiten können, damit keine einzige Hilfsvariable null wird.

Betrachten wir zwei nicht konstante Polynome $p, q \in \mathbb{K}[x_1, \ldots, x_n, y]$. Wir bezeichnen mit $\mathrm{Res}_y(p,q)$ die Resultante von p und q, die als Polynome in einer Unbestimmten y angesehen werden. In Unterabschnitt 7.3.3 haben wir gesehen, dass es ein Polynom von $\mathbb{K}[x_1, \ldots, x_n]$ gibt, das in $\boldsymbol{u} \in \mathbb{K}^n$ genau dann zu null wird, wenn $p(u_1, \ldots, u_n, y)$ und $q(u_1, \ldots, u_n)$ (zwei

Polynome aus $\mathbb{K}[y]$) eine Nullstelle gemeinsam haben, außer vielleicht wenn die Leitkoeffizienten (in y) von p in $\boldsymbol{u}$ selbst null sind. Wir können noch mehr sagen: in der Tat erzeugt $\mathrm{Res}_y(p,q)$ das Eliminationsideal[7] $\langle p,q\rangle \cap \mathbb{K}[x_1\ldots,x_n]$.

Somit können alle Berechnungen zur Elimination, die wir zu zwei Polynomen angestellt haben, durch die Resultanten ersetzt werden. Beispielsweise ist die Gleichung der Evolvente der Kreise (9.7)

```
sage: eq.derivative(t).resultant(eq, t)
x^2 - y^2 - 1/2
```

Wir werden hier bei zwei Polynomen bleiben. Es ist möglich, das Ergebnis oder seine Verallgemeinerungen für allgemeinere Aufgaben der Eliminierung zu verwenden, jedoch ist die Theorie komplizierter und die entsprechenden Werkzeuge stehen in Sage noch nicht zur Verfügung.

9.2.5. Systeme der Dimension null

Wir können die Probleme mit der alleinigen Berechnung der Eliminationsideale lösen, und Sage bietet kaum anderes „black box“ - Werkzeug zur Lösung allgemeiner polynomialer Systeme. Die Situation ist anders, wenn es um Systeme mit Dimension null geht.

Wir sagen, ein Ideal $J \subset \mathbb{K}[\boldsymbol{x}]$ hat die Dimension null, wenn der Quotientenring $\mathbb{K}[\boldsymbol{x}]/J$ ein endlichdimensionaler Vektorraum ist. Auf einem *algebraisch abgeschlossenen Körper* ist das äquivalent zu der Aussage, dass die Varietät $V(J)$ von einer endlichen Anzahl von Punkten gebildet wird. Somit erzeugen die Systeme (9.1) und (9.3) Ideale der Dimension null - wir sagen auch, dass sie ihrerseits die Dimension null haben. Andererseits hat das Ideal $\langle (x^2+y^2)(x^2+y^2+1)\rangle$ von $\mathbb{Q}[x,y]$ die Dimension 1, obwohl die einzige reelle Lösung $(0,0)$ ist:

```
sage: R.<x,y> = QQ[]
sage: ((x^2 + y^2)*(x^2 + y^2 + 1)*R).dimension()
1
```

Systeme der Dimension null werden klarer gelöst als diejenigen, welche die allgemeinen Werkzeuge des vorigen Abschnitts zulassen. Wir haben bei der Arbeit mit dem Beispiel in Unterabschnitt 9.2.1 bereits mehrere dieser Möglichkeiten gesehen.

Auflisten der Lösungen. Zunächst ermöglicht die Tatsache, nur eine endliche Anzahl von Lösungen zu haben, ihre Auflistung entweder exakt oder näherungsweise.

Der Sage-Ausdruck `J.variety(L)` dient zur Berechnung der Varietät $V_L(J)$. Er zeitigt einen Fehler, wenn J nicht die Dimension null hat. In der Voreinstellung sucht er die Lösungen des Systems mit Koordinaten im allgemeinen Basiskörper des Polynomrings. Beispielsweise ist die durch J_1 definierte Subvarietät von $\mathbb{Q}^n$ leer.

```
sage: J1 = (x^2 + y^2 - 1, 16*x^2*y^2 - 1)*R
sage: J1.variety()
[]
```

[7] Die Punkte, an denen die Leitkoeffizienten verschwinden und die Eigenschaft der Spezialisierung der Resultanten nicht gilt, sind diejenigen, die bei der Projektion durch den Übergang zum Zariski-Abschluss hinzugefügt worden sind.

Doch ebenso wie die Methode `roots` der Polynome mit einer Unbestimmten funktioniert `variety` bei allen Arten von Definitionsmengen L. Für uns ist der wichtigste der Körper der algebraischen Zahlen. Man kann tatsächlich beweisen, dass die Lösungen eines Systems der Dimension null mit Koeffizienten in $\mathbb{K}$ ihrerseits Koordinaten im algebraischen Abschluss von $\mathbb{K}$ besitzen. Somit ist es möglich, die einem Ideal $J \subset \mathbb{Q}[\boldsymbol{x}]$ zugeordnete komplexe Varietät $V_{\mathbb{C}}(J) = V_{\bar{\mathbb{Q}}}(J)$ direkt zu berechnen;

```
sage: J1.variety(QQbar)[0:2]
[{y: -0.9659258262890683?, x: -0.2588190451025208?},
 {y: -0.9659258262890683?, x: 0.2588190451025208?}]
```

Übung 35. Es ist zu beweisen, dass die Lösungen von (S_1) Koordinaten in $\mathbb{Q}[\sqrt{2-\sqrt{3}}]$ haben. Sie sind mit Radikalen anzugeben.

Dreieckszerlegung. Intern geht `J.variety(L)` über die Berechnung einer *Dreieckszerlegung* des Ideals J. Diese Zerlegung ist selbst schon interessant, denn sie ergibt für den Fortgang der Rechnung oft eine bequemere Beschreibung der Varietät $V(J)$, die nämlich einfacher zu interpretieren ist als die Ausgabe von `variety` (siehe Unterabschnitt 9.2.1), besonders dann, wenn die Lösungen zahlreich sind.

Ein polynomiales System heißt dreieckig, wenn es die Form

$$\begin{cases} p_1(x_1) & := x_1^{d_1} + a_{1,d_1} x_1^{d_1-1} + \ldots + a_{1,0} & = 0 \\ p_(x_1, x_2) & := x_2^{d_2} + a_{2,d_2-1}(x_1) x_2^{d_2-1} + \ldots + a_{2,0}(x_1) & = 0 \\ & \vdots & \\ p_n(x_1, \ldots, x_n) & := x_n^{d_n} + a_{n,d_n-1}(x_1, \ldots, x_{n-1}) x_n^{d_n-1} + \ldots & = 0 \end{cases}$$

hat, anders gesagt, wenn in jedem Polynom p_i nur Variablen $x_1, \ldots, x_i$ vorkommen. und wenn es in der Variablen x_i unitär ist. Liegt ein System der Dimension null in dieser Gestalt vor, wird seine Lösung auf eine endliche Anzahl polynomialer Gleichungen mit einer Variablen zurückgeführt: es genügt, die Wurzeln x_1 von p_1 zu finden, sie in p_2 einzusetzen, dann die Wurzeln x_2 von p_2 zu suchen und so weiter. Diese Strategie funktioniert bei der Suche sowohl nach exakten wie auch genäherten Lösungen.

Nicht jedes System ist zu einem Dreieckssystem äquivalent. Es sei zum Beispiel das Ideal J definiert durch:

```
sage: R.<x,y> = PolynomialRing(QQ, order='lex')
sage: C = ideal(x^2+y^2-1)
sage: D = ideal((x+y-1)*(x+y+1))
sage: J = C + D
```

Als Bild (siehe Abb. 9.4):

```
sage: opts = {'axes':True, 'gridlines':True, 'frame':False,
'aspect_ratio':1, 'axes_pad':0, 'xmin':-1.3, 'xmax':1.3,
'ymin':-1.3, 'ymax':1.3, 'fontsize': 8}
show(C.plot() + D.plot(), figsize=[2,2], **opts)
```

Die Varietät $V(J)$ enthält zwei Punkte mit der Abszisse 0, doch nur einen mit der Abszisse -1, und ebenso einen Punkt mit der Ordinate -1, gegenüber zwei Punkten mit der Ordinate null. Das Ideal J kann deshalb nicht durch ein System in Dreiecksform beschrieben werden.

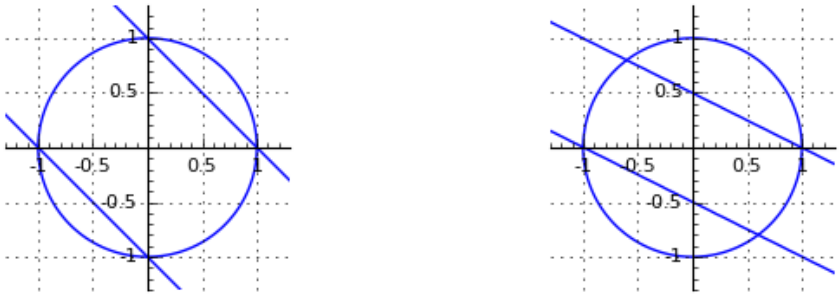


Abb. 9.4 - In beiden Fällen ist die dem Ideal J zugeordnete Varietät der Schnitt eines Kreises mit der Vereinigung zweier Geraden.

Man kann auf der anderen Seite zeigen, dass jedes Ideal der Dimension null als endliche Schnittmenge von den Dreieckssystemen erzeugten Ideale geschrieben wird. Die Methode `triangular_decomposition` berechnet eine solche Zerlegung:

```
sage: J.triangular_decomposition()
 [Ideal (y, x^2 - 1) of Multivariate Polynomial Ring in x, y
over Rational Field,
Ideal (25*y^2 - 16, 4*x + 3*y) of Multivariate Polynomial Ring in x, y
over Rational Field]
```

Geometrisch erhalten wir eine Darstellung der Varietät als Vereinigung der den einfacheren Systemen zugeordneten Varietäten, die oft einfach genug sind, um eine gute Beschreibung der Lösungen zu formulieren.

Etliche Schwierigkeiten Zu Recht können wir uns fragen, welche Bedeutung die Dreieckszerlegung für die Auflistung der Lösungen hat. Nach all dem ist es bei einem System der Dimension null immer möglich, durch eine Berechnung des Eliminationsideals ein Polynom mit einer einzigen Variablen zu finden, dessen Wurzeln exakt die ersten Koordinaten der Lösungen sind. Durch Rückwärts-Einsetzen dieser Wurzeln in das Gleichungssystem vermindern wir die Anzahl der Variablen, was erlaubt, den Vorgang zu wiederholen, bis wir das System vollständig gelöst haben.

Doch der „Aufstieg“ im System durch Verbreitung der Teilergebnisse kann heikel sein. Modifizieren wir das obige System ein wenig:

```
sage: D = ideal((x+2*y-1)*(x+2*y+1)); J = C + D
sage: J.variety()
[{y: -4/5, x: 3/5}, {y: 0, x: -1}, {y: 0, x: 1}, {y: 4/5, x: -3/5}]
sage: [T.gens() for T in J.triangular_decomposition()]
[[y, x^2 - 1], [25*y^2 - 16, 4*x + 3*y]]
```

Die Form der Dreieckszerlegung verändert sich nicht: für jede Komnponente verfügen wir über eine Gleichung nur in y und eine andere Gleichung, die erlaubt, x als Funktion von y auszudrücken.

Eliminieren wir deshalb x, um eine Gleichung in y allein zu erhalten, wobei das Produkt der beiden vorhergehenden auftritt:

```
sage: Jy = J.elimination_ideal(x); Jy.gens()
[25*y^3 - 16*y]
```

Um x zu finden, gibt es nun nichts weiter einzusetzen als die Wurzeln dieser Gleichung in die Gleichungen, die das Ideal J definieren. Mit der ersten Gleichung $x^2 + y^2 - 1 = 0$ kommt

```
sage: ys = QQ['y'](Jy.0).roots(); ys
[(4/5, 1), (0, 1), (-4/5, 1)]
sage: QQ['x'](J.1(y=ys[0][0])).roots()
[(-3/5, 1), (-13/5, 1)]
```

Einer der beiden erhaltenen Werte ist korrekt - wir haben $(-3/5, 4/5) \in V(J)$ - doch der andere entspricht keiner Lösung: wir müssen die gefundenen Werte mittels der zweiten Ausgangsgleichung $(x + 2y - 1)(x + 2y + 1)$ überprüfen, um ihn auszuschließen.

Das Problem kompliziert sich, wenn wir die univariaten Gleichungen numerisch lösen, was wegen des Aufwands bei algebraischen Zahlen zuweilen erforderlich ist:

```
sage: ys = CDF['y'](Jy.0).roots(); ys
[(-0.8000000000000002, 1), (0.0, 1), (0.8, 1)]
sage: [CDF['x'](p(y=ys[0][0])).roots() for p in J.gens()]
[[(-0.5999999999999999 - 1.306289919090511e-16*I, 1),
  (0.6000000000000001 + 1.3062899190905113e-16*I, 1)],
 [(0.6000000000000001 - 3.1350958058172247e-16*I, 1),
  (2.600000000000001 + 3.135095805817224e-16*I, 1)]]
```

Durch Einsetzen von $y \simeq -0.8$ in die beiden Erzeugenden von J finden wir hier zwei Werte von x nahe 0.6. Wie können wir sicher sein, dass dies Näherungen der Koordinate x einer exakten Lösung $(x, y) = (0.6, -0.8)$ sind und keine Gespenster wie im vorigen Beispiel? Diese Erscheinungen verstärken sich, wenn die Anzahl der Variablen und der Gleichungen zunimmt. Wenn das System andererseits dreieckig ist, gibt es bei jedem Schritt des Aufstiegs nur eine Gleichung zu betrachten, und da diese immer unitär ist, beeinflussen die numerischen Näherungen die Anzahl der Lösungen nicht.

Fahren wir in dieser Richtung fort. Für das folgende System berechnet `J.variety()` (exakt) eine Dreieckszerlegung von J und sucht dann numerisch die reellen Lösungen des oder der erhaltenen Systeme. Das ergibt eine eindeutige reelle Lösung:

```
sage: R.<x,y> = QQ[]; J = ideal([ x^7-(100*x-1)^2, y-x^7+1 ])
sage: J.variety(RealField(51))
[{y: 396340.890166545, x: -14.1660266425312}]
```

Führt man jedoch die Rechnung konsequent exakt aus, sieht man, dass nicht nur Lösungen fehlen, sondern die numerisch gefundene Lösung auch fehlerhaft ist:

```
sage: J.variety(AA)
[{x: 0.009999999000000035?, y: -0.999999999999990?},
{x: 0.010000001000000035?, y: -0.999999999999990?},
{x: 6.3055689986413 85?, y: 396340.8901665450?}]
```

Moral: die Dreieckszerlegung ist kein Allheilmittel und entbindet den Anwender nicht davon, die Ergebnisse von Näherungsrechnungen mit Sorgfalt zu interpretieren.

Es existiert eine große Anzahl weiterer Verfahren, um Lösungen von Systemen der Dimension null zu parametrieren und anzunähern, die den Problemen mehr oder weniger gut angepasst sind, die in Sage aber nicht implementiert sind. Übung 36 gibt einen Einblick in bestimmte dabei verwendete Ideen.

Quotientenalgebra. Quotienten durch Ideale der Dimension null sind im Allgemeinen viel besser handhabbar als die Quotienten von Polynomringen, denn Rechnungen in der Quotientenalgebra werden auf endlichdimensionale lineare Algebra zurückgeführt.

Anspruchsvollere Mathematik

Sage verfügt auch über eine große Zahl von Funktionen der kommutativen Algebra und der algebraischen Geometrie, die das Niveau dieses Buches jedoch weit übersteigen. Der interessierte Leser ist eingeladen, die Dokumentation der Ideale von Polynomen zu erkunden wie auch diejenige des Moduls `sage.schemes`. Über das Interface zu den spezialisierten Programmen Singular, CoCoA und Macauley2 sind noch weitere Funktionalitäten zugänglich.

Wenn $J \subset \mathbb{K}[\boldsymbol{x}]$ ein Ideal der Dimension null ist, dann ist die Dimension $\dim \mathbb{K}[\boldsymbol{x}]/J$ des Quotientenrings als $\mathbb{K}$-Vektorraum eine Schranke für die Anzahl der Punkte von $V(J)$. (Tatsächlich existiert zu jedem $u \in V(J)$ ein Polynom mit Koeffizienten aus $\mathbb{K}$, das in jedem anderen Punkt von V den Wert 0 oder 1 annimmt. Zwei dieser Polynome können nicht äquivalent sein modulo J.) Man kann sich diese Dimemsion als Anzahl der Lösungen des Systems „mit Multiplizitäten“ im algebraischen Abschluss von $\mathbb{K}$ vorstellen. Beispielsweise hatten wir festgestellt, dass die vier Lösungen des in Unterabschnitt 9.2.3 eingeführten Systems (S_2) jeweils die „doppelte“ Schnittmenge der beiden Kurven sind. Das erklärt, weshalb gilt:

```
sage: len(J2.variety(QQbar)), J2.vector_space_dimension()
(4, 8)
```

Die Methode `normal_basis` berechnet eine Liste von Monomen, deren Projektionen in $\mathbb{K}[\boldsymbol{x}]/J$ eine Basis bilden:

```
sage: J2.normal_basis()
[x*y^3, y^3, x*y^2, y^2, x*y, y, x, 1]
```

Die zurückgegebene Basis hängt von der Ordnung ab, die bei der Bildung des Polynomrings gewählt worden ist; genauer beschreiben wir sie in Unterabschnitt 9.3.3.

Übung 36. Sei J ein Ideal der Dimension null aus $\mathbb{Q}[x,y]$ und χ_x das charakteristische Polynom der linearen Abbildung

$$\begin{aligned} m_x : \ \mathbb{Q}[x,y]/J &\to \mathbb{Q}[x,y]/J \\ &\mapsto xp + J. \end{aligned}$$

Berechnen Sie χ_x für den Fall $J = J_2 = \langle x^2 + y^2 - 1, 4x^2y^2 - 1\rangle$. Zeigen Sie, dass jede Wurzel die Abszisse eines Punktes der Varietät $V_{\mathbb{C}}(J)$ ist.

9.3. Gröbnerbasen

Bis jetzt haben wir die Funktionalitäten der algebraischen Elimination und der Lösung polynomialer Systeme benutzt, die Sage als *black box* anbietet. Dieser Abschnitt führt nun einige mathematische Werkzeuge und ihnen zugrunde liegende Algorithmen ein. Das Ziel ist, von ihnen Gebrauch zu machen und gleichzeitig Funktionen, die zuvor vorgesteltt worden sind, auf höherem Niveau klug anzuwenden.

Wichtige monomiale Ordnungen für das Beispiel $\mathbb{Q}[x,y,z]$	
`lex`	$x^\alpha < x^\beta \iff \alpha_1 < \beta_1$ oder $(\alpha_1 = \beta_1$ und $\alpha_2 < \beta_2)$ oder ... oder $(\alpha_1 = \beta_1, \ldots, \alpha_{n-1} = \beta_{n-1}$ und $\alpha_n < \beta_n)$ $x^3 > x^2y > x^2z > x^2 > xy^2 > xyz > xy > xz^2 > xz > x > y^3$ $> y^2z > y^2 > yz^2 > yz > y > z^3 > z^2 > z > 1$
`invlex`	$x^\alpha < x^\beta \iff \alpha_n < \beta_n$ oder $(\alpha_n = beta_n$ und $\alpha_{n-1} < \beta_{n-1})$ oder ... oder $(\alpha_n = \beta_n, \ldots, \alpha_2 = \beta_2$ und $\alpha_1 < \beta_1)$ $z^3 > yz^2 > xz^2 > z^2 > y^2z > xyz > yz > x^2z > xz > z > y^3$ $> xy^2 > y^2 > x^2y > xy > y > x^3 > x^2 > x > 1$
`deglex`	$x^\alpha < x^\beta \iff \lvert\alpha\rvert < \lvert\beta\rvert$ oder $(\lvert\alpha\rvert = \lvert\beta\rvert$ und $x^\alpha <_{\text{lex}} x^\beta)$ $x^3 > x^2y > xy^2 > xyz > xz^2 > y^3 > y^2z > yz^2 > z^3 > x^2$ $> xy > xz > y^2 > yz > z^2 > x > y > z > 1$
`degrevlex`	$x^\alpha < x^\beta \iff \lvert\alpha\rvert < \lvert\beta\rvert$ oder $(\lvert\alpha\rvert = \lvert\beta\rvert$ und $x^\alpha >_{\text{invlex}} x^\beta)$ $x^3 > x^2y > xy^2 > y^3 > x^2z > xyz > y^2z > xz^2 > yz^2 > z^3 > x^2$ $> xy > y^2 > xz > yz > z^2 > x > y > z > 1$

Bildung der monomialen Ordnungen	
Objekt, das eine vordefinierte Ordnung auf n Elementen darstellt	`TermOrder('nom', n)`
Matrizenordnung: $x^\alpha <_M x^\beta \iff M\alpha <_{textlex} M\beta$	`TermOrder(M)`
Blöcke: $x^\alpha y^\beta < x^\gamma y^\delta \iff \alpha <_1 \gamma$ oder $(\alpha = \gamma$ und $\beta <_2 \delta)$	`T1 + T2`

Tab. 9.5 - Monomiale Ordnungen

Die von Sage für Rechnungen auf Eliminationsidealen angewandten Techniken beruhen auf dem Begriff der Gröbnerbasis. Wir können sie uns als Erweiterung der Darstellung durch den Hauptgenerator der Ideale von $\mathbb{K}[x]$ auf mehrere Unbestimmte vorstellen. Das zentrale Problem dieses Abschnitts sind die Definition und die Berechnung einer Normalform der Elemente der Quotientenringe $\mathbb{K}[\boldsymbol{x}]$. Unser Standpunkt bleibt der des Anwenders.: wir defineren die Gröbnerbasen, zeigen, wie wir sie in Sage bekommen und wozu sie dienen können, wir gehen aber nicht auf die zur Berechnung verwendeten Algorithmen ein.

9.3.1. Monomiale Ordnungen

Eine *monomiale Ordnung* oder (globale) *zulässige Ordnung* ist eine totale Ordnung auf den Monomen x^α eines Polynomrings $\mathbb{K}[\boldsymbol{x}]$, die folgenden Bedingungen genügt:

$$\boldsymbol{x^\alpha < x^\beta} \implies \boldsymbol{x^{\alpha+\gamma} < x^{\beta+\gamma}} \quad \text{und} \quad \boldsymbol{\gamma \neq 0} \implies 1 < \boldsymbol{x^\gamma} \tag{9.9}$$

für alle Exponenten $\boldsymbol{\alpha, \beta, \gamma}$. Auf die gleiche Weise können wir $<$ als eine Ordnung auf den Exponenten $\boldsymbol{\alpha} \in \mathbb{N}^n$ betrachten oder auch auf den Termen $c\boldsymbol{x^\alpha}$. Die Leitmonome, Leitkoeffizienten und Leitterme eines Polynoms p (siehe Unterabschnitt 9.1.3) für die vorliegende Ordnung sind die mit den größten Exponenten; wir bezeichnen sie mit $\operatorname{lm} p$, $\operatorname{lc} p$ und $\operatorname{lt} p$.

Die erste der Aussagen (9.9) drückt eine Bedingung der Kompatibilität mit der Multiplikation aus: das Produkt mit einem festen Monom verändert die Ordnung nicht. Die zweite Aussage

bestimmt, dass $<$ eine Wohlordnung ist, d.h. dass keine streng abnehmende endliche Folge von Monomen existiert. Wir stellen fest, das die allein zulässige Ordnung auf $\mathbb{K}[x]$ die übliche ist mit $x^n > x^{n-1} > \ldots > 1$.

In Unterabschnitt 9.1.1 haben wir gesehen, dass Sage bei der Definition eines Ringes die Festlegung einer Ordnung erlaubt, und zwar mit einer Konstruktion wie

```
sage: R.<x,y,z,t> = PolynomialRing(QQ, order='lex')
```

Tabelle 9.5 verzeichnet die wichtigsten vordefinierten Ordnungen[8]: `lex` ist die lexikographische Ordnung der Exponenten, `invlex` die von rechts nach links gelesene lexikographische Ordnung der Exponenten. Die Definition von `degrevlex` ist etwas komplizierter: die Monome sind nach der Summe der Exponenten geordnet und dann in *absteigender* lexikographischer Ordnung der *von rechts gelesenen Exponenten*. Diese merkwürdige Ordnung ist dennoch die, welche angewendet wird, wenn die Option `order` weggelassen wird, denn für bestimmte Rechnungen ist sie effizienter als die anderen.

Gemeinhin (aber nicht immer!) legen wir die Reihenfolge der Variablen und der Monome $x_1 > x_2 > \ldots > x_n$ zusammen fest, und wir sprechen beispielsweise oft von der „Ordnung `lex`, sodass $x > y > z$“ ist statt von der „Ordnung `lex` auf $\mathbb{K}[x, y, z]$“. Die vordefinierten Ordnungen `lex, deglex` und `degrevlex` gehorchen dieser Konvention; was die Ordnung `invlex` auf $\mathbb{K}[x, y, z]$ betrifft, ist das auch die Ordnung `lex`, sodass $z > y > x$, d.h. die Ordnung auf $\mathbb{K}[z, y, x]$.

9.3.2. Division durch eine Familie von Polynomen

Sei $G = \{g_1, g_2, \ldots, g_s\}$ eine endliche Menge von Polynomen von $\mathbb{K}[\boldsymbol{x}]$ mit festgelegter monomialer Ordnung $<$. Mit $\langle G \rangle$ bezeichnen wir das durch G erzeugte Ideal von $\mathbb{K}[\boldsymbol{x}]$.

Die Division eine Polynoms $p \in \mathbb{K}[\boldsymbol{x}]$ durch G ist ein multivariates Analogon zur euklidschen Division in $\mathbb{K}[x]$. Wie letztere ordnet sie p einen Rest zu, der in Sage durch den Ausdruck `p.reduce(G)` gegeben wird. Das ist ein „kleineres“ Polynom, welches zur selben Äquivalenzklasse modulo $\langle G \rangle$ gehört:

```
sage: ((x+y+z)^2).reduce([x-t, y-t^2, z^2-t])
2*z*t^2 + 2*z*t + t^4 + 2*t^3 + t^2 + t
```

Der Rest wird erhalten, indem von p soviele Vielfache der Elemente von G abgezogen werden wie möglich. Bei der Subtraktion heben sich der Leitterm von G und ein Term von p auf. Im Unterschied zum univariaten Fall kann es vorkommen, dass sich ein Term von p heraushebt, der kein Leitterm ist. Wir fordern daher nur, dass ein im Sinne der monomialem Ordnung größter Term zu null wird.

[8] Sage kennt noch weitere Ordnungen (sog. lokale Ordnungen), bei denen 1 das größte Monom ist statt das kleinste zu sein. Beispielsweise haben wir bei der Ordnung `neglex` auf $\mathbb{Q}[x, y, z]$ $1 > z > z^2 > zu^3 > y > yz > yz^2 > y^2 > y^2z > y^3 > x > xz > xz^2 > xy > xyz > xy^2 > x^2 > x^2z > x^2y > x^3$. Die lokalen Ordnungen sind im Sinne der Definition (9.9) nicht zulässig, und wir verwenden sie auch nicht in diesem Buch, doch kann der interessierte Leser die Tabelle 9.5 mit der in Unterabschnitt 9.1.1 definierten Funktion `test_poly` vervollständigen.

Formell bezeichnen wir für $p \in \mathbb{K}[\boldsymbol{x}]$ mit $\mathrm{lt}_G p$ den Term von p mit dem größten Exponenten, der durch einen Leitterm eines Elementes von G teilbar ist. Wir nennen *elementare Reduktion* jede Transformation der Form

$$p \mapsto \tilde{p} = p - c\boldsymbol{x}^{\boldsymbol{\alpha}} g, \quad \text{worin } g \in G \text{ und } \mathrm{lt}_G p = c\boldsymbol{x}^{\boldsymbol{\alpha}} \mathrm{lt} g. \tag{9.10}$$

Eine elementare Reduktion erhält die Äquivalenzklasse von p modulo $\langle G \rangle$ und lässt das größte Monom von p, das sie zuordnet, verschwinden: wir haben

$$\tilde{p} - p \in \langle G \rangle \quad \text{und} \quad \mathrm{lt}_G \tilde{p} < \mathrm{lt}_G p.$$

Da $<$ eine Wohlordnung ist, ist es nicht möglich, unendlich viele elementare Reduktionen nacheinander auf p anzuwenden. Jede Folge elementarer Reduktionen endet deshalb bei einem Polynom, das nicht reduziert werden kann, und das ist der Rest der Division.

Halten wir fest, dass dieses Verfahren die bekannten Eliminationsmethoden sowohl für Polynome mit einer Unbestimmten wir für lineare Systeme verallgemeinert. Gibt es nur eine Variable, reduziert sich die Division durch eine Einsermenge $G = \{g\}$ genau auf die euklidsche Division von p durch g. Im anderen Extremfall eines Polynoms mit mehreren Unbestimmten, wobei jedoch alle Monome vom 1. Grade sind, stimmt sie mit der elementaren Reduktionsoperation des Gauß-Jordan-Verfahrens überein.

Doch anders als das, was in diesen Spezialfällen geschieht, hängt der Rest generell von der Wahl der elementaren Reduktionen ab. (Man sagt, das Regelsystem der Transformation (9.10) ist nicht konfluent.) Somit führt die Änderung der Reihenfolge, in der die Elemente von G gegeben sind, im folgenden Beispiel zur Wahl verschiedener Reduktionen:

```
sage: R.<x,y> = PolynomialRing(QQ, order='lex')
sage: (g, h) = (x-y, x-y^2); p = x*y - x
sage: p.reduce([g, h]) # zwei Reduktionen durch h
y^3 - y^2
sage: p.reduce([h, g]) # zwei Reduktionen durch g
y^2 - y
```

Selbst wenn die Ordnung der Elemente von G als determiniert angesehen wird (sodass für gegebene p und G ein eindeutiges Ergebnis erhalten wird), wie stellt man beispielsweise sicher, dass die Folge der gewählten elementaren Reduktionen von p durch $\{g, h\}$ die folgende Relation zeitigen wird, die beweist, dass $p \in \langle g, h \rangle$ ist?

```
sage: p - y*g + h
0
```

9.3.3. Gröbnerbasen

Die Einschränkungen bei der multivariaten Division erklären die im Unterabschnitt 9.2.3 erwähnten Schwierigkeiten, für die Elemente der Algebren $\mathbb{K}[\boldsymbol{x}]/J$ eine Normalform zu bekommen: die Division durch die Erzeugenden des Ideals reicht nicht aus… zumindest nicht generell! Es existieren nämlich spezielle Generatoren, für welche die Division konfluent ist und eine Normalform berechnet. Diese Systeme heißen *Gröbnerbasen* oder *Standardbasen*.

Treppen. Eine schöne Möglichkeit Gröbner-Bassen kennen zu lernen, ergibt sich durch den Begriff der Treppe eines Ideals. Wir möchten jedes von null verschiedene Polynom aus $\mathbb{K}[x_1, \ldots, x_n]$ auf einen Punkt $\mathbb{N}^n$ abbilden, der durch seinen Leitexponenten gegeben ist, und die Teilmenge $E \subset \mathbb{N}^n$ markieren, die durch ein Ideal J belegt ist (siehe Abb. 9.5 bis 9.7). Die erhaltene Darstellung (die von der monomialen Ordnung abhängt), hat Treppenform: tatsächlich haben wir $\boldsymbol{\alpha} + \mathbb{N}^n \subset E$ für jedes $\boldsymbol{\alpha} \in E$. Die Elemente von $J \setminus \{0\}$ befinden sich im grau unterlegten Bereich oberhalb oder an der Treppenline. Die Punkte streng unterhalb der Treppenlinie entsprechen ausschließlich den Polynomen aus $\mathbb{K}[\boldsymbol{x}] \setminus J$, es liegen jedoch nich alle Polynome aus $\mathbb{K}[\boldsymbol{x}] \setminus J$ unter der Treppenlinie.

Beispielsweise ist jedes Monom in einem Polynom aus $\langle x^3, xy^2z, xz^2 \rangle$, Leitmonom oder nicht, ein Vielfaches eines der Polynome x^3, xy^2z und xz^2. Die Leitmonome sind deshalb genau die $x^\alpha y^\beta z^\gamma$, die Exponent für Exponent einer der Ungleichungen $(\alpha, \beta, \gamma) \geq (3, 0, 0)$, $(\alpha, \beta, \gamma) \geq (1, 2, 1)$ oder $(\alpha, \beta, \gamma) \geq (1, 0, 2)$ genügen (Abb. 9.5). Ein Polynom, dessen Leitexponenten diese Bedingungen nicht erfüllen, zum Beispiel $x^2 + xz^2$ bei lexikographischer Ordnung mit $x > y > z$, kann dem Ideal nicht angehören. Bestimmte Polynome wie $x^3 + x$ sind nicht mehr im Ideal, obwohl ein Monom unterhalb der Treppenlinie liegt. Die Situation ist für jedes von Monomen erzeugte Ideal entsprechend.

Für ein beliebiges Ideal wird die Gestalt der Treppe nicht einfach aus den Erzeugenden abgelesen. Wenn wir also mit $\boldsymbol{\delta}_1 \ldots, \boldsymbol{\delta}_s$ die Leitexponenten der Erzeugenden bezeichnen, haben wir in allen Beispielen der Abb. 9.6 und 9.7 $\bigcup_{i=1}^{s} (\boldsymbol{\delta} + \mathbb{N}^n) \subsetneq E$, außer in der zweiten. Wir können indessen zeigen, dass E immer als Vereinigung einer endlichen Anzahl von Mengen der Form $\boldsymbol{\alpha} + \mathbb{N}^n$ geschrieben wird. Intuitiv heißt das, die Treppenlinie hat nur endlich viele Ecken. Dieses Ergebnis heißt manchmal Lemma von Dickson.

Gröbnerbasen Eine Gröbnerbasis ist schlicht eine Familie von Erzeugenden, die die Form der Treppenlinie ausmachen, genauer, die an jeder ihrer Ecken ein Polynom zählt.

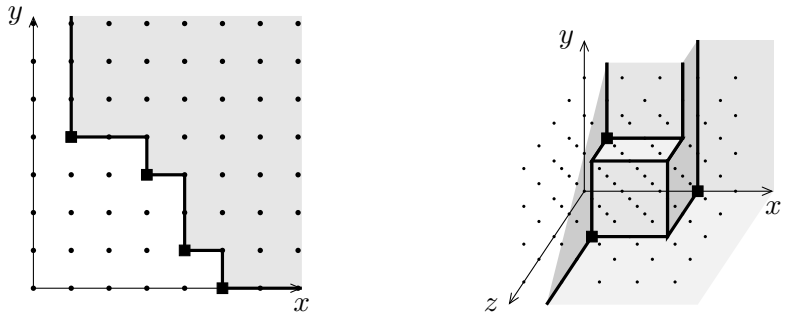
Definition. Eine Gröbnerbasis eines Ideals $J \subset \mathbb{K}[\boldsymbol{x}]$ bezogen auf eine monomiale Ordnung $<$ ist eine endliche Teilmenge G von J, sodass für jedes von null verschiedene $p \in J$ ein $g \in G$ existiert, für dessen Leitmonom (zur Ordnung $<$) gilt: $\operatorname{lm} g$ teilt $\operatorname{lm} p$.

Die Prüfung, ob die Erzeugenden, die ein Ideal definieren, eine Gröbnerbasis bilden, wird in Sage mit der Methode `basis_is_groebner` durchgeführt. Wir haben schon bemerkt, dass jede Menge von Monomen eine Gröbnerbasis ist:

```
sage: R.<x,y> = PolynomialRing(QQ, order='lex')
sage: R.ideal(x*y^4, x^2*y^3, x^4*y, x^5).basis_is_groebner()
True
```

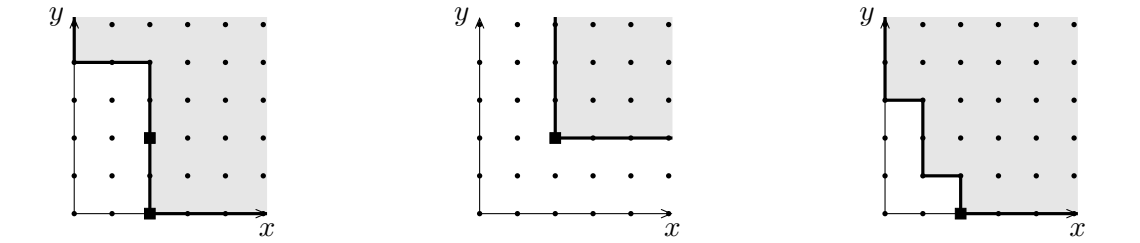
Andererseits ist das System $\{x^2 + y^2 - 1, 16x^2y^2 - 1\}$, das den Schnitt von Kreis und Hyperbeln kodiert, keine Gröbnerbasis:

```
sage: R.ideal(x^2+y^2-1, 16*x^2*y^2-1).basis_is_groebner()
False
```



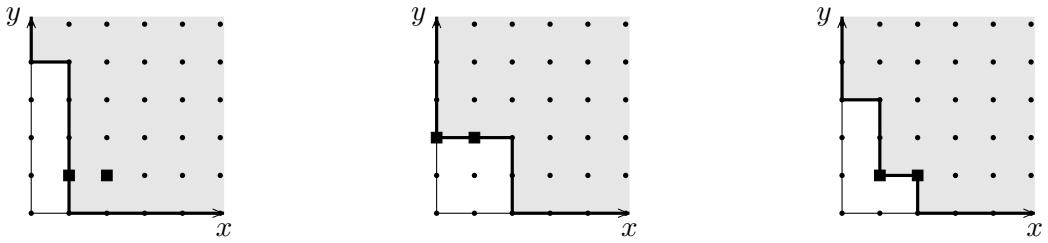
$\langle xy^4, x^3y^3, x^4y, x^5\rangle$ $\langle x^3, xy^2z, xz^2\rangle$

Abb. 9.5 - Treppen der von Monomen erzeugten Ideale.



$\langle x^2+y^2-1, 16\,x^2y^2-1\rangle$ $\langle 16\,x^2y^2-1\rangle$ $\langle x^2+y^2-1, (x+y)^2-1\rangle$

Abb. 9.6 - Treppen der in diesem Kapitel angeführten Ideale von $\mathbb{Q}[x,y]$. In diesen drei Fällen sind die Treppen und die Positionen der Generatoren für die Ordnungen `lex`, `deglex` und `degrevlex` dieselben.



$\texttt{lex}(x,y)$ $\texttt{invlex}(x,y) = \texttt{lex}(y,x)$ $\texttt{degrevlex}(x,y)$

Abb. 9.7 - Treppe des Ideals $\langle xy+x+y^2+1, x^2y+xy^2+1\rangle \subset \mathrm{QQ}[x,y]$ bezogen auf verschiedene monomiale Ordnungen.

In jedem Schema entspricht das grau unterlegte Gebiet den Leittermen der *Elemente* des Ideals. Die schwarzen Quadrate geben die Positionen der für seine Beschreibung verwendeten *Generatoren* an.

Gemäß der Gestalt der Treppenlinie (Abb. 9.6) fehlt dafür ein Polynom von J_1 des Leitmonoms y^4.

Das oben angedeutete und auf dem Dickson-Lemma gründende Argument zeigt, dass jedes Ideal Gröbnerbasen zulässt[9]. Berechnen wir Gröbnerbasen von J_1 und weiteren Idealen, deren

[9] Dieses Ergebnis kann man auch als eine effektive Version des Hilbertschen Basissatzes sehen, der besagt, dass die Ideale von $\mathbb{K}[\boldsymbol{x}]$ durch eine endliche Anzahl von Elementen erzeugt werden. Ein klassischer Beweis dieses Satzes erinnert stark an die Bildung einer Gröbnerbasis für die lexikographische Ordnung.

Treppenlinien in Abb, 9.6 dargestellt sind. Für J_1 kommt

```
sage: R.ideal(x^2+y^2-1, 16*x^2*y^2-1).groebner_basis()
[x^2 + y^2 - 1, y^4 - y^2 + 1/16]
```

Wie erwartet erscheinen hier die Leitmonome x^2 und y^4. Ihr Auftreten erklärt, wie sich die Treppenlinie auf die Achsen zurückzieht, woraus wir erkennen, dass das für Systeme der Dimension null charakteristisch ist. Bezogen allein auf die Doppelhyperbel finden wir

```
sage: R.ideal(16*x^2*y^2-1).groebner_basis()
[x^2*y^2 - 1/16]
```

d. h. ein Vielfaches des erzeugenden Polynoms. Ganz generell ist jedes einzelne Polynom eine Gröbnerbasis. Das dritte Beispiel zeigt, dass eine Gröbnerbasis mehrere Polynome benötigen kann als ein beliebiges System von Erzeugenden:

```
sage: R.ideal(x^2+y^2-1, (x+y)^2-1).groebner_basis()
[x^2 + y^2 - 1, x*y, y^3 - y]
```

Aufgrund der Einfachheit der vorstehenden Beispiele hängen diese drei Gröbnerbasen nicht oder nur wenig von der monomialen Ordnung ab. Das ist im allgeneinen nicht so. Die Abb. 9.7 stellt Treppenlinien für drei klassische monomiale Ordnungen dar, die demselben Ideal von $\mathbb{Q}[vx]$ zugeordnet sind. Die entsprechenden Gröbnerbasen sind:

```
sage: R_lex.<x,y> = PolynomialRing(QQ, order='lex')
sage: J_lex = (x*y+x+y^2+1, x^2*y+x*y^2+1)*R_lex; J_lex.gens()
[x*y + x + y^2 + 1, x^2*y + x*y^2 + 1]
sage: J_lex.groebner_basis()
[x - 1/2*y^3 + y^2 + 3/2, y^4 - y^3 - 3*y - 1]

sage: R_invlex = PolynomialRing(QQ, 'x,y', order='invlex')
sage: J_invlex = J_lex.change_ring(R_invlex); J_invlex.gens()
[y^2 + x*y + x + 1, x*y^2 + x^2*y + 1]
sage: J_invlex.groebner_basis()
[y^2 + x*y + x + 1, x^2 + x - 1]

sage: R_drl = PolynomialRing(QQ, 'x,y', order='degrevlex')
sage: J_drl = J_lex.change_ring(R_drl); J_drl.gens()
 [x*y + y^2 + x + 1, x^2*y + x*y^2 + 1]
sage: J_drl.groebner_basis()
[y^3 - 2*y^2 - 2*x - 3, x^2 + x - 1, x*y + y^2 + x + 1]
```

Reduktion	
multivariate Divison von p durch G	`p.reduce(G)`
zwischenreduzierte Erzeugende	`J.interreduced_basis()`

Gröbnerbasen	
Test, ob Gröbnerbasis	`J.basis_is_groebner()`
Gröbnerbasis (reduziert)	`J.groebner_basis()`
Wechsel der Ordnung für `lex`	`J.transformed_basis()`
Wechsel der Ordnung $R_1 \mapsto R_2$	`J.transformed_basis('fglm', other_ring=R2)`

Tab. 9.6 - Gröbnerbasen

Die Gröbnerbasis zur Ordnung `lex` hebt die Regel für die neue Schreibweise $x = \frac{1}{2}y^3 - y^2 - \frac{3}{2}$ deutlich hervor, dank der wir die Elemente des Quotientenrings als Funktion nur einer Variablen y ausdrücken können. Die verlängerte Form der entsprechenden Treppenlinie spiegelt diese Möglichkeit wieder. Ebenso zeigt die Gröbnerbasis zur Ordnung `invlex` an, dass wir die Potenzen von y durch die Gleichung $y^2 = -xy - x - 1$ eliminieren können. Darauf kommen wir am Ende des Kapitels noch einmal zurück.

9.3.4. Eigenschaften der Gröbnerbasen

Die Gröbnerbasen dienen zur Implementierung der in Abschnitt 9.2 untersuchten Operationen. Wir benutzen sie nämlich zu Berechnung der Normalformen von Idealen von Polynomringen, und der Elemente der Quotienten durch diese Ideale, zur Eliminierung von Variablen in polynomialen Systemen oder auch zur Bestimmung der Charakteristik der Lösungen wie ihrer Dimension.

Division durch eine Gröbnerbasis. Die Division durch eine Gröbnerbasis G eines Polynoms aus $\langle G\rangle$ kann bei einem von null verschiedenen Element aus $\langle G\rangle$ nicht terminieren. Das ist eine unmittelbare Konsequenz der Definition: ein solches Element befände sich oberhalb der zu $\langle G\rangle$ gehörenden Treppenlinie und wäre daher auch durch G teilbar. Jedes Element in $\langle G\rangle$ reduziert sich somit bei der Division durch G auf null. Insbesondere erzeugt eine Gröbnerbasis eines Ideals J wiederum J.

Ebenso kann die Division eines Polynoms $p \notin J$ durch eine Gröbnerbasis von J nur bei einem Polynom „unter der Treppe“ terminieren, nun gehören aber zwei verschiedene Polynome „unter der Treppe“ zu verschiedenen Äquivalenzklassen modulo J (weil ihre Differenz auch „unter der Treppe“ ist). Die Division durch eine Gröbnerbasis liefert daher eine Normalform für die Elemente des Quotientenrings $\mathbb{K}[\boldsymbol{x}]/J$, und das unabhängig von der Reihenfolge, in der die elementaren Reduktionen vorgenommen werden. Die Normalform einer Äquivalenzklasse $p+J$ ist ihr einziger Repräsentant, der unter der Treppenlinie liegt oder null ist. Es ist diese Normalform, welche die in Unterabschnitt 9.2.3 vorgestellten Operationen der Quotientenalgebra berechnen. Um das Beispiel der Abb. 9.7 fortzusetzen, wird die Reduktion

```
sage: p = (x + y)^5
sage: J_lex.reduce(p)
17/2*y^3 - 12*y^2 + 4*y - 49/2
```

zerlegt in eine Berechnung der Gröbnerbasis und eine anschließende Division

```
sage: p.reduce(J_lex.groebner_basis())
17/2*y^3 - 12*y^2 + 4*y - 49/2
```

Das Ergebnis einer Projektion in den Quotienten ist im Wesentlichen dasselbe:

```
sage: R_lex.quo(J_lex)(p)
17/2*ybar^3 - 12*ybar^2 + 4*ybar - 49/2
```

Natürlich verursacht ein Wechsel der monomialen Ordnung eine andere Bestimmung der Normalform:

```
sage: R_drl.quo(J_drl)(p)
5*ybar^2 + 17*xbar + 4*ybar + 1
```

Die Monome, die in der Normalform auftreten, entsprechen Punkten unter der Treppenlinie.

Somit hat das Ideal J die Dimension null genau dann, wenn die Anzahl der Punkte unter seiner Treppe endlich ist, und diese Anzahl der Punkte ist die Dimension des Quotientenrings $\mathbb{K}[\boldsymbol{x}]/J$. In diesem Fall ist die in Unterabschnitt 9.2.5 beschriebene Basis, die von der Methode `normal_basis` zurückgegeben wird, einfach die Menge der Monome unter der Treppenlinie für die vorliegende monomiale Ordnung.:

```
sage: J_lex.normal_basis()
[y^3, y^2, y, 1]
sage: J_invlex.normal_basis()
[x*y, y, x, 1]
sage: J_drl.normal_basis()
[y^2, y, x, 1]
```

Nebenbei bemerken wir, dass die Anzahl der Monome unter der Treppenlinie unabhängig von der monomialen Ordnung ist.

Dimension. Wir sind nun gerüstet, eine Definition der Dimemsion eines Ideals für den allgemeinen Fall zu geben: J hat die Dimemsion d, wenn die Anzahl der Punkte unter der Treppenlinie, die den Monomen des Totalgrades höchstens t entspricht, von der Ordnung t^d ist, wenn $t \to \infty$ geht. Zum Beispiel hat das Ideal $\langle 16x^2x^2 - 1\rangle$ (Abb. 9.6) die Dimension 1:

```
sage: ideal(16*x^2*y^2-1).dimension()
1
```

In der Tat ist die Anzahl der Monome m unter der Treppenlinie so, dass $\deg_x m + \deg_y m \leq t$ für $t \geq 3$ gleich $4t - 2$ ist. Genauso haben die beiden Ideale in Abb. 9.5 die Dimensionen 1 bzw. 2. Man kann zeigen, dass der Wert der Dimension von der monomialen Ordnung nicht abhängt, sondern bis auf entartete Fälle der „geometrischen“ Dimension der dem Ideal zugehörigen Varietät entspricht.

Reduzierte Basen. Eine endliche Menge von Polynomen, die eine Gröbnerbasis enthält, ist selber eine Gröbnerbasis, weshalb ein von null verschiedenes Ideal unendlich viele Gröbnerbasen hat. Eine Gröbnerbasis $G = \{g_1, \ldots, g_s\}$ heißt reduziert, wenn

- die Leitkoeffizienten der g_i sind sämtlich 1 (und $0 \notin G$);
- kein g_i ist im Sinne der Regeln (9.10) durch den Rest der Basis $G \setminus \{g_i\}$ reduzierbar.

Bei festgelegter monomialer Ordnung lässt jedes Ideal eine eindeutige reduzierte Gröbnerbasis zu. Beispielsweise ist die reduzierte Gröbnerbasis des Ideals $\langle 1\rangle$ die einelementige Menge $\{1\}$ für welche Ordnung auch immer. Die reduzierten Gröbnerbasen liefern daher für die Ideale von $\mathbb{K}[\boldsymbol{x}]$ eine Normalform.

Eine reduzierte Gröbnerbasis ist in dem Sinne minimal, dass das, was nach Wegnahme eines beliebigen Elements übrig bleibt, kein Erzeugendensystem des Ideals mehr ist. Konkret umfasst sie genau ein Polynom pro „Ecke“ der Treppenlinie. Sie kann aus einer beliebigen Gröbnerbasis G berechnet werden, indem jedes Element $g \in G$ durch seinen Rest der Division durch $G \setminus \{g\}$, soweit das möglich ist, ersetzt wird. Das ist es, was die Methode `interreduced_basis` macht. Die zu null reduzierten Polynome werden gelöscht.

Elimination. Die lexikographischen Ordnungen erfreuen sich folgender grundlegender Eigenschaft: *wenn G eine Gröbnerbasis für die lexikographische Ordnung von $J \subset \mathbb{K}[x_1, \ldots, x_n]$ ist, dann sind die $G \cap \mathbb{K}[x_{k+1,\ldots,x_n}]$ Gröbnerbasen der Eliminationsideale*[10] $J \cap \mathbb{K}[x_{k+1,\ldots,x_n}]$. Eine lexikographische Gröbnerbasis wird in Blöcke zerlegt, deren letzter nur von x_n abhängt, der vorletzte von x_n und x_{n-1} usw.[11]:

```
sage: R.<t,x,y,z> = PolynomialRing(QQ, order='lex')
sage: J = R.ideal(t+x+y+z-1, t^2-x^2-y^2-z^2-1, t-x*y)
sage: [u.polynomial(u.variable(0)) for u in J.groebner_basis()]
[t + x + y + z - 1,
(y + 1)*x + y + z - 1,
(z - 2)*x + y*z - 2*y - 2*z + 1,
(z - 2)*y^2 + (-2*z + 1)*y - z^2 + z - 1]
```

In diesem Beispiel hängt das letzte Basispolynom nur von y und z ab. Zunächst kommt ein Block von zwei Polynomen in x, y und z und als erstes ein Polynom, in dem sämtliche Variablen vorkommen. Die aufeinander folgenden Eliminationsideale werden unmittelbar abgelesen.

Wir haben indessen gesehen (Unterabschnitt 9.2.5), dass die letzteren keine vollständige Beschreibung des Ideals liefern. Hier ist der Block mit z allein leer, daher erscheint jeder Wert von z, vielleicht bis auf endlich viele Ausnahmen, als letzte Koordinate einer Lösung. Wir sind versucht, die möglichen y-Werte für jedes z mittels der letzten Gleichung auszudrücken. Es sind zwei an der Zahl, außer für $z = 2$, wozu nur $y = -1$ passt. Nur beim Übergang zur vorstehenden Gleichung sehen wir, dass die Festlegung $z = 2$ widersprüchlich ist. Umgekehrt folgt gemäß der letzten Gleichung wiederum $z = 2$ aus $y = -1$ und ist daher ausgeschlossen. Es erweist sich daher, dass jeder Leitkoeffizient der Polynome (geschrieben in ihrer jeweiligen Haupvariablen, wie im Auszug weiter oben) für $z \neq 2$ zu null wird.

Übung 37 *(Trigonometrische Beziehungen).* Es soll $(\sin\theta)^6$ als Polynom in $u(\theta) = \sin\theta + \cos\theta$ und $v(\theta) = \sin(2\theta) + \cos(2\theta)$ geschrieben werden.

9.3.5. Berechnung

Wir verweisen den an Algorithmen zur Berechnung von Gröbnerbasen interessierten Leser nochmals auf die in der Einleitung erwähnte Literatur [CLO07, FSED09, EM07]. Als Ergänzung bieter der Modul `sage.rings.polynomial.toy_buchberger` eine „pädagogische“ Implementierung des Buchberger-Algorithmus und weiterer damit verbundener Algorithmen, die seiner theoretischen Beschreibung eng folgt.

[10] Für gegebenes k stimmt dies noch allgemeiner bei jeder Ordnung mit $i \leq k < j \Rightarrow x_i > x_j$. Eine Ordnung mit dieser Eigenschaft heißt blockweise (siehe auch Tab. 9.5).

[11] Das ist deshalb eine „Dreiecksform“ des Systems, die von den Erzeugenden des Ideals gebildet wird, wenn auch in schwächerem Sinne als die in Unterabschnitt 9.2.5 definierte: *a priori* können wir über die Anzahl der Polynome jedes Blocks oder ihre Leitterme nichts aussagen.

Wechsel der Ordnung

Die interessantesten Gröbnerbasen sind nicht die, die am einfachsten zu berechnen sind: oft ist die Ordnung `degrevlex` am wenigsten aufwendig, doch mehr Informationen entnimmt man einer lexikographischen Gröbnerbasis. Außerdem braucht man manchmal Gröbnerbasen eines Ideals zu verschiedenen Ordnungen.
Das motiviert zur Einführung weiterer allgemeiner Algorithmen zur Berechnung von Gröbnerbasen, sogenannten Algorithmen für den Ordnungswechsel. Sie berechnen aus einer Gröbnerbasis zu einer monomialen Ordnung zum selben Ideal eine Gröbnerbasis zu einer anderen monomialen Ordnung. Oft sind sie effizienter als diejenigen, die von einem beliebigen Erzeugendensystem ausgehen. So besteht eine oftmals fruchtbare Strategie, eine lexikographische Gröbnerbasis zu erhalten, darin, eine Gröbnerbasis zur Ordnung `degrevlex` zu berechnen und dann ein Verfahren für den Ordnungswechsel anzuwenden.
Die Methode `transformed_basis` ermöglicht die Berechnung von Gröbnerbasen durch Ordnungswechsel „von Hand“, wenn das Ideal die Ordnung null hat oder die Zielordnung `lex` ist. Notfalls beginnt die Methode mit der Berechnung einer Gröbnerbasis zu der Ordnung, die zum Polynomring gehört.

Merken wir uns jedenfalls, dass die Berechnung einer Gröbnerbasis eine aufwendige Operation ist, aufwendig hinsichtlich Rechenzeit und Speicherbedarf, in ungünstigen Fällen sogar sehr aufwendig. Die Methode `groebner_basis` lässt außerdem zahlreiche Optionen[12] zu, die dem erfahrenen Anwender erlauben, einen Algorithmus zur Berechnung einer Gröbnerbasis je nach Eigenart des Problems von Hand festzulegen.

Betrachten wir die Ideale $C_n(K) \subset \mathbb{K}[x_1, \ldots, x_{n-1}]$, die definiert sind durch

$$\begin{aligned} C_2(K) &= \langle x_0 + x_1, x_0x_1 - 1\rangle \\ C_3(K) &= \langle x_0 + x_1 + x_2, x_0x_1 + x_0x_2 + x_1x_1, x_0x_1x_2 - 1\rangle \\ &\vdots \\ C_n(K) &= \Big\langle \sum_{i\in\mathbb{Z}/n\mathbb{Z}} \prod_{j=0}^{k} x_{i+j} \Big\rangle_{k=0}^{n-2} + \langle x_0 \ldots x_{n-1} - 1\rangle, \end{aligned} \tag{9.11}$$

und in Sage zu bilden durch Befehle wie

```
sage: from sage.rings.ideal import Cyclic
sage: Cyclic(QQ['x,y,z'])
Ideal (x + y + z, x*y + x*z + y*z, x*y*z - 1) of
Multivariate Polynomial Ring in x, y, z over Rational Field
```

Dies sind klassische Testaufgaben zur Leistungsbewertung von Werkzeugen zur Lösung polynomialer Systeme. Auf einer Maschine, wo Sage für die Berechnung der reduzierten Gröbnerbasis von $C_6(\mathbb{Q})$ weniger als eine Sekunde benötigt:

```
sage: def C(R, n): return Cyclic(PolynomialRing(R, 'x', n))

sage: %time len(C(QQ, 6).groebner_basis())
CPU times: user 0.25 s, sys: 0.01 s, total: 0.26 s
```

[12] Wegen Details siehe ihre Hilfeseite sowie diejenigen der internen Methoden des betreffenden Ideals, deren Namen mit `_groebner_basis` beginnt.

```
Wall time: 0.97 s
45
```

endet die Berechnung der Gröbnerbasis von $C_7(\mathbb{Q})$ nicht in einem Dutzend Stunden und benötigt 3 Go Speicher.

Statt die Gröbnerbasis auf den rationalen Zahlen berechnen zu lassen, versuchen wir, $\mathbb{Q}$ durch einen endlichen Körper $\mathbb{F}_p$ zu ersetzen. Die Idee, normal bei symbolischer Rechnung, ist die Eingrenzung des Aufwands der Operationen mit den Koeffizienten: Operationen mit den Elementen von $\mathbb{F}_p$ benötigen eine konstante Zeit, während die Anzahl der Ziffern der rationalen Zahlen die Tendenz hat, im Verlauf der Rechnung gewaltig anzuwachsen. Wir wählen p hinreichend klein, damit die Rechnungen auf $\mathbb{F}_p$ direkt auf den ganzen Zahlen der Maschine ausgeführt werden können. Doch p darf auch nicht zu klein gewählt werden, damit die Gröbnerbasis auf $\mathbb{F}_p$ die Chance hat, an einem guten Teil der Struktur von $\mathbb{Q}$ teilzuhaben.

Mit einem vertretbaren p hat beispielsweise die Gröbnerbasis von $C_6(\mathbb{F}_p)$ die gleiche Anzahl von Elementen wie diejenige von $C_6(\mathbb{Q})$:

```
sage: p = previous_prime(2^30)
sage: len(C(GF(p), 6).groebner_basis())
45
```

Bei Vergrößerung des zu behandelnden Systems sehen wir den Einfluss des Koeffizientenkörpers auf die Rechenzeit und der ist weit davon entfernt, vernachlässigbar zu sein: die Fälle $n = 7$ und $n = 8$ werden problemlos machbar.

```
sage: %time len(C(GF(p), 7).groebner_basis())
CPU times: user 3.71 s, sys: 0.00 s, total: 3.71 s
Wall time: 3.74 s
209
sage: %time len(C(GF(p), 8).groebner_basis())
CPU times: user 104.30 s, sys: 0.07 s, total: 104.37 s
Wall time: 104.49 s
372
```

Diese Beispiele illustrieren noch ein weiteres wichtiges Phänomen: die Ausgabe einer Berechnung der Gröbnerbasis kann viel größer werden als die Eingabe. Somit zeigt die letze Rechnung oben, dass jede Gröbnerbasis, reduziert oder nicht, von $C_8(\mathbb{F}_p)$ (mit diesem Wert für p) zur Ordnung `degrevlex` mindestens 372 Elemente zählt, wenn C_8 von nur 8 Polynomen erzeugt wird.

10. Differentialgleichungen und rekursiv definierte Folgen

> Betrachten Sie diese Differentialgleichung solange,
> bis sich die Lösung von selbst einstellt.
>
> George Pólya

10.1. Differentialgleichungen

10.1.1. Einführung

Wenn die Methode von George Pólya nicht verfangen will, ziehen Sie Sage hinzu, auch wenn das Gebiet der symbolischen Lösung von Differentialgleichungen eine Schwäche vieler Rechenprogramme bleibt. Sage ist jedoch in voller Entwicklung und vergrößert sein Leistungsspektrum mit jeder neuen Version.

Wenn man möchte, kann man Sage aufrufen, um eine qualitative Untersuchung zu erhalten, und in der Tat leiten dann Sages numerische und grafische Werkzeuge die Intuition. Das ist Gegenstand von Abschnitt 14.2 des Kapitels, das der numerischen Integration gewidmet ist. Werkzeuge zur grafischen Untersuchung werden in Unterabschnitt 4.1.6 angegeben. Lösungsverfahren mit Hilfe von Reihen finden sich in Unterabschnitt 7.5.2.

Man kann die exakte Lösung von Differentialgleichungen vorziehen. Sage kann hier zuweilen helfen, indem es direkt eine symbolische Antwort gibt, wie wir in diesem Kapitel sehen werden.

In den meisten Fällen muss Sage mit einer geschickten Umstellung dieser Gleichungen geholfen werden. Vor allen Dingen muss man beachten, dass die erwartete Lösung einer Differentialgleichung eine in einem bestimmten Intervall differenzierbare *Funktion* ist, dass Sage aber *Ausdrücke* ohne Definitionsbereich manipuliert. Die Maschine benötigt daher einen menschliche Eingriff, um zu einer hinreichend genauen Lösung zu gelangen.

Wir werden zunächst gewöhnliche Differentialgleichungen 1. Ordnung allgemein studieren und einige besondere Fälle wie die linearen Gleichungen, die Gleichungen mit trennbaren Variablen, die homogenen Gleichungen, eine Gleichung, die von einem Parameter abhängt Unterabschnitt 10.1.2), dann eher summarisch die Gleichungen 2. Ordnung, sowie ein Beispiel einer partiellen Differentialgleichung (Unterabschnitt 10.1.3). Wir werden mit dem Gebrauch der Laplace-Transformation (Unterabschnitt 10.1.4) und der Lösung bestimmter Systeme von Differentialgleichungen schließen.

Wir erinnern uns, dass eine *gewöhnliche Differentialgleichung* (mannchmal mit GDG abgekürzt oder ODE auf englisch) eine Gleichung ist, in der eine (unbekannte) Funktion einer Veränderlichen auftritt sowie eine oder mehrere Ableitungen dieser Funktion.

In der Gleichung $y'(x) + x \cdot y(x) = \sin(x)$ wird die unbekannte Funktion y die *abhängige Variable* genannt und x die *unabhängige Variable*, mit deren Größe sich y ändert.

Eine *partiellen Differentialgleichung* (manchmal mit PDG abgekürzt oder PDE auf englisch) ist eine Funktion, in der mehrere unabhängige Variablen auftreten und somit auch partielle Ableitungen der abhängigen Variablen nach diesen unabhängigen Variablen.

Wenn nicht anders erwähnt, *betrachten wir in diesem Kapitel Funktionen einer reellen Variablen.*

10.1.2. Gewöhnliche Differentialgleichungen 1. Ordnung

Grundlegende Befehle. Wir wollen eine GDG 1. Ordnung lösen

$$F(x, y(x), y'(x)) = 0$$

Wir beginnen mit der Definition einer Variablen x und einer von ihr abhängigen Variablen y:

```
sage: x = var('x')
sage: y = function('y')(x)
```

Dann schreibt man

```
sage: desolve(gleichung, variable, ics = ..., ivar = ...,
 ....:         show_method = ..., contrib_ode = ...)
```

worin

- `gleichung` die Differentialgleichung ist. Gleichheit wird durch == symbolisiert. Beispielsweise wird die Gleichung $y' = 2y + x$ so geschrieben: `diff(y,x) == 2*y + x`;
- `variable` ist der Name der abhängigen Variablen, d.h. in $y' = 2y + x$ ist das die Funktion y;
- `ics` ist ein optionales Argument, mit dem die Anfangsbedingungen angegeben werden können. Für eine Gleichung 1. Ordnung ist das eine Liste $[x_0, y_0]$, für eine Gleichung 2. Ordnung ist das $[x_0, y_0, x_1, y_1]$ oder $[x_0, y_0, y_0']$;
- `ivar` ist ein optionales Argument, mit dem die unabhängige Variable präzisiert werden kann, d.h. in $y' = 2y + x$ ist das x. Dieses Argument muss unbedingt dann angegeben werden, falls Gleichungen von Parametern abhängen wie zum Beispiel $y' = ay + bx + c$;
- `show_method` ist ein optionales Argument, das mit `False` voreingestellt ist. Andernfalls verlangt es von Sage, das Lösungsverfahren anzugeben. Das können sein `linear`, `separabel`, `exact`, `homogeneous`, `bernoulli`, `generalzed homogeneous`. Sage gibt dann eine Liste zurück,deren erstes Argument die Lösung ist und deren zweites das verwendete Lösungsverfahren;
- `contrib_ode` ist ein optionales Argument, das mit `False` voreingestellt ist. Bei `True` kann Sage mit Gleichungen von Riccati, Lagrange, Clairaut und mit anderen pathologischen Fällen umgehen.

Gleichungen 1. Ordnung können mit Sage direkt gelöst werden. Wir werden in diesem Abschnitt studieren, wie mit Sage die linearen Gleichungen mit trennbaren Variablen, die Gleichungen von Bernoulli, die homogenen Gleichungen, die exakten Gleichungen, sowie die Gleichungen von Riccati, Lagrange und Clairaut zu lösen sind.

Lineare Gleichungen. Dabei handelt es sich um Gleichungen des Typs

$$y' = P(x)y = Q(x),$$

wobei es sich bei P und Q um Funktionen handelt, die auf den gegebenen Intervallen stetig sind.
Beispiel: $y' + 3y = e^x$.

```
sage: x=var('x'); y = function('y')(x)

sage: desolve(diff(y,x)+3*y==exp(x), y, show_method=True)
[1/4*(4*_C + e^(4*x))*e^(-3*x), 'linear']
```

Gleichungen mit trennbaren Variablen. Hierbei handelt es sich um Gleichungen des Typs

$$P(x) = y'Q(y),$$

wobei es sich bei P und Q um Funktionen handelt, die auf den gegebenen Intervallen stetig sind.
Beispiel: $yy' = x$.

```
sage: desolve(y*diff(y,x) == x, y, show_method=True)
[1/2*y(x)^2 == 1/2*x^2 + _C, 'separable']
```

Achtung! Manchmal erkennt Sage nicht, dass es sich um eine Gleichung mit trennbaren Variablen handelt und behandlet sie als exakte Gleichungen. Beispiel: $y' = e^{x+y}$.

```
sage: desolve(diff(y,x) == exp(x+y), y, show_method=True)
[-(e^(x + y(x)) + 1)*e^(-y(x)) == _C, 'exact']
```

Gleichungen von Bernoulli. Das sind Gleichungen des Typs

$$y' = P(x)y = Q(x)y^\alpha,$$

wobei es sich bei P und Q um Funktionen handelt, die auf den gegebenen Intervallen stetig sind.
Beispiel: $y' - y = xy^4$.

```
sage: desolve(diff(y,x)-y == x*y^4, y, show_method=True)
[e^x/(-1/3*(3*x - 1)*e^(3*x) + _C)^(1/3), 'bernoulli']
```

Homogene Gleichungen. Es handelt sich um Gleichungen des Typs

$$y' = \frac{P(x,y)}{Q(x,y)},$$

wobei es sich bei P und Q um Funktionen handelt, die auf den gegebenen Intervallen stetig sind.
Beispiel: $x^2y' = y^2 + xy + x^2$.

```
sage: desolve(x^2*diff(y,x) == y^2 + x*y + x^2, y, show_method=True)
[_C*x == e^(arctan(y(x)/x)), 'homogeneous']
```

Die Lösungen werden nicht in expliziter Form ausgegeben. Später werden wir sehen, wie wir uns in bestimmten Fällen helfen können.

Exakte Gleichungen. Es handelt sich um Gleichungen des Typs

$$f' = \frac{\partial f}{\partial x}dx + \frac{\partial f}{\partial y}dy$$

mit einer differenzierbaren Funktion f zweier Variablen.
Beispiel: $y' = \dfrac{\cos(y) - 2x}{y + x\sin(y)}$ mit $f = x^2 - x\cos y + y^2/2$.

```
sage: desolve(diff(y,x) == (cos(y)-2*x)/(y+x*sin(y)), y,
....:         show_method=True)
[x^2 - x*cos(y(x)) + 1/2*y(x)^2 == _C, 'exact']
```

Auch hier werden die Lösungen nicht explizit ausgegeben.

Gleichungen von Riccati. Es handelt sich um Gleichungen des Typs

$$y' = P(x)y^2 + Q(x)y + R(x),$$

wobei es sich bei P, Q und R um Funktionen handelt, die auf den gegebenen Intervallen stetig sind.
Beispiel: $y' = xy^2 + \dfrac{1}{x}y - \dfrac{1}{x^2}$.

In diesem Fall muss die Option `contrib_ode` auf `True` gesetzt werden, damit Sage mit komplexeren Verfahren nach einer Lösung sucht.

```
sage: desolve(diff(y,x) == x*y^2 + y/x - 1/x^2, y,
....:         contrib_ode=True, show_method=True)[1]
'riccati'
```

Gleichungen von Lagrange und von Clairaut. Wenn die Gleichung die Form $y = xP(y') + Q(y')$ hat mit auf einem bestimmten Intervall stetigen Funktionen P und Q, spricht man von einer Lagrange-Gleichung. Ist $P = 1$, spricht man von einer Clairaut-Gleichung.
Beispiel: $y = xy' - y'^2$.

```
sage: desolve(y == x*diff(y,x) - diff(y,x)^2, y,
....:         contrib_ode=True, show_method=True)
[[y(x) == -_C^2 + _C*x, y(x) == 1/4*x^2], 'clairault']
```

Lineare Gleichung. Lösen wir beispielsweise $y' + 2y = x^2 - 2x + 3$;

```
sage: x =  var('x'); y = function('y')(x)

sage: DE = diff(y,x) + 2*y == x**2 - 2*x + 3
sage: desolve(DE, y)
1/4*((2*x^2 - 2*x + 1)*e^(2*x) - 2*(2*x - 1)*e^(2*x) + 4*_C
+ 6*e^(2*x))*e^(-2*x)
```

Wir ordnen das ein wenig mit der Methode `expand`:

```
sage: desolve(DE, y).expand()
1/2*x^2 + _C*e^(-2*x) - 3/2*x + 9/4
```

Wir gewöhnen uns daher an, `desolve(...).expand()` zu schreiben. Welches Verfahren ist zur Anwendung gekommen?

```
sage: desolve(DE, y, show_method=True)[1]
'linear'
```

Nun wollen wir Anfangsbedingungen hinzufügen, zum Beispiel $x(0) = 1$:

```
sage: desolve(DE, y, ics=[0,1]).expand()
1/2*x^2 - 3/2*x - 5/4*e^(-2*x) + 9/4
```

Gleichungen mit separierbaren Variablen. Untersuchen wir die Gleichung $y' \ln(y) = y \sin(x)$:

```
sage: x = var('x'); y = function('y')(x)
sage: desolve(diff(y,x)*log(y) == y*sin(x), y, show_method=True)
[1/2*log(y(x))^2 == _C - cos(x), 'separable']
```

Sage ist mit uns einer Meinung: das ist tatsächlich eine Gleichung mit trennbaren Variablen.

Gewöhnen wir uns an, unsere Lösungen zu benennen, damit wir sie später wiederverwenden können:

```
sage: ed = desolve(diff(y,x)*log(y) == y*sin(x), y); ed
1/2*log(y(x))^2 == _C - cos(x)
```

Hier ist $y(x)$ nicht in expliziter Form gegeben: $\frac{1}{2} * log(y(x))^2 == \_C - cos(x)$.

Man kann mit `solve` einen Ausdruck für $y(x)$ verlangen. Wir schauen uns nun `ed` und `y` genauer an:

```
sage: solve(ed, y)
[y(x) == e^(-sqrt(2*_C - 2*cos(x))), y(x) == e^(sqrt(2*_C - 2*cos(x)))]
```

Wir müssen dabei besonders auf den Term `sqrt(2*_C - 2*cos(x))` achten, auch wenn Sage uns nicht gleich warnt, müssen wir deshalb dafür sorgen, dass _C ≥ 1 bleibt.

Um das Verhalten der Kurven der Lösungen zu erhalten, müssen wir die rechte Seite der Gleichung mit der Methode `rhs()` weiterverwenden. Statt beispielsweise die rechte Seite der ersten Lösung nach Einsetzen von 5 für `_C` zu bekommen, sehen wir zunächst eine Fehlermeldung:

```
sage: solve(ed, y)[0].subs_expr(_C==5).rhs()
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-5-77ebca8faf1a> in <module>()
----> 1 solve(ed, y)[Integer(0)].subs_expr(_C==Integer(5)).rhs()

NameError: name '_C' is not defined
```

In der Tat haben wir `_C` nicht definiert. Sage hat es gemerkt. Um auf `_C` zugreifen zu können, müssen wir die Methode `variables()` verwenden, welche die Liste der in einem Ausdruck vorkommenden Variablen ausgibt.

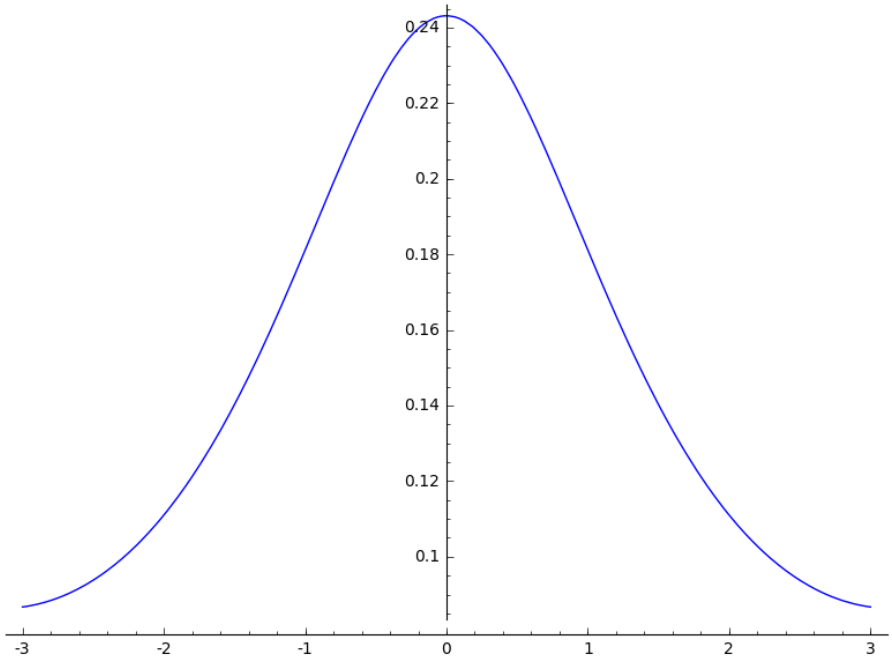
```
sage: ed.variables()
(_C, x)
```

Nur $\_C$ und x werden also als Variablen angesehen, denn y ist als Funktion der Variablen `x` definiert worden. Wir bekommen $\_C$ daher mit `ed.variables()[0]`:

```
sage: _C = ed.variables()[0]
sage: solve(ed, y)[0].subs(_C == 5).rhs()
e^(-sqrt(-2*cos(x) + 10))
```

Ein anderes Beispiel: Um den Graphen der ersten Lösung mit `_C=2` zu erhalten, schreiben wir
`sage: plot(solve(ed, y)[0].subs(_C == 2).rhs(), x, -3. 3)`

und wir erhalten



Um mehrere Kurven zu bekommen (siehe Abb. 10.1) arbeiten wir mit einer Schleife:

```
sage: P = Graphics()
sage: for k in range(1,20,2):
....:     P += plot(solve(ed, y)[0].subs(_C == 1 + k/4).rhs(), x, -3, 3)
sage: P
```

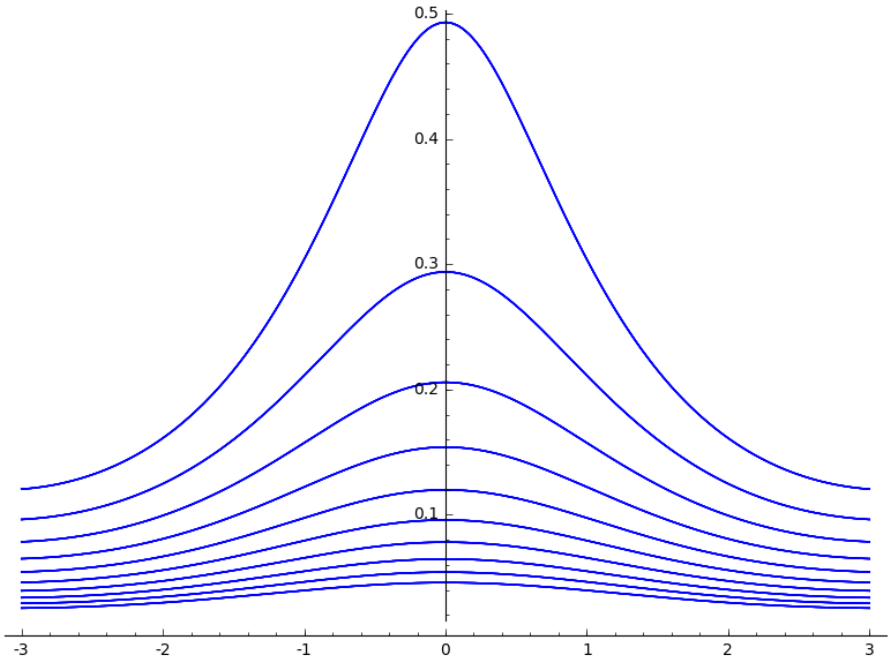
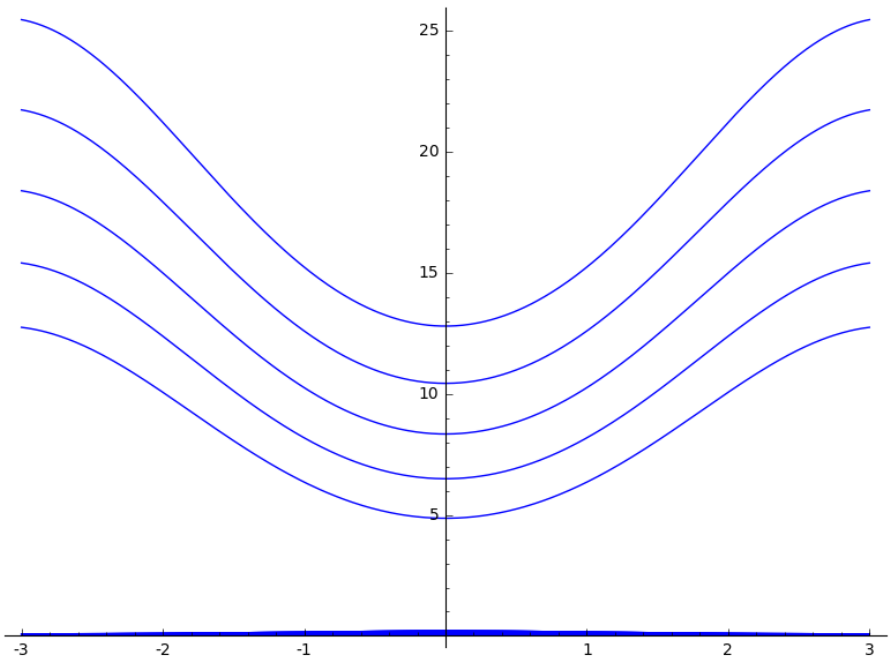


Abb. 10.1 - Mehrere Lösungen von $y' \ln y = y \sin(x)$.

Mit zwei Schleifen könnten wir versuchen, die Bilder zweier Lösungen einzufangen,

```
sage: P = Graphics()
sage: for j in [0, 1]:
....:     for k in range(1,10,2):
....:         f = solve(ed, y)[0].subs(_C == 2 + k/4).rhs()
....:         P += plot(f, x, -3, 3) sage: P
```

doch die Differenz der Maßstäbe beider Lösungen verhindert, dass man die Kurven der ersten Lösung unterscheiden kann:



Übung 38. *(Differentialgleichungen mit trennbaren Variablen).* Lösen Sie folgende Differentialgleichungen mit trennbaren Variablen auf $\mathbb{R}$:

$$1.\ (E_1):\ \frac{yy'}{\sqrt{1+y^2}} = \sin(x); \qquad 2.\ (E_2):\ y' = \frac{\sin(x)}{\cos(x)}.$$

Homogene Gleichungen. Wir wollen die homogene Gleichung $xy' = y + \sqrt{y^2+x^2}$ lösen. In der Tat ist dies eine homogene Gleichung, denn man kann sie auch so schreiben:

$$\frac{dy}{dx} = \frac{y+\sqrt{y^2+x^2}}{x} = \frac{N(y,x)}{M(y,x)}$$

oder $N(ky,kx) = kN(y,x)$ und $M(ky,kx) = kM(y,x)$.

Daher genügt es, diese Gleichung durch die Substitution $y(x) = x \cdot u(x)$ in eine Gleichung mit trennbaren Variablen zu überführen:

```
sage: u = function('u')(x)
sage: y = x*u
sage: DE = x*diff(y,x) == y + sqrt(x**2 + y**2)
```

Wir wenden die Substitution auf die Ausgangsgleichung an.

Da die Gleichung für $x = 0$ nicht definiert ist, werden wir sie auf den Intervallen $]-\infty, 0[$ und $]0, \infty[$ lösen. Arbeiten wir zunächst auf $]0, \infty[$.

```
sage: assume x > 0
sage: desolve(DE, u)
x == _C*e^arcsinh(u(x))
```

Wir erhalten u nicht in expliziter Form. Um dem zu begegnen, werden wir mit dem Maxima-Befehl `ev` (wie evaluieren) und der Option `logarc=True`, die angibt, dass die Areafunktionen mit Hilfe von Logarithmen umgewandelt werden. Danach können wir u durch den Befehl `solve` erhalten:

```
sage: S = desolve(DE, u)._maxima_().ev(logarc=True).sage().solve(u); S
 [u(x) == -(sqrt(u(x)^2 + 1)*_C - x)/_C]
```

Hier ist `S` eine Liste, die nur aus einer Gleichung besteht; `S[0]` ist daher die Gleichung selbst.

Trotzdem wird die Gleichung nicht immer explizit gelöst. Wir werden Sage ein wenig zur Hand gehen, indem wir es zur Lösung der äquivalenten Gleichung

$$\_C^2(u^2+1) = (x - u \cdot \_C)^2$$

mit folgenden Befehlen auffordern:

```
sage: _C = S[0].variables()[0]
sage: solu = (x-S[0]*_C)^2; solu
(_C*u(x) - x)^2 == (u(x)^2 + 1)*_C^2
sage: sol = solu.solve(u); sol
[u(x) == -1/2*(_C^2 - x^2)/(_C*x)]
```

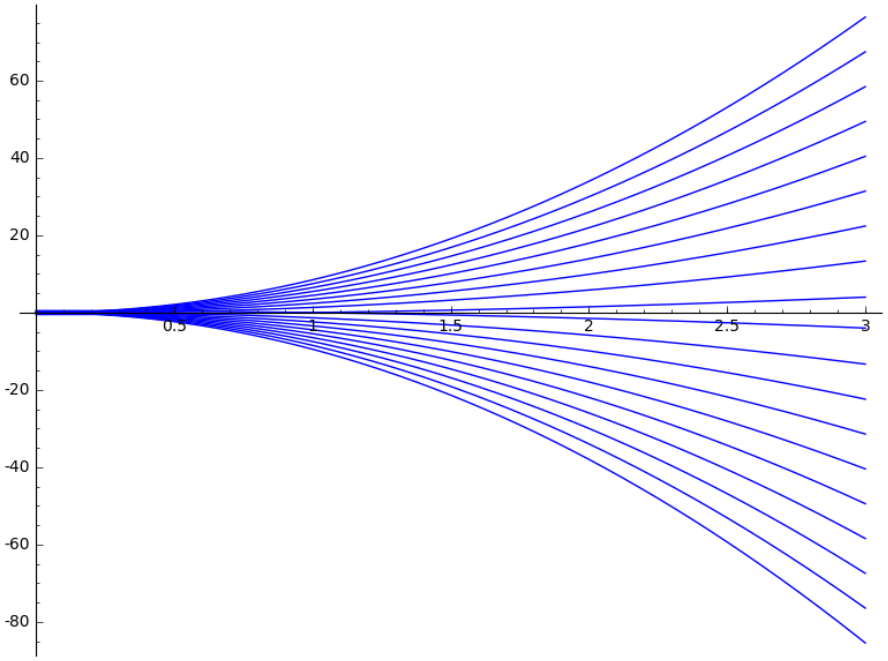
Wir müssen jetzt nur noch zurück zu y gelangen:

```
sage: y(x) = x*sol[0].rhs(); y(x)
-1/2*(_C^2 - x^2)/_C
```

Nunmehr erhalten wir die Lösungen in expliziter Form:

$$y(x) = \frac{x^2 - \_C^2}{2 \cdot \_C}.$$

Jetzt müssen die Lösungen auf $]0, \infty[$ noch gezeichnet werden, wobei darauf zu achten ist, dass $\_C$ nicht null wird.




```
sage: P = Graphics()
sage: for k in range(-19,19,2):
....:     P += plot(y(x).subs(_C == 1/k), x, 0, 3)
sage: P
```

Übung 39. *(Homogene Differentialgleichungen).* Lösen Sie auf $\mathbb{R}$ folgende Differentialgleichung: (E_3) : $xyy' = x^2 + y^2$.

Eine Parametergleichung: das Modell von Verhulst. Die relative Rate des Wachstums einer Bevölkerung ist eine mit der Größe der Bevölkerung linear abnehmende Funktion. Bei ihrer Untersuchung kann eine Gleichung der Form

$$y' = ay - y^2$$

zu lösen sein mit den positiv reellen Parametrern a und b.

```
sage: x = var('x'); y = function('y')(x); a, b = var('a, b')
sage: DE = diff(y,x) - a*y == -b*y^ 2
sage: sol = desolve(DE, [y,x]); sol
-(log(b*y(x) - a) - log(y(x)))/a == _C + x
```

Wir erhalten damit y nicht explizit. Versuchen wir, es mit `solve` zu isolieren.

```
sage: Sol = solve(sol, y)[0]; Sol
log(y(x)) == (_C + x)*a + log(b*y(x) - a)
```

Die explizite Lösung haben wir immer noch nicht. Wir werden die Terme auf die linke Seite umgruppieren und diesen Ausdruck mit Hilfe von `simplify_log()` vereinfachen:

```
sage: Sol(x) = S.lhs() - Sol.rhs(); Sol(x)
 -(_C + x)*a - log(b*y(x) - a) + log(y(x))
sage: Sol.simplify_log(); Sol(x)
-(_C + x)*a + log(y(x)/(b*y(x) - a))
sage: solve(Sol, y)[0].simplify()
y(x) == a*e^(_C*a + a*x)/(b*e^(_C*a + a*x) - 1)
```

10.1.3. Gleichungen 2. Odnung

Lineare Gleichungen mit konstanten Koeffizienten. Lösen wir jetzt eine lineare Gleichung 2. Ordnung mit konstanten Koeffizienten, zum Beispiel

$$y'' + 3y = x^2 - 7x + 31.$$

Wir verwenden die gleiche Syntax wie für Gleichungen 1. Grades. Die 2. Ableitung von y nach x bekommt man mit `diff(y,x,2)`.

```
sage: x = var('x'); y = function('y')(x)
sage: DE = diff(y,x,2) + 3*y == x^2 - 7*x + 31
sage: desolve(DE, y).expand()
1/3*x^2 + _K2*cos(sqrt(3)*x) + _K1*sin(sqrt(3)*x) - 7/3*x + 91/9
```

Fügen wir noch Anfangsbedingungen hinzu, beispielsweise $y(0) = 1$ und $y'(0) = 2$:

```
sage: desolve(DE, y, ics=[0,1,2]).expand()
1/3*x^2 + 13/9*sqrt(3)*sin(sqrt(3)*x) - 7/3*x - 82/9*cos(sqrt(3)*x) + 91/9
```

oder auch $y(0) = 1$ und $y(-1) = 0$:

```
sage: desolve(DE, y, ics=[0,1,-1,0]).expand()
1/3*x^2 - 7/3*x - 82/9*cos(sqrt(3))*sin(sqrt(3)*x)/sin(sqrt(3)) +
115/9*sin(sqrt(3)*x)/sin(sqrt(3)) - 82/9*cos(sqrt(3)*x) + 91/9
```

das heißt

$$\frac{1}{3}x^2 - \frac{7}{3}x - \frac{82\sin(\sqrt{3}x)\cos(\sqrt{3})}{9\sin(\sqrt{3})} + \frac{115\sin(\sqrt{3}x)}{9\sin(\sqrt{3})} - \frac{82}{9}\cos(\sqrt{3}x) + \frac{91}{9}.$$

Lösung einer PDG: die Wärmeleitungsgleichung. Wir untersuchen die berühmte Wärmeleitungsgleichung. Die Temperatur z verteilt sich in einem geraden homogenen Stab der Länge l gemäß der Gleichung (in der x die Abszisse längs des Stabes und t die Zeit ist):

$$\frac{\partial^2 z}{\partial x^2}(x,t) = C\frac{\partial z}{\partial t}(x,t).$$

Wir geben folgende Anfangs- und Randbedingungen vor:

$$\forall t \in \mathbb{R}^+ : \quad z(0,t) = 0 \quad z(l,t) = 0 \quad \forall x \in]0,l[: \quad z(x,0 = 1).$$

Wir suchen die nicht verschwindenden Lösungen der Form

$$z(x,t) = f(x)g(t).$$

Dies ist das Verfahren der Trennung der Variablen:

```
sage: x, t = var('x, t'); f = function('f')(x); g = function('g')(x)
sage: z = f*g
sage: eq(x,t) = diff(z,x,2) == diff(z,t); eq(z,t)
g(t)*diff(f(x), x, x) == f(x)*diff(g(t), t)
```

Das ist Gleichung

$$g(t)\frac{\mathrm{d}^2 f(x)}{\mathrm{d}\, x^2} = f(x)\frac{\mathrm{d}\, gt}{\mathrm{d}\, t}.$$

Wir dividieren durch $f(x)g(t)$, das wir ungleich null voraussetzen:

```
sage: eqn = eq/z; eqn(x,t)
diff(f(x), x, x)/f(x) == diff(g(t), t)/g(t)
```

Wir bekommen dann eine Gleichung, bei der jede Seite nur von einer Variablen abhängt:

$$\frac{1}{f(x)}\frac{\mathrm{d}\, 2f(x)}{\mathrm{d}\, x^2} = \frac{1}{g(t)}\frac{\mathrm{d}\, g(t)}{\mathrm{d}\, t}.$$

Beide Seiten müssen konstant sein, und wir führen eine Konstante k ein:

```
sage: k = var('k')
sage: eq1(x,t) = eqn(x,t).lhs() == k; eq2(x,t) = eqn(x,t).rhs() == k
```

Lösen wir jede der Gleichungen für sich und beginnen mit der zweiten;

```
sage: g(t) = desolve(eq2(x,t),[g,t]); g(t)
_C*e^(k*t)
```

also ist $g(t) = ce^{kt}$ mit einer Konstanten c.

Bei der ersten kommen wir nicht gleich weiter:

```
sage: desolve(eq1,[f,x])
Traceback (click to the left of this block for traceback)
...
Is k positive, negative or zero?
```

Verwenden wir `assume`:

```
sage: assume(k>0); desolve(eq1,[f,x])
_K1*e^(sqrt(k)*x) + _K2*e^(-sqrt(k)*x)
```

Das bedeutet

$$f(x) = k_1 e^{x\sqrt{k}} + k_2 e^{-x\sqrt{k}}.$$

10.1.4. Laplace-Transformation

Die Laplace-Transformation erlaubt die Umwandlung einer Differentialgleichung mit Anfangsbedingungen in eine algebraische Gleichung, und die Rücktransformation ermöglich dann die Rückkehr zu einer eventuell existierenden Lösung der Differentialgleichung. Zur Erinnerung: wenn f eine auf $\mathbb{R}$ definierte Funktion ist, die im Intervall $]-\infty, 0[$ verrschwindet, dann nennen wir die unter bestimmten Bedingungen durch

$$\mathcal{L}(f(x)) = F(s) = \int_0^{+\infty} e^{-sx} f(x)\,\mathrm{d}\,x$$

definierte Funktion F die Laplace-Transformierte von f.

Leicht erhalten wir die Laplace-Transformierten von polynomialen Funktionen, von trigonometrischen Funktionen, Exponentialfunktionen usw. Diese Transformierten haben höchst interessante Eigenschaften, insbesondere was die Transformierten einer Ableitung betrifft: wenn f' auf $\mathbb{R}_+$ stückweise stetig ist, dann ist

$$\mathcal{L}(f'(x)) = s\mathcal{L}(f(x)) - f(0),$$

und wenn f' den für f geltenden Bedingungen genügt, dann ist

$$\mathcal{L}(f''(x)) = s^2\mathcal{L}(f(x)) - sf(0) - f'(0).$$

Beispiel. Wir wollen die Differentialgleichung $y'' - 3y' - 4y = \sin(x)$ mit den Anfangsbedingungen $y(0) = 1$ und $y'(0) = -1$ mit Hilfe der Laplace-Transformierten lösen. Dann ist:

$$\mathcal{L}(y'' - 3y' - 4y) = \mathcal{L}(\sin(x)),$$

das heißt:

$$(s^2 - 3s - 4)\mathcal{L}(y) - sy(0) - y'(0) + ey(0) = \mathcal{L}(\sin(x)).$$

Sollten wir die Listen der Laplace-Transformierten der gängigen Funktionen vergessen haben, können wir die Transformierte des Sinus auch mit Sage finden:

```
sage: x, s = var('x, s'); f = function('f',x)
sage: f(x) = sin(x); f.laplace(x,s)
 x |--> 1/(s^2 + 1)
```

Somit erhalten wir einen Ausdruck für die Laplace-Transformierte von y:

$$\mathcal{L}(y) = \frac{1}{(s^2 - 3s - 4)(s^2 + 1)} + \frac{s - 4}{s^2 - 3s - 4}.$$

Mit Sage erhalten wir nun die Rücktransformierte:

```
sage: X(s) = 1/(s^2-3*s-4)/(s^2+1) + (s-4)/(s^2-3*s-4)
sage: X(s).inverse_laplace(s, x)
 3/34*cos(x) + 1/85*e^(4*x) + 9/10*e^(-x) - 5/34*sin(x)
```

Mit etwas Schummelei können wir $X(s)$ zunächst in einfachere Elemente zerlegen:

```
sage: X(s).partial_fraction()
 1/34*(3*s - 5)/(s^2 + 1) + 9/10/(s + 1) + 1/85/(s - 4)
```

Wir müssen nur noch eine Liste der Rücktransformationen zur Hand nehmen. Indessen könnten wir auch direkt die Blackbox `desolve_laplace` benutzen, die uns die Lösung sofort liefert:

```
sage: x = var('x'); y = function('y',x)
sage: eq = diff(y,x,x) - 3*diff(y,x) - 4*y - sin(x) == 0
sage: desolve_laplace(eq, y)
 1/85*(17*y(0) + 17*D[0](y)(0) + 1)*e^(4*x) + 1/10*(8*y(0) - 2*D[0](y)(0)
- 1)*e^(-x) + 3/34*cos(x) - 5/34*sin(x)
sage: desolve_laplace(eq, y, ics=[0,1,-1])
3/34*cos(x) + 1/85*e^(4*x) + 9/10*e^(-x) - 5/34*sin(x)
```

10.1.5. Lineare Differentialsysteme

Ein Beispiel eines einfachen linearen Differentialsystems 1. Ordnung. Wir wollen das folgende Differentialsystem lösen:

$$\begin{cases} y'(x) = A \cdot y(x) \\ y(0) = c \end{cases}$$

mit

$$A = \begin{bmatrix} 2 & -2 & 0 \\ -2 & 0 & 2 \\ 0 & 2 & 2 \end{bmatrix}, \qquad y(x) = \begin{bmatrix} y_1(x) \\ y_2(x) \\ y_3(x) \end{bmatrix}, \qquad c = \begin{bmatrix} 2 \\ 1 \\ -2 \end{bmatrix}.$$

Wir schreiben:

```
sage: x = var('x'); y1 = function('y1', x)
sage: y2 = function('y2', x); y3 = function('y3', x)
sage: y = vector([y1, y2, y3])
sage: A = matrix([[2,-2,0],[-2,0,2],[0,2,2]])
sage: system = [diff(y[i], x) - (A * y)[i] for i in range(3)]
sage: desolve_system(system, [y1, y2, y3], ics=[0,2,1,-2])
[y1(x) == e^(4*x) + e^(-2*x),
 y2(x) == -e^(4*x) + 2*e^(-2*x),
 y3(x) == -e^(4*x) - e^(-2*x)]
```

Für die Anfangsbedingungen gilt diese Syntax: `ics = [x0,y1(x0),y2(x0),y3(x0)]`.

Mit einer Matrix komplexer Eigenwerte. Diesmal nehmen wir

$$A = \begin{bmatrix} 3 & -4 \\ 1 & 3 \end{bmatrix}, \qquad c = \begin{bmatrix} 2 \\ 0 \end{bmatrix}.$$

Mit Sage:

```
sage: x = var('x'); y1 = function('y1', x); y2 = function('y2', x)
sage: y = vector([y1,y2])
sage: A = matrix([[3,-4],[1,3]])
sage: system = [diff(y[i], x) - (A * y)[i] for i in range(2)]
sage: desolve_system(system, [y1, y2], ics=[0,2,0])
 [y1(x) == 2*cos(2*x)*e^(3*x), y2(x) == e^(3*x)*sin(2*x)]
```

kommt

$$\begin{cases} y_1(x) = 2\cos(2x)e^{3x} \\ y_2(x) = \sin(2x)e^{3x}. \end{cases}$$

Ein System 2. Ordnung. Wir wollen dieses System lösen:

$$\begin{cases} y_1''(x) - 2y_1(x) + 6y_2(x) - y_1'(x) - 3y_2'(x) = 0 \\ y_2''(x) - 2y_1(x) - 6y_2(x) - y_1'(x) - y_2'(x) = 0 \end{cases}$$

Wir führen es auf ein System 1. Ordnung zurück und setzen dazu

$$u = (u_1, u_2, u_3, u_4) = (y_1, y_2, y_1', y_2').$$

Dann haben wir

$$\begin{cases} u_1' = u_3 \\ u_2' = u_4 \\ u_3' = 2u_1 - 6u_2 + u_3 + 3u_4 \\ u_4' = -2u_1 + 6u_2 + u_3 - u_4, \end{cases}$$

das heißt $u' = A \cdot u(x)$ mit

$$A = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 2 & -6 & 1 & 3 \\ -2 & 6 & 1 & -1 \end{bmatrix}.$$

Mit Sage:

```
sage: x = var('x'); u1 = function('u1', x); u2 = function('u2', x)
sage: u3 = function('u3', x); u4 = function('u4', x)
sage: u = vector([u1,u2,u3,u4])
sage: A = matrix([[0,0,1,0],[0,0,0,1],[2,-6,1,3],[-2,6,1,-1]])
sage: system = [diff(u[i], x) - (A*u)[i] for i in range(4)]
sage: sol = desolve_system(system, [u1, u2, u3, u4])
```

Wir berücksichtigen nur die beiden ersten Komponenten, denn es sind y_1 und y_2, die uns interessieren, d.h. u_1 und u_2:

```
sage: sol[0]
u1(x) == 1/12*(2*u1(0) - 6*u2(0) + 5*u3(0) + 3*u4(0))*e^(2*x) +
         1/24*(2*u1(0) - 6*u2(0) - u3(0) + 3*u4(0))*e^(-4*x) + 3/4*u1(0) +
         3/4*u2(0) - 3/8*u3(0) - 3/8*u4(0)
```

```
sage: sol[1]
u2(x) == -1/12*(2*u1(0) - 6*u2(0) - u3(0) - 3*u4(0))*e^(2*x) -
         1/24*(2*u1(0) - 6*u2(0) - u3(0) + 3*u4(0))*e^(-4*x) + 1/4*u1(0) +
         1/4*u2(0) - 1/8*u3(0) - 1/8*u4(0)
```

Das kann bei guter Beobachtungsgabe zusammengefasst werden zu

$$\begin{cases} y_1(x) = k_1 e^{2x} + k_2 e^{-4x} + 3k_3 \\ y_2(x) = k_4 e^{2x} - k_2 e^{-4x} + k_3 \end{cases}$$

mit k_1, k_2, k_3 und k_4 als Parametern, die von den Anfangsbedingungen abhängen.

10.2. Rekursiv definierte Folgen

10.2.1. Durch $u_{n+1} = f(u_n)$ definierte Folgen

Definition einer Folge. Wir betrachten eine durch die Beziehung $u_{n+1} = f(u_n)$ mit $u_0 = a$ rekursiv definierte Folge. Natürlich können wir die Folge mit einem rekursiven Algorithmus definieren. Nehmen wir als Beispiel eine logistische Folge (eine durch eine Beziehung der Form $x_{n+1} = rx_n(1 - x_n)$ definierte Folge):

$$f : x \mapsto 3.83 \cdot \left(1 - \frac{x}{100000}\right) \quad \text{und } u_0 = 20000.$$

Diese Folge können wir in Sage definieren durch

Differentialgleichungen	
Variablendeklaration	`x=var('x')`
Funktionsdeklaration	`y=function('y')(x)`
Lösung einer Gleichung	`desolve(gleichung, y, <optionen>)`
Lösung eines Systems	`desolve_system([gl1, ...]], [y1, ...]], <optionen>)`
Anfangsbedingungen 1. Ordnung	$[x_0, y(x_0)]$
Anfangsbedingungen 2. Ordnung	$[x_0, y(x_0), x_1, y(x_1)]$
	$[x_0, y(x_0), y'(x_0)]$
Anfangsbedingungen für Systeme	$[x_0, y_1(x_0), y_2(x_0), \ldots]$
unabhängige Variable	`ivar=x`
Lösungsverfahren	`show_method=True`
Aufruf sdpezieller Verfahren	`contrib_ode=True`
Laplace-Transformierte	
Transformation einer Funktion $f : x \mapsto f(x)$	`f.laplace(x,s)`
Rücktransformation von $X(s)$	`X(s).inverse_laplace(s,x)`
Lösung durch Laplace-Transformation	`desolve_laplace(gleichung,funktion)`
Verschiedene Befehle	
Ausdruck der 1. Ableitung	`diff(y,x)`
ausmultiplizierte Form eines Ausdrucks	`expr.expand()`
In einem Ausdruck vorkommende Variablen	`expr.variables()`
Substitution einer Variablen	`expr.subs(var==val)`
rechte Seite eine Gleichung	`gleichung.rhs()`
linke Seite einer Gleichung	`gleichung.lhs`

Tab. 10.1 - Befehle für die Lösung von Differentialgleichungen

```
sage: x = var('x'); f = function('f',x)
sage: f(x) = 3.83*x*(1 - x/100000)
sage: def u(n):
....:     if n==0: return(20000)
....:     else: return f(u(n-1))
```

Wir können eine iterative Definition vorziehen:

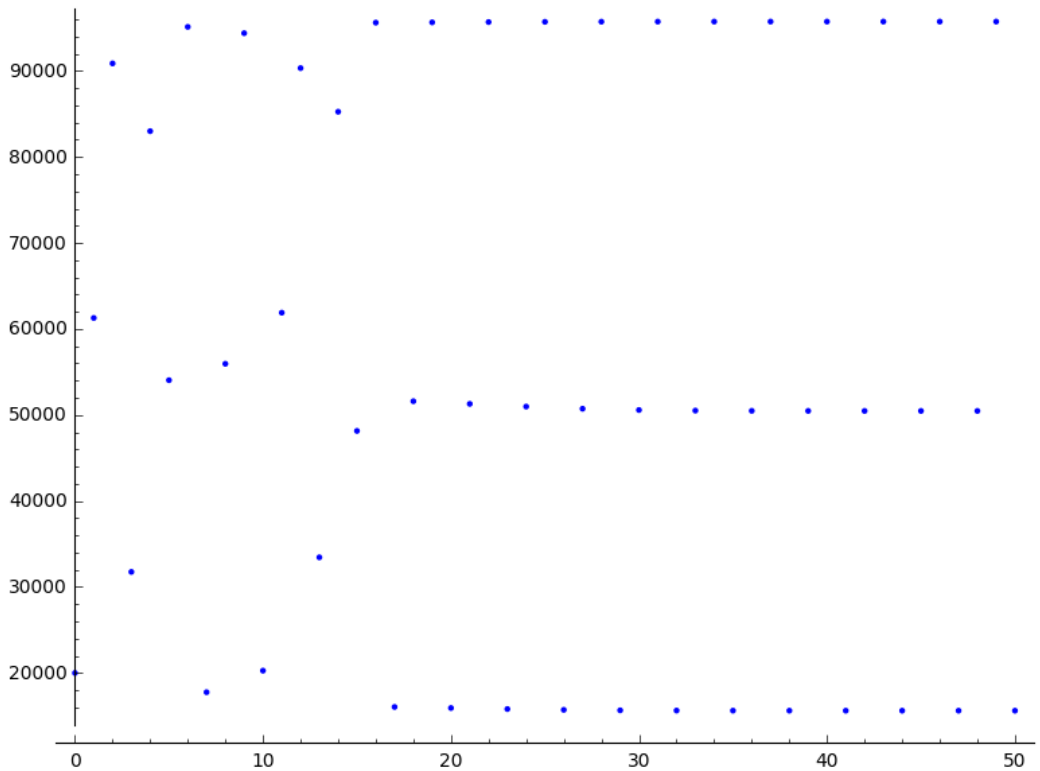
```
sage: def v(n):
....:     V = 20000;
....:     for k in [1..n]:
....:         V = f(V)
....:     return V
```

Graphische Darstellung. Wir können die Punkte mit den Koordinaten (k, u_k) zeichnen lassen:

```
sage: def wolke(u,n):
....:     L = [[0,u(0)]];
....:     for k in [1..n]:
....:         L += [[k,u(k)]]
....:     points(L).show()
```

Ausgehend von der folgenden Abbildung vermuten wir die Existenz von drei Häufungspunkten:

```
sage: wolke(u,50)
```



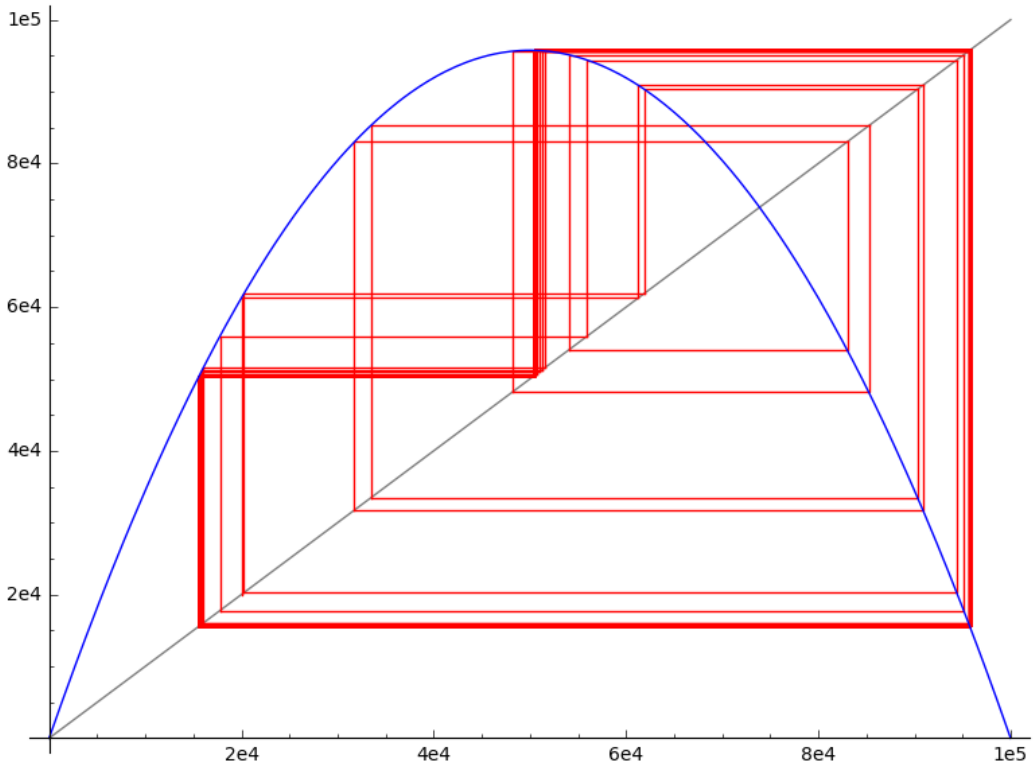
Wir hätten stattdessen auch eine Darstellung wählen können, wo der Graph von f auf die Winkelhalbierende trifft. Da diese nicht von sich aus in Sage vorhanden ist, werden wir eine kleine Prozedur schreiben, welche die Arbeit erledigt:

```
sage: def schnecke(f,x,u0,n,xmin,xmax):
....:     u = u0
....:     P = plot(x, x, xmin, xmax, color='gray')
....:     for i in range(n):
....:         P += line([[u,u],[u,f(u)],[f(u),f(u)]], color = 'red')
....:         u = f(u)
....:     P += f.plot(x, xmin, xmax, color='blue') # Courbe de f
....:     P.show()
```

Beispielsweise beobachten wir für die gleiche Folge:

```
sage: f(x) = 3.83*x*(1 - x/100000)
sage: schnecke(f,x,20000,100,0,100000)
```

ebenfalls drei Häufungspunkte:



10.2.2. Lineare rekursiv definierte Folgen

Sage kann auch mit Folgen des Typs

$$a_k u_{n+k} + a_{k-1} u_{n+k-1} + \cdots + a_1 u_{n+1} + a_0 u_n = 0$$

umgehen, wobei $(a_i)_{0 \le i \le k}$ eine Familie von Skalaren ist.

Wir betrachen als Beispiel die Folge

$$u_0 = -1,\ u_1 = 1,\ u_{n+2} = \frac{3}{2}u_{n+1} - \frac{1}{2}u_n.$$

In Sage 7 ist die bekannte Funktion `rsolve` nicht direkt nutzbar. Man findet sie in `Sympy`, was gewisse Unanehmlichketen zur Folge hat, wie die Syntaxänderung bei der Deklaration der Variablen. Hier der nötige Vorspann für die Definition der Folge:

```
sage: from sympy import Function, Symbol
sage: u = Function('u'); n = Symbol('n', integer=True)
```

Anschließend muss die lineare Relation in der Form $a_k u_{n+k} + \ldots + a_0 u_n = 0$ definiert werden:

```
sage: f = u(n+2)-(3/2)*u(n+1)+(1/2)*u(n)
```

Schließlich rufen wir `rsolve` auf und achten dabei genau darauf, wie die Anfangsbedingungen deklariert werden:

```
sage: from sympy import rsolve
sage: rsolve(f, u(n), {u(0):-1,u(1):1})
3 - 4*2**(-n)
```

das heißt $u_n = 3 - \frac{1}{2^{n-2}}$.

Bemerkung. Zu beachten sind gewisse Einschränkungen bei `rsolve`. Zum Beispiel wird bei rekursiv definierten Folgen 2. Ordnung ein korrektes Resultat nur erhalten, wenn die charakteristische Gleichung zwei verschiedene reelle Wurzeln hat wie im vorstehendem Beispiel. Sonst drückt Sage die Lösung mit Hilfe des Pochhammer-Symbols aus (`RisingFactorial(x,n)`), und das ist wenig ergiebig.

10.2.3. Rekursiv definierte Folgen „mit zweitem Glied“

Sage kann auch mit Folgen dieses Typs umgehen:

$$a_k(n)u_{n+k} + a_{k-1}(n)u_{n+k-1} + \ldots + a_1(n)u_{n+1} + a - 0(n)u_n = f(n)$$

mit einer Familie von Polynomen $(a_i)_{0\leq i\leq k}$ in n und einer polynomialen, einer gebrochen rationalen oder einer hypergeometrischen Funktion f vonh n.

Entsprechend der Natur von $f(n)$ sind die Befehle verschieden:

- `rsolve_poly` wenn f eine Polynomfunktion ist,
- `rsolve_ratio`, wenn f eine gebrochen rationale Funktion ist,
- `rsolve_hyper`, wenn f eine hypergeometrische Funktion ist.

Wir definieren die $a_i(n)$ als Liste $[a_0(n), \ldots, a_{k-1}(n), a_k(n)]$. Um beispielsweise die Komplexität von Mergesort zu untersuchen, sind wir gehalten, die Folge

$$u_{n+1} = 2u_n + 2^{n+2},\ u_0 = 0$$

zu studieren. Die Rechnung ergibt

```
sage: from sympy import rsolve_hyper
sage: n = Symbol('n', integer=True)
sage: rsolve_hyper([-2,1],2**(n+2),n)
2**n*C0 + 2**(n + 2)*(C0 + n/2)
```

und wegen $u_0 = 0$ ist `C0=0`, daher ist $u_n = n \cdot 2^{n+1}$.

Teil III.

Numerisches Rechnen

11. Gleitpunktzahlen

In den folgenden Kapiteln stehen die Gleitpunktzahlen im Zentrum der Aufmerksamkeit. Es ist sinnvoll, sie zu studieren, denn ihr Verhalten folgt genauen Regeln.

Wie sollen reelle Zahlen maschinell dargestellt werden? Da diese Zahlen im allgemeinen nicht mit einer endlichen Menge an Information kodiert werden können, sind sie auf einem Rechner nicht immer darstellbar: man muss sie also durch eine Größe mit endlichem Speicherbedarf annähern.

Es hat sich zur näherungsweisen Darstellung reeller Zahlen mit einer festen Menge von Information ein Standard herausgebildet: die Gleitpunkt-Darstellung.[1]

In diesem Kapitel findet man: eine summarische Beschreibung der Gleitpunktzahlen und verschiedener in Sage verfügbarer Typen dieser Zahlen sowie die Demonstration einiger ihrer Eigenschaften. Etliche Beispiele zeigen bestimmte Schwierigkeiten, denen man beim Rechnen mit Gleitpunktzahlen begegnet, einige Kniffe, mit denes es gelegentlich gelingt, sie zu umschiffen, wobei wir erwarten, dass der Leser die nötige Bedachtsamkeit entwickelt. Als Konsequenz versuchen wir einige Eigenschaften zu benennen, welche die numerischen Methoden haben müssen, um mit diesen Zahlen verwendet werden zu können.

Für weitere Informationen kann der Leser die im Netz verfügbaren Dokumente [BZ10] und [Go191] konsultieren oder das Buch [MBdD+10].

11.1. Einführung

11.1.1. Definition

Eine Menge $F(\beta, r, m, M)$ von Gleitpunktzahlen ist durch vier Parameter definiert: eine Basis $\beta \geq 2$, eine Anzahl von Ziffern r und zwei ganze Zahlen m bzw. M. Die Elemente von $F(\beta, r, m, M)$ sind Zahlen der Form

$$x = (-1)^s \, 0.d_1 d_2 \ldots d_r \cdot \beta^j,$$

worin die Ziffern d_i ganze Zahlen sind, die $0 \leq d_i < \beta$ für $i \geq 1$ und $0 < d_1 < \beta$ erfüllen. Die Anzahl r der Ziffern ist die *Genauigkeit*, das Vorzeichenbit s kann 0 oder 1 sein, der *Exponent* j liegt zwischen den beiden ganzen Zahlen m und M und $0.d_1 d_2 \ldots d_r$ ist die *Mantisse*.

[1] In der deutschsprachigen Literatur hält sich noch der Begriff Gleitkomma, auch wenn in der Informatik grundsätzlich der Punkt als Dezimaltrennzeichen dient und das Komma als Trennzeichen in Listen und dergleichen fungiert.

11.1.2. Eigenschaften, Beispiele

Die Normalisierung $0 < d_1 < \beta$ garantiert, dass alle Zahlen dieselbe Anzahl von signifikanten Ziffern haben. Man bemerkt, dass die Zahl null mit der Konvention $d_1 > 0$ nicht darstellbar ist: null braucht eine spezielle Darstellung.

Als Beispiel: die -0.028 gschriebene Zahl zur Basis 10 (Festpunkt-Darstellung) wird als $-0.28 \cdot 10^{-1}$ dargestellt (vorbehaltlich natürlich, dass $r \geq 2$ und $m \leq -1 \leq M$). Da zur rechnerinternen Binärdarstellung die Basis 2 dient, ist β gleich 2 in den verschiedenen von Sage angebotenen Mengen von Gleitpunktzahlen und wir werden uns daher im Verlaufe dieses Kapitels immer in diesem Rahmen bewegen. Um ein Beispiel zu geben, $0.101 \cdot 2^1$ stellt mit den Werten aus der Menge $F(2, 3, -1, 2)$ die Zahl $5/4$ dar..

Da für $\beta = 2$ der einzige mögliche Wert von d_1 gleich $d_1 = 1$ ist, kann er in einer Maschinenimplementierung weggelassen werden und, immer mit den Werten aus der Menge $F(2, 3, -1, 2)$, kann beispielsweise $5/4$ mit 5 bits, nämlich als 00110 maschinencodiert werden, worin die erste Ziffer von links das Vorzeichen + darstellt, die beiden folgenden Ziffern, nämlich 01 die Mantisse (101) und die letzten beiden Ziffern den Exponenten (00 codiert den Wert -1 des Exponenten, 01 den Wert 0 usw.).

Dem Leser muss völlig klar sein, dass die Mengen $F(\beta, r, m, M)$ nur eine endliche Untermenge der reellen Zahlen beschreiben. Zur Darstellung einer reellen Zahl x, die zwischen zwei aufeinander folgenden Zahlen aus $F(\beta, r, m, M)$ liegt, wird eine *Rundung* genannte Anwendung benötigt, die festlegt, welche Zahl eine Näherung von x ist: man kann die Zahl nehmen, die x am nächsten liegt, doch sind auch andere Entscheidungen möglich; die Norm bestimmt, dass die Rundung $F(\beta, r, m, M)$ unverändert lässt. Die Menge der darstellbaren Zahlen ist beschränkt, und die Mengen der Gleitpunktzahlen enthalten auch die Sonderwerte $+\infty$ und $-\infty$, die nicht nur unendliche Werte (wie $1/0$) darstellen[2], sondern auch alle Werte oberhalb der größten darstellbaren positiven Zahl (oder unterhalb der kleinsten darstellbaren negativen Zahl). Es gibt auch ein Zeichen zur Darstellung undefinierter Operationen wie $0/0$.

11.1.3. Normalisierung

Nach einigen tastenden Versuchen wird die Notwendigkeit einer Norm spürbar, damit identische Programme auf vschiedenen Rechnern gleiche Resiultate liefern. Seit 1985 definiert die Norm IEEE-754[3] mehrere Zahlenmengen, darunter die in 64 Bits gespeicherten Zahlen „doppelter Genauigkeit“): Das Vorzeichen s ist mit 1 Bit codiert, die Mantisse bekommt 53 Bits (wovon nur 52 gespeichert werden), der Exponent 11 Bits. Die Zahlen haben die Gestalt

$$(-1)^s \, 0.d_1 d_2 \ldots d_{53} \cdot 2^{j-1023}.$$

Sie entsprechen dem Typ `double` in C.

[2] Diese Festlegungen entsprechen der Norm IEEE-754.
[3] siehe https://de.wikipedia.org/wiki/IEEE_754

11.2. Die Gleitpunktzahlen

Sage stellt zwei Arten von Gleitpunktzahlen bereit:

1. Gleitpunktzahlen „doppelter Genauigkeit“, wie in 11.1.3 beschrieben: das sind die vom Prozessor des Rechners gelieferten Zahlen. In Sage gehören diese Zahlen zur Klasse RDF:

   ```
   sage: xrdf = RDF(3.0)
   ```

2. Gleitpunktzahlen beliebiger Genauigkeit: jede Instanz der Klasse `RealField` definiert eine Menge von Gleitpunktzahlen gegebener Genauigkeit (und eventuell mit einem gegebenen Rundungsmodus: siehe 11.3.2). Um eine Zahl `x100` mit 100 Bits Genauigkeit zu deklarieren, wird man beispielsweise schreiben:

   ```
   sage: R100 = RealField(100) # Genauigkeit: 100 bit
   sage: x100 = R100(3/8); x100
   0.37500000000000000000000000000
   ```

Die Zahlen der Menge RealField(p) haben die Form

$$(-1^s)\,0.d_1d_2\ldots d_p\cdot 2^e$$

mit $s\in\{0,1\}$: die Mantisse ist p Binärziffern breit, und e kann 30 Bits haben (auf manchen Maschinen auch mehr). Man kann die Genauigkeit auch offen lassen:

```
sage: Rdefaut = RealField() # voreingestellte Genauigkeit 53 Bits
sage: xdefault = Rdefault(2/3)
```

und man kann die Genauigkeit aller Gleitpunktzahlen mit der Methode `prec()` testen:

```
sage: xrdf.prec()
53
sage: x100.prec()
100
sage: xdefault.prec()
53
```

Also haben die Zahlen der Menge `RealField()` und die der Menge RDF dieselbe Genauigkeit, doch akzeptiert `RealField()` auch größere Exponenten. Die Menge `RealField()` mit der Genauigkeit 53 Bits ist in Sage der voreingestellte Typ für „reelle“ Zahlen.

```
sage: x = 1.0; print type(x)
<type 'sage_rings.real_mpfr.RealLiteral'>
sage: x.prec()
53
```

Hier zeigt `real_mpfr.RealLiteral` an, dass die Menge der Zahlen, zu der x gehört, mit der Bibliothek GNU MPFR implementiert ist.

Erinnern wir uns, dass der Typ einer Zahl durch die rechte Seite einer Zuweisung automatisch definiert ist:

```
sage: x = 1.0          # x gehört zum RealField()
sage: x = 0.1e+1       # idem: x gehört zum RealField()
```

```
sage: x = 1           # x ist eine ganze Zahl
sage: x = RDF(1)      # x ist Gleitpunktzahl doppelter Maschinengenauigkeit
sage: x = RDF(1.)     # idem: x ist Gleitpunktzahl doppelter Genauigkeit
sage: x = RDF(0.1e+1) # idem
sage: x = 4/3         # x ist eine rationale Zahl
sage: R = RealField(20)
sage: x = R(1)        # x ist Gleitpunktzahl mit 20 Bits Genauigkeit
```

und es werden normale Konversionen rationaler Zahlen bewerkstelligt:

```
sage: RDF(8/3)
2.66666666667
sage: R100 = RealField(100); R100(8/3)
2.6666666666666666666666666667
```

sowie auch die Konversion zwischen Zahlen, die zu verschiedenen Mengen von Gleitpunktzahlen gehören:

```
sage: x = R100(8/3)
sage: R = RealField(); R(x)
2.66666666666667
sage: RDF(x)
2.66666666667
```

Die verschiedenen Mengen von Gleitpunktzahlen enthalten auch die speziellen Werte +0, -0, +infinity, -infinity und NaN:

```
sage: 1.0/0.0
+infinity
sage: RDF(1)/RDF(0)
+infinity
sage: |RDF(-1.0)/RDF(0.)|
-infiniy
```

Der Wert NaN ist nicht definierten Ergebnissen zugeordnet.

```
sage: 0.0/0.0
NaN
sage: RDF(1)/RDF(0)
NaN
```

x	R2	RDF(x).ulp()	R100(x).ulp()
10^{-30}	3.9e-31	1.751623080406e-46	1.24460305557222834142881281 08e-60
10^{-10}	2.9e-11	1.292469707114e-26	9.18354961579912115600575419 70e-41
10^{-3}	0.00049	2.168404344980e-19	1.54074395550978868244478235 41e-33
1	0.50	2.220446049260e-16	1.57772181044202361082345713 06e-30
10^{3}	510.	1.136868377216e-13	8.07793566946316088741610050 85e-28
10^{10}	4.3e9	1.907348632812e-6	1.35525271560688054250931600 11e-20
10^{30}	3.2e29	1.407374883554e14	1.00000000000000000000000000

Tabelle 11.1 - Abstände zwischen Gleitpunktzahlen.

11.2.1. Welchen Zahlentyp wählen?

Die Gleitpunktzahlen beliebiger Genauigkeit ermöglichen Rechnungen sehr hoher Genauigkeit, während die Genauigkeit für die Zahlen RDF konstant ist. Im Gegensatz dazu rufen Rechnungen mit Zahlen aus `RealField` die Softwarebibliothek GNU MPFR auf, sodass bei den Zahlen RDF die Rechnungen durch die Gleitpunkt-Arithmetik des Prozessors ausgeführt werden, was erheblich schneller ist. Wir geben in 13.2.10 einen Vergleich, wo sich die Effizienz der Maschinenzahlen mit dem Gebrauch der für diese Zahlen optimierten Bibliotheken verbindet. Halten wir fest, dass unter den numerischen Berechnungsverfahren, denen man in den folgenden Kapiteln begegnet, viele sind, die allein die Gleitpunktzahlen RDF verwenden, und, was immer man auch tut, Gleitpunktzahlen jeweils in dieses Format konvertieren.

R2, eine Zahlenmenge zum Spielen. Die Gleitpunktzahlen beliebiger Genauigkeit, die für Rechnungen, die hohe Genauigkeit verlangen, unersetzlich sind, erlauben daneben auch die Definition einer Klasse von Gleitpunktzahlen, die sehr ungenau sind und deshalb quasi als deren Karikatur Eigenschaften der Gleitpunktzahlen verdeutlichen: die Menge R2 der Zahlen mit 2 Bits Genauigkeit.

```
sage: R2 = RealField(2)
```

11.3. Einige Eigenschaften der Gleitpunktzahlen

11.3.1. Mengen voller Löcher

In jeder Menge von Gleitpunktzahlen gibt die Methode `ulp()` *(unit in the last place)* die Größe des Intervalls an, das eine Zahl von der nächstgelegenen darstellbaren Zahl (in der Richtung von null weg) trennt.

```
sage: x2 = R2(1.); x2.ulp()
0.50
sage: xr = 1.; xr.ulp()
2.22044604925031e-16
```

Der Leser wird den für `x2.ulp()` angegebenen Wert leicht verifizieren können.

Die Tabelle 11.1 gibt die Größe des Intervalls an, das eine Zahl x - oder genauer $R(x)$, wobei R die betrachtete Menge ist - von ihrem nächsten Nachbarn trennt (in der Richtung von null weg) für verschiedene Zahlenmengen (R100 ist die Menge `RealField(100)`) und verschiedene Werte von x.

Wie leicht zu sehen ist, wächst die Ausdehnung der *Löcher*, die zwei aufeinander folgende Zahlen trennen, mit der Größe der Zahlen.

Übung 40 *(ein etwas überraschender Wert).* Zu zeigen ist, dass R100(10^{30}).ulp() genau 1.0000000000000000000000000000 ist.

11.3.2. Das Runden

Wie ist eine Zahl anzunähern, die in einer Menge von Gleitpunktzahlen nicht enthalten ist? Man kann das Runden auf verschiedene Weisen definieren:

- zur nächstgelegenen darstellbaren Zahl hin - so wird in der Menge RDF vorgegangen und das ist auch das voreingestellte Verhalten der mit `RealField` erzeugten Zahlen. Für eine Zahl genau zwischen zwei darstellbaren Zahlen wird zur geraden Mantisse hin gerundet;
- in Richtung $-\infty$; `RealField(p,rnd='RNDD')` ergibt dieses Verhalten mit einer Genauigkeit von p Bits;
- in Richtung null; Beispiel: `RealField(p,rnd='RNDZ')`;
- in Richtung $+\infty$; Beispiel: `RealField(p,rnd='RNDU'`.

11.3.3. Einige Eigenschaften

Das notwendige Runden in Mengen von Gleitpunktzahlen ruft etliche gefährliche Effekte hervor. Sehen wir uns einige davon genauer an.

Eine gefährliche Erscheinung. Hierbei handelt es sich um die als *Auslöschung* (engl. *catastrophic cancellation*) bekannte Einbuße an Genauigkeit, die aus der Subtraktion zweier benachbarter Zahlen folgt: genauer handelt es sich um eine Vergrößerung der Fehler:

```
sage: a = 10000.0; b = 9999.5; c = 0.1; c
0.100000000000000
sage: a1 = a+c # a wird beeinträchtigt
sage: a1-b
0.600000000000364
```

Hier bewirkt der an a angebrachte *Fehler* `c` die ungenaue Berechnung der Differenz (die letzten drei Stellen sind falsch).

Anwendung: Berechnung der Wurzeln eines Trinoms zweiten Grades. Sogar die Lösung einer Gleichung zweiten Grades $ax^2 + bx + c = 0$ kann Probleme bereiten. Betrachten wir den Fall $a = 1.0$, $b = 10.0^4$, $c = 1$:

```
sage: a = 1.0; b = 10.0^4; c = 1
sage: delta =b^2-4*a*c
sage: x = (-b-sqrt(delta)))/(2*a); y = (-b+sqrt(delta)))/(2*a)
sage: x, y
(-9999.99990000000, -0.000100000001111766)
```

Die Summe der Wurzeln ist korrekt, das Produkt aber nicht:

```
sage: x+x+b/a
0.000000000000000
sage: x*y-c/a
1.11766307320238e-9
```

Der Fehler resultiert aus der Auslöschung, die eintritt, wenn man `-b` und `sqrt(delta)` addiert, nachdem `y` berechnet ist. Hier kann man eine bessere Näherung von `y` erzielen:

```
sage: y = (c/a)/x; y
-0.000100000001000000
sage: x+y+b/a
0.000000000000000
```

Wir stellen fest, dass bei der Rundung die Summe der Wurzeln korrekt bleibt. Der Leser mag alle für die Wahl von `a, b` und `c` möglichen Fälle ins Auge fassen, um sich zu überzeugen, dass das Schreiben eines robusten numerischen Programms zur Berechnung der Wurzeln eines Trinoms 2. Grades keine simple Angelegenheit ist.

Die Mengen der Gleitpunktzahlen sind bezüglich der Addition keine Gruppen. Tatsächlich ist die Addition nicht assoziativ. Nehmen wir die Menge `R2` (mit 2 Bits Genauigkeit):

```
sage: x1 = R2(1/2); x2 = R2(4); x3 = R2(-4)
sage: x1, x2, x3
(0.50, 4.0, -4.0)
sage: : x1+(x2+x3)|
0.50
\sage: (x1+x2)+x3
0.00
```

Daraus können wir folgern, dass die verschiedenen möglichen Reihenfolgen der Rechnungen in einem Programm nicht ohne Einfluss auf das Ergebnis sind!

Rekursiv definierte Folgen mit Gleitpunktzahlen. Betrachten wir[4] die rekursiv definierte Folge $u_{n+1} = 4u_n - 1$. Für $u_0 = 1/3$ ist die Folge stationär: $u_i = 1/3$ für alle i.

```
sage: x = RDF(1/3)
sage: for i in range(1,100): x = 4*x-1; print x
0.333333333333
0.333333333333
0.333333333333
...
-1.0
-5.0
-21.0
-85.0
-341.0
-1365.0
-5461.0
-21845.0
...
```

Die berechnete Folge divergiert! Wir können erkennen, dass dieses Verhalten ziemlich selbstverständlich ist, denn es handelt sich um eine klassische Erscheinung von Instabilität. Ein Fehler von u_0 wird bei jeder Iteration mit 4 multipliziert, und wir wissen, dass die Gleitpunktarithmetik Fehler mit sich bringt, die dann bei jeder Iteration vergrößert werden.

[4] Dank an Marc Deléglise (Institut Camille Jordan, Lyon) für dieses Beispiel.

Berechnen wir nun die rekursiv definierte Folge $u_{n+1} = 3u_n - 1$ mit $u_0 = 1/2$. Wir erwarten ein ähnliches Ergebnis: die exakt berechnete Folge ist konstant, doch ein Fehler wird bei jedem Schritt verdreifacht.

```
sage: x = RDF(1/2)
sage: for i in range(1,100): x = 3*x-1; print x
0.5
0.5
0.5
...
0.5
```

Diesmal bleibt die Folge konstant. Wie sind die unterschiedlichen Verhaltensweisen zu erklären? Betrachten wir jeweils die binäre Darstellung von u_0.

Im ersten Beispiel ($u_{n+1} = 4u_u - 1$) haben wir

$$\frac{1}{3} = \frac{1}{4}\sum_{i=0}^{\infty}\frac{1}{4^i} = \frac{1}{4}\sum_{i=0}^{\infty}\frac{1}{2^{2i}},$$

also ist $1/3$ in den Mengen der Gleitpunktzahlen, über die wir verfügen (mit $\beta = 2$), nicht exakt darstellbar, mit welcher Genauigkeit auch immer. Der Leser ist eingeladen, die vorstehende Rechnung mit Zahlen großer Genauigkeit, zum Beispiel aus `RealField(1000)` zu wiederholen, um zu verifizieren, dass die berechnete Folge immer divergiert. Wir wollen erwähnen, dass wenn man im ersten Programm die Zeile

```
sage: x = RDF(1/3)
```

ersetzt durch

```
sage: x = 1/3,
```

dass dann die Rechnungen in der Menge der rationalen Zahlen ausgeführt werden und die iterierten Werte immer $1/3$ ergeben.

Im zweiten Beispiel ($u_{n+1} = 3u_n - 1$) werden u_0 und $3/2$ mit der Basis 2 als 0.1 bzw. 1.1 geschrieben; sie sind daher mit den verschiedenen Mengen von Gleitpunktzahlen ohne Rundung exakt darstellbar. Die Rechnung ist somit genau und die Folge bleibt konstant.

Die folgende Übung zeigt, dass eine Folge, die mit einer Menge von Gleitpunktzahlen programmiert ist, gegen einen falschen Grenzwert konvergieren kann.

Übung 41 *(Beispiel von Jean-Michel Muller).* Man betrachte die rekursiv definierte Folge (siehe [MBsD+10, S. 9]):

$$u_n = 111 - \frac{1130}{u_{n-1}} + \frac{3000}{u_{n-1}u_{n-2}}.$$

Es kann gezeigt werden, dass die allgemeine Lösung diese Form hat:

$$u_n = \frac{\alpha 100^{n+1} + \beta 6^{n+1} + \gamma 5^{n+1}}{\alpha 100^n + \beta 6^n + \gamma 5^n}.$$

1. Man wähle $u_0 = 2$ und $u_1 = -4$: welches sind die Werte von α, β und γ? Gegen welchen Grenzwert konvergiert die Folge?

2. Die rekursiv definierte Folge ist zu programmieren (immer mit $u_0 = 2$ und $u_1 = -4$) mit der Menge `RealField()` (oder mit `RDF`). Was kann festgestellt werden?

3. Erläutern Sie dieses Verhalten.

4. Führen Sie die gleiche Rechnung mit einer Menge großer Genauigkeit aus, beispielsweise mit `RealField(5000)`. Kommentieren Sie das Ergebnis.

5. Die Folge ist auf $\mathbb{Q}$ definiert. Programmieren Sie sie mit rationalen Zahlen und kommentieren Sie das Ergebnis.

Gleitpunktzahlen und die Summenbildung bei numerischen Reihen. Wir betrachten eine reelle numerische Reihe mit dem positiven allgemeinen Term u_n. Die Berechnung der Partialsummen $\sum_{i=0}^{m} u_i$ mit einer Menge von Gleitpunktzahlen wird durch Rundungsfehler gestört werden. Der Leser kann sich damit vergnügen zu zeigen, dass u_n gegen 0 strebt, wenn n über alle Grenzen wächst, und wenn die Partialsummen im Bereich der darstellbaren Zahlen bleiben, dann ist ab einem bestimmten m die mit Rundung berechnete Folge $\sum_{i=0}^{m} u_i$ stationär (siehe [Sch91]). Kurzum, in der Welt der Gleitpunktzahlen ist das Leben einfach: Reihen deren positiver allgemeiner Term gegen 0 tendiert, konvergieren unter der Bedingung, dass die Partialsummen nicht zu sehr anwachsen!

Schauen wir uns das bei der (divergenten) harmonischen Reihe mit dem Term $u_n = 1/n$ an:

```
sage: def sumharmo(p):
....:     RFP = RealField(p)
....:     y = RFP(1.); x = RFP(0.); n = 1
....:     while x <> y:
....:         y = x; x += 1/n; n += 1
....:     return p, n, x
```

Testen wir diese Funktion mit verschiedenen Genauigkeiten `p`:

```
sage: sumharmo(2)
(2, 5, 2.0)
sage: sumharmo(20)
(20, 131073, 12.631)
```

Der Leser kann mit Papier und Stift nachvollziehen, dass mit unserer Spielmenge von Gleitpunktzahlen `R2` die Funktion nach 5 Iterationen gegen den Wert 2.0 konvergiert. Offenbar hängt das Ergebnis von der Genauigkeit `p` ab; und ebenso kann der Leser (immer mit Papier und Stift) verifizieren, dass für $n > \beta^p$ die berechnete Summe stationär ist. Trotzdem Vorsicht: mit der voreingestellten Genauigkeit von 53 Bits und bei 10^9 Operationen pro Sekunde braucht es $2^{53}/10^9/3600$ Stunden, das sind etwa 104 Tage, bis man einen stationären Wert erhält!

Verbesserte Berechnung bestimmer rekursiv definierter Folgen. Mit einiger Vorsicht ist es möglich, manche Resultate zu verbessern: hier ein Beispiel.

Häufig tritt eine rekursiv definierte Folge dieser Form auf:

$$y_{n+1} = y + \delta_n,$$

wobei die Zahlen δ_n dem absoluten Wert nach klein sind gegenüber y_n: zu denken ist beispielsweise an die Integration von Differentialgleichungen in der Himmelsmechanik zur Simulation des Sonnensystems, wo große Werte (von Entfernungen oder Geschwindigkeiten) über lange Zeiträume kleine Störungen erleiden [HLW02]. Auch wenn wir die Terme δ_n genau berechnen können, führen die Rundundungsfehler bei der Addition $y_{n+1} = y + \delta_n$ zu beträchtlichen Abweichungen. Nehmen wir als Beispiel die durch $y_0 = 10^{13}$, $\delta_0 = 1$ und $\delta_{n+1} = a\delta_n$ mit $a = 1 - 10^{-8}$ definierte Folge. Hier das naive Programm zur Berechnung von y_n:

```
sage: def iter(y,delta,a,n):
....:     for i in range(0,n):
....:         y += delta
....:         delta *= a
....:     return y
```

Mit den rationalen Werten, die wir für y_0, δ_0 und a gewählt haben, können wir mit Sage den exakten Wert der Iterierten exakt berechnen:

```
sage: def exakt(y,delta,a,n):
....:     return y+delta*(1-a^n)/(1-a)
```

Berechnen wir jetzt 100000 Iterierte (zum Beispiel) mit Gleitpunktzahlen `RDF` und vergleichen das Ergebnis mit dem genauen Wert:

```
sage: y0 = RDF(10^13); delta0 = RDF(1); a = RDF(1-10^(-8)); n = 100000
sage: ii = iter(y0,delta0,a,n)
sage: s = exact(10^13,1,1-10^(-8),n)
sage: print "exakt - klassische Summierung:", s-ii
exakt - klassische Summierung: -45.498046875
```

Jetzt der Algorithmus für die *kompensierte Summierung*:

```
sage: def sumcomp(y,delta,e,n,a):
....:     for i in range(0,n):
....:         b = y
....:         e += delta
....:         y = b+e
....:         e += (b-y)
....:         delta = a*delta # nouvelle valeur de delta
....:     return y
```

Um dessen Verhalten zu verstehen, betrachten wir das untenstehende Schema (wir folgen hier den Darstellungen in [Hig93] und [HLW02]), worin die Kästen die Mantissen der Zahlen darstellen. Die Position der Kästen entspricht dem Exponenten (je weiter links der Kasten steht, desto größer ist der Exponent):

$b = y_n$	b_1	b_2		
e			e_1	0
δ_n		δ_1	δ_2	
$e = e + \delta_n$		δ_1	$e_1 + \delta_2$	
$y_{n+1} = b + e$	b_1	$b_2 + \delta_1$		
$e = e + (b - y_{n+1})$			$e_1 + \delta_2$	0

Der Rundungsfehler wird in e angesammelt, und wir stellen fest, dass keine Ziffer von δ_n verloren geht, während mit dem naiven Verfahren die Bits δ_2 aus der Rechnung verschwinden.

Führen wir die Berechnung der 100000 Iterierten mit dem Verfahren der kompensierten Summierung nochmals aus:

```
sage: c = sumcomp(y0,delta0,RDF(0.0),n,a)
sage: print "exakt - kompensierte Summierung: %.5f" %\
....:         RDF(s-RR(c).exact_rational())
exakt - kompensierte Summierung: -0.00042
```

Der absolute Fehler beträgt mit dem naiven Algorithmus -45.5, und -0.00042 mit der kompensierten Summierung! Nebenbei sehen wir auch, dass wir für die relativen Fehler bei naiver Summierung $4.55 \cdot 10^{-12}$ bekommen und $4.16 \cdot 10^{-17}$ bei kompensierter Summierung.

11.3.4. Komplexe Gleitpunktzahlen

Sage schlägt zwei Familien von komplexen Zahlen vor, die rechnerintern als Paare von Gleitpunktzahlen dargestellt werden, wie sie in den vorstehehenden Beispielen schon aufgetreten sind:

1. Komplexe Zahlen doppelter Genauigkeit `ComplexDoubleField` (oder abgekürzt `CDF`). Es sind Zahlen der Form $x + i \cdot y$, wobei x und y Gleitpunktzahlen doppelter Genauigkeit sind. Die Zahlen werden wie folgt erzeugt:

   ```
   sage: x = CDF(2,1.); x
   2.0 + 1.0*I
   sage: y = CDF(20,0); y
   20.0
   ```

   oder auch so:

   ```
   sage: z = ComplexDoubleElement(2.,1.); z
   2.0 + 1.0*I
   ```

2. Komplexe Zahlen beliebiger Genauigkeit `ComplexField`. Das sind Zahlen der Form $x + i \cdot y$, wobei x und y dieselbe Genauigkeit von p Bits besitzen. Eine Instanz der Klasse `ComplexField` erzeugt ein Zahlenpaar gegebener Genauigkeit (voreingestellt ist 53):

   ```
   sage: C = ComplexField(); C(2,3)
   2.00000000000000 + 3.00000000000000*I
   sage: C100 = ComplexField(100); C100(2,3)
   2.0000000000000000000000000000 + 3.0000000000000000000000000000*I
   ```

Wohlgemerkt, Rechnungen mit diesen Zahlen werfen die gleichen Rundungsprobleme auf wie die mit reellen Gleitpunktzahlen.

11.3.5. Die Methoden

Wir haben bereits die Methoden `prec` und `ulp` kennengelernt. Die verschiedenenen Zahlenmengen, denen wir hier begegnen, stellen eine große Anzahl weiterer Methoden bereit. Geben wir einige Beispiele:

- Methoden, die Konstanten zurückgeben. Beispiele:

```
sage: R200 = RealField(200); R200.pi()
3.1415926535897932384626433832795028841971693993751058209749
sage: R200.euler_constant()
0.57721566490153286060651209008240243104215933593992359880577
```

- trigonometrische Funktionen `sin, cos, arcsin, arccos` usw. Beispiele:

```
sage: x = RDF.pi()/2; x.cos() # Gleitpunkt-Näherungswert für null!
6.123233995736757e-17
sage: x.cos().arccos() - x
0.0
```

- Logarithmen (`log, log10, log2` usw.), hyperbolische Winkelfunktionen und deren Inverse (`sinh, arcsinh, cosh, arccosh` usw.),

- spezielle Funktionen (`gamma, j0, j1, jn(k)` usw.)

Der Leser möge die Dokumentation von Sage konsultieren, um eine vollständige Liste der sehr zahlreichen Methoden zu bekommen, die verfügbar sind. Wir erinnern uns, dass diese Liste online erhalten werden kann durch Eingabe von

```
sage: x = 1.0; x.
```

und ansschließendes Drücken der Tabulator-Taste. Zu jeder Methode erhalten wir ihre Definition, ihre eventuellen Parameter und ein Beispiel für die Anwendung, wenn wir eingeben (hier für die *Gamma*-Funktion von Euler $\Gamma(x)$):

```
sage: x.gamma?
```

Mengen der Gleitpunktzahlen	
reelle Zahlen mit Genauigkeit p Bits	`RealField(p)`
Maschinen-Gleitpunktzahlen	`RDF`
komplexe Zahlen mit Genauigkeit p Bits	`ComplexField(p)`
komplexe Maschinenzahlen	`CDF`

Tab. 11.2 - Zusammenfassung: Gleitpunktzahlen

11.4. Schlussbemerkungen

Die in Sage implantierten numerischen Verfahren, die in den folgenden Kapiteln beschrieben werden, sind alle theoretisch untersucht worden; ihre theoretische Analyse besteht unter anderem in der Untersuchung des Konvergenzverhaltens der iterativen Methoden, des durch Vereinfachung eines Problems verursachten Fehlers, um es überhaupt berechenbar zu machen, aber auch des Verhaltens der Rrechnungen bei Auftreten von Störungen, zum Beispiel solchen die durch die ungenaue Gleitpunktarithmetik entstehen.

Betrachten wir einen Algorithmus $\mathcal{F}$, der aus den Daten d das Ergebnis $x = \mathcal{F}(d)$ berechnet. Dieser Algorithmus wird nur brauchbar sein, wenn er die Fehler von d nicht übermäßig vergrößert; einer Störung ε von d entspricht eine gestörte Lösung $x_\varepsilon = \mathcal{F}(d+\varepsilon)$. Der entstehende Fehler $x_\varepsilon - x$ muss in vernünftiger Weise von ε abhängen (er muss stetig sein, nicht zu schnell wachsen,...): die Algorithmen müssen Stabilität besitzen, um brauchbar zu sein. In Kapitel 13 erörtern wir die Stabilitätsprobleme der in der linearen Algebra benutzten Algorithmen.

Wir müssen auch sagen, dass bestimmte Probleme mit endlicher Genauigkeit definitiv nicht realisierbar sind wie beispielsweise die Berechnung der in Übung 41 gegebenen Folge: jede Störung, so klein sie auch sein mag, bewirkt die Konvergenz der Folge gegen einen falschen Grenzwert. Dies ist eine typische Erscheinung der Instabilität bei dem Problem, das wir zu lösen versuchen: die experimentelle Untersuchung einer Folge mittels Gleitpunktzahlen muss mit großer Vorsicht erfolgen.

Der Leser kann die Praxis des Rechnens mit Gleitpunktzahlen vielleicht für hoffnungslos halten, doch dieses Urteil muss abgemildert werden: die große Mehrheit der für das Rechnen verfügbaren Ressourcen wird für Operationen auf diesen Zahlenmengen benutzt: Näherungslösungen von partiellen Differentialgleichungen, Optimierungen der Signalverarbeitung usw. Die Gleitpunktzahlen müssen mit Misstrauen betrachtet werden, sie haben die Entwicklung der Rechnung und deren Anwendung aber nicht verhindert: es sind nicht die Rundungsfehler, die die Zuverlässigkeit der Wettervorhersage begrenzen, um nur dieses Beispiel zu erwähnen.

12. Nichtlineare Gleichungen

Dieses Kapitel erklärt, wie mit Sage eine nichtlineare Gleichung zu *lösen* ist. In einem ersten Abschnitt untersuchen wir polynomiale Gleichungen und zeigen die Grenzen bei der Suche nach exakten Lösungen. Danach beschreiben wir die Wirkungsweise einiger klassischer Methoden für eine numerischen Lösung. Dabei weisen wir darauf hin, welche Algorithmen für die Lösung in Sage implementiert sind.

12.1. Algebraische Gleichungen

Unter einer algebraischen Gleichung verstehen wir eine Gleichung der Form $p(x) = 0$, wobei p ein Polynom einer Unbestimmten bezeichnet, dessen Koeffizienten zu einem Integritätsring A gehören. Wir sagen, dass ein Element $\alpha \in A$ eine *Wurzel* des Polynoms p ist, wenn $p(\alpha) = 0$ gilt.

Sei α ein Element von A. Die euklidsche Division von p durch $p - \alpha$ sichert die Existenz eines konstanten Polynoms r, sodass gilt:

$$p = (x - \alpha)q + r.$$

Wenn wir diese Gleichung in α auswerten, bekommen wir $r = p(\alpha)$. Daher wird p von dem Polynom $x - \alpha$ genau dann geteilt, wenn α eine Wurzel von p ist. Diese Bemerkung erlaubt die Einführung des Begriffs der *Vielfachheit* einer Wurzel α eines Polynoms: es handelt sich um die größte ganze Zahl m, sodass $(x - \alpha)^m$ p teilt. Wir stellen fest, dass die Summe der Vielfachheiten aller Wurzeln von p kleiner oder gleich dem Grad von p ist.

12.1.1. Die Methode `Polynomial.roots()`

Die Lösung der algebraischen Gleichung $p(x) = 0$ besteht in der Identifizierung der Wurzeln des Polynoms p mitsamt ihrer Vielfachheiten. Die Methode `Polynomial.roots()` liefert die Wurzeln eines Polynoms. Sie erhält bis zu drei Parameter, die alle optional sind. Der Parameter `ring` ermöglicht die Präzisierung, in welchem Ring die Wurzeln zu suchen sind. Wird für diesen Parameter kein Wert festgelegt, wird der Ring der Koeffizienten des Polynoms angenommen. Der boolesche Wert `multiplicities` bezeichnet die Art der von `Polynomial.roots()` zurückgegebenen Informationen: jede Wurzel kann von ihrer Vielfacheit begleitet werden. Der Parameter `algorithm` dient zur Festlegung des zu verwendenden Algorithmus; die möglichen Werte werden im weiteren Verlauf erklärt (siehe Unterabschnitt 12.2.2).

```
sage: R.<x> = PolynomialRing(RealField(prec=10))
sage: p = 2*x^7 - 21*x^6 + 64*x^5 - 67*x^4 + 90*x^3 + 265*x^2 - 900*x + 375
sage: p.roots()
[(-1.7, 1), (0.50, 1), (1.7, 1), (5.0, 2)]
sage: p.roots(ring=ComplexField(10), multiplicities=False)
[-1.7, 0.50, 1.7, 5.0, -2.2*I, 2.2*I]
```

```
sage: p.roots(ring=RationalField())
[(1/2, 1), (5, 2)]
```

12.1.2. Zahlendarstellung

Erinnern wir uns, wie in Sage die gängigen Ringe bezeichnet werden (siehe Abschnitt 5.2). Die ganzen Zahlen werden durch Objekte der Klasse `Integer` dargestellt, und für Umwandlungen wird der *Vorfahr* `ZZ` benutzt oder die Funktion `IntegerRing`, der das Objekt `ZZ` zurückgibt. Genauso werden die rationalen Zahlen durch Objekte der Klasse `Rational` dargestellt; der gemeinsame Vorfahr dieser Objekte ist das Objekt `QQ`, das die Funktion `RationalField` zurückgibt. In beiden Fällen verwendet Sage für Rechnungen mit beliebiger Genauigkeit die Bibliothek GMP. Ohne auf Einzelheiten der Realisierung dieser Bibliothek einzugehen, sind die mit Sage manipulierten ganzen Zahlen von beliebiger Größe, die einzige Beschränkung ergibt sich aus der verfügbaren Speichergröße der Maschine, auf der das Programm läuft.

Es stehen mehrere genäherte Darstellungen reeller Zahlen zur Verfügung (siehe Kapitel 11). Es gibt ein `RealField()` für Darstellungen mit Gleitpunktzahlen einer gegebener Genauigkeit, insbesondere `RR` mit einer Genauigkeit von 53 Bits. Es gibt aber auch `RDF` und die Funktion `RealDoubleField()` für Maschinenzahlen doppelter Genauigkeit; und dann noch die Klassen `RIF` und `RealIntervalField`, bei denen eine reelle Zahl durch ein Intervall dargestellt wird, in dem sie liegt. Die Grenzen dieses Intervalls sind Gleitpunktzahlen.

Die analogen Darstellungen von komplexen Zahlen heißen: `CC, CDF` und `CIF`. Auch hier ist jedem Objekt eine Funktion zugeordnet; es sind `ComplexField()`, `ComplexDoubleField`, und `ComplexIntervalField()`.

Die von `Polynomial.roots()` ausgeführten Rechnungen sind exakt oder je nach Art der Darstellung der Koeffizienten des Polynoms bzw. einem eventuellen Wert des Parameters `ring` genähert: beispielsweise sind die Rechnungen mit `ZZ` oder `QQ` exakt, mit `RR` genähert. Im zweiten Teil dieses Kapitels präzisieren wir den Algorithmus für die Berechnung der Wurzeln und die Rolle der Parameter `ring` und `algorithm` (siehe Abschnitt 12.2).

Die Lösung algebraischer Gleichungen ist mit dem Zahlbegriff eng verbunden. Der *Zerlegungskörper* des Polynoms p (als nicht konstant vorausgesetzt) ist die kleinste Erweiterung des Körpers der Koeffizienten von p, worin p ein Produkt der Polynome 1. Grades ist; wir zeigen, dass eine solche Erweiterung immer existiert. Mit Sage können wir den Zerlegungskörper eines irreduziblen Polynoms mit der Methode `Polynomial.root_field()` bilden. Danach können wir mit den im Zerlegungskörper implizit enthaltenen Wurzeln rechnen.

```
sage: R.<x> = PolynomialRing(QQ, 'x')
sage: p = x^4 + x^3 + x^2 + x + 1
sage: K.<alpha> = p.root_field()
sage: p.roots(ring=K, multiplicities=None)
[alpha, alpha^2, alpha^3, -alpha^3 - alpha^2 - alpha - 1]
sage: alpha^5
1
```

12.1.3. Der Gauß-d'Alembertsche Fundamentalsatz der Algebra

Der Zerlegungskörper des Polynoms mit reellen Koeffizienten $x^2 + 1$ ist nichts anderes als der Körper der komplexen Zahlen. Es ist bemerkenswert, dass jedes nicht konstante Polynom mit komplexen Koeffizienten mindestens eine komplexe Wurzel besitzt: das besagt der *Fundamentalsatz der Algebra*. In der Konsequenz ist jedes komplexe, nicht konstante Polynom ein Produkt von Polynomen 1. Grades. Etwas weiter oben haben wir bemerkt, dass die Summe der Vielfachheiten der Wurzeln eines Polynoms p kleiner oder gleich dem Grad von p ist. Wir wissen jetzt, dass im Körper der komplexen Zahlen diese Summe genau gleich dem Grad von p ist. Anders gesagt besitzt jede polynomiale Gleichung n-ten Grades n komplexe Wurzeln, die samt ihrem Vielfachheiten gezählt werden.

Schauen wir einmal, wie die Methode `Polynomial.roots()` dieses Resultat zu illustrieren vermag. Im folgenden Beispiel bilden wir den Ring der Polynome mit reellen Koeffizienten (wir begnügen uns mit einer Darstellung, die mit Gleitpunktzahlen der Genauigkeit 53 Bit arbeitet). Dann wird aus diesem Ring ein Polynom mit einem Grad unter 15 zufällig ausgewählt. Schließlich summieren wir Vielfachheiten der mit der Methode `Polynomial.roots()` berechneten komplexen Wurzeln auf und vergleichen diese Summe mit dem Grad des Polynoms.

```
sage: R.<x> = PolynomialRing(RR, 'x')
sage: d = ZZ.random_element(1, 15)
sage: p = R.random_element(d)
sage: p.degree() == sum(r[1] for r in p.roots(CC))
True
```

12.1.4. Verteilung der Wurzeln

Wir fahren fort mit einer überraschenden Demonstration der Leistungsfähigkeit der Methode `Polynomial.roots`: wir zeichnen alle Punkte der komplexen Ebene, deren Zahlenwert Wurzel eines Polynoms 12. Grades mit den Koeffizienten 1 oder -1 ist. Die Wahl des Grades ist ein mehr oder weniger willkürlicher Kompromiss; wir erhalten damit eine genaue Zeichnung in annehmbarer Zeit. Der Gebrauch von Näherungswerten für die komplexen Zahlen erfolgt ebenfalls aus Leistungsgründen (siehe Kapitel 13).

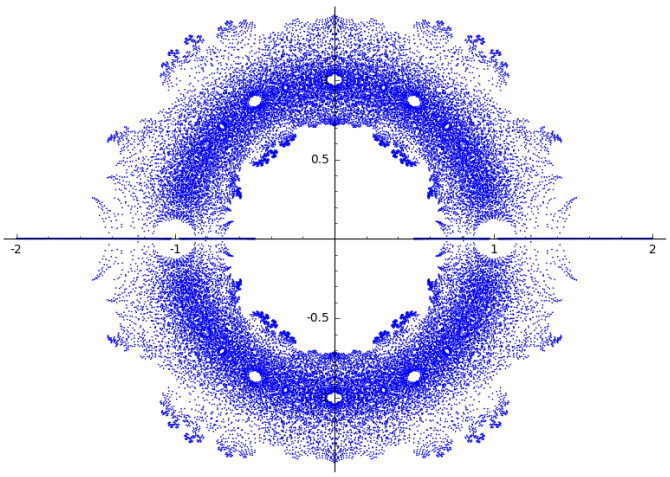


Abb. 12.1 - Verteilung der Wurzeln aller Polynome 12. Grades mit Koeffizienten 1 oder -1.

```
sage: def build_complex_roots(degree):
....:     R.<x> = PolynomialRing(CDF, 'x')
....:     v = []
....:     for c in cartesian_product([[-1, 1]] * (degree + 1)):
....:         v.extend(R(list(c)).roots(multiplicities=False))
....:     return v
sage: data = build_complex_roots(12)
sage: points(data, pointsize=1, aspect_ratio=1)
```

12.1.5. Lösung durch Wurzelausdrücke

In bestimmten Fällen ist es möglich, die exakten Werte der Wurzeln eines Polynoms zu berechnen. Das ist beispielsweise der Fall, sobald man die Wurzeln als Funktion der Koeffizienten oder durch Wurzelausdrücke (Quadratwurzeln, Kubikwurzeln usw.) ausdrücken kann. In solchen Fällen sprechen von *Lösung durch Wurzelausdrücke*.

Um diesen Lösungstyp mit Sage auszuführen, müssen wir mit Objekten der Klasse `Expression` arbeiten, die symbolische Ausdrücke darstellen. Wir haben gesehen, dass die durch Objekte der Klasse `Integer` dargestellten ganzen Zahlen denselben *Vorfahr* haben, nämlich das Objekt `ZZ`. Ebenso haben die Objekte der Klasse `Expression` einen gleichen Vorfahr: es handelt sich um das Objekt `SR` (kurz für *Symbolic Ring*); er bietet nicht zuletzt Möglichkeiten der Konversion zur Klasse `Expression`.

Quadratische Gleichungen.

```
sage: a, b, c, x = var('a, b, c, x')
sage: p = a * x^2 + b * x + c
sage: type(p)
<type 'sage.symbolic.expression.Expression'>
sage: p.parent()
sage: Symbolic Ring
sage: p.roots(x)
[(-1/2*(b + sqrt(b^2 - 4*a*c))/a, 1),
(-1/2*(b - sqrt(b^2 - 4*a*c))/a, 1)]
```

Ein Grad größer als 2. Es ist auch möglich, komplexe algebraische Gleichungen 3. und 4. Grades mit Wurzelausdrücken zu lösen. Umgekehrt ist es unmöglich, polynomiale Gleichungen höheren als 4. Grades mit Wurzelausdrücken zu lösen (siehe Unterabschnitt 7.3.4). Diese Unmöglichkeit lässt uns numerische Lösungsverfahren ins Auge fassen (siehe Abschnitt 12.2).

```
sage: a, b, c, d, e, f, x = var('a, b, c, d, e, f, x')
sage: p = a*x^5+b*x^4+c*x^3+d*x^2+e*x+f
sage: p.roots(x)
Traceback (most recent call last):
...
RuntimeError: no explicit roots found
```

Wir wollen mit Sage ein Lösungsverfahren für Gleichungen 3. Grades auf dem Körper der komplexen Zahlen vorführen. Zuerst zeigen wir, dass sich die allgemeine Gleichung 3. Grades auf die Form $x^3 + px + q = 0$ zurückführen lässt.

```
sage: x, a, b, c, d = var('x, a, b, c, d')
sage: P = a * x^3 + b * x^2 + c * x + d
sage: alpha = var('alpha')
sage: P.subs(x = x + alpha).expand().coefficient(x, 2)
3*a*alpha + b
sage: P.subs(x = x - b / (3 * a)).expand().collect(x)
a*x^3 - 1/3*(b^2/a - 3*c)*x + 2/27*b^3/a^2 - 1/3*b*c/a + d
```

Um die Wurzeln einer Gleichung der Form $x^3+px+q = 0$ zu erhalten, setzen wir $x = u+v$.

```
sage: p, q, u, v = var('p, q, u, v')
sage: P = x^3 + p * x + q
sage: P.subs(x = u + v).expand()
u^3 + 3*u^2*v + 3*u*v^2 + v^3 + p*u + p*v + q
```

Wir nehmen den letzten Ausdruck gleich null an. Wir sehen dann, dass $u^3 + v^3 + q = 0$ gleich $3uv + p = 0$ ist; und weiter, wenn die Gleichheit verifiziert ist, sind u^3 und v^3 Wurzeln einer Gleichung 2. Grades: $(X - u^3)(X - v^3) = X^2 - (u^3 + v^3)X + (uv)^3 = X^2 + qX - p^3/27$.

```
sage: P.subs({x: u + v, q: -u^3 - v^3}).factor()
sage: (3*u*v + p)*(u + v)
sage: P.subs({x: u+v, q: -u^3 - v^3, p: -3 * u * v}).expand()
0
sage: X = var('X')
sage: solve([X^2 + q*X - p^3 / 27 == 0], X, solution_dict=True)
[{X: -1/2*q - 1/18*sqrt(12*p^3 + 81*q^2)},
 {X: -1/2*q + 1/18*sqrt(12*p^3 + 81*q^2)}]
```

Die Lösungen der Gleichung $x^3 + px + q = 0$ sind daher die Summen $u + v$, wobei u und v die Kubikwurzeln sind von

$$-\frac{\sqrt{4p^3 + 27q^2}\sqrt{3}}{18} - \frac{q}{2} \quad \text{und} \quad \frac{\sqrt{4p^3 + 27q^2}\sqrt{3}}{18} - \frac{q}{2}$$

was $3uv + p = 0$ verifiziert.

12.1.6. Die Methode `Expression.roots()`

Die vorstehenden Beispiele verwenden die Methode `Expression.roots()`. Diese Methode gibt eine Liste mit exakten Wurzeln zurück, doch ohne die Garantie, tatsächlich alle zu finden. Unter den optionalen Parametern dieser Methode findet man die Parameter `ring` und `multiplicities`, die uns schon bei der Methode `Polynomial.roots()` begegnet sind. Es ist wichtig, sich daran zu erinnern, dass die Methode `Expression.roots()` nicht allein auf polynomiale Ausdrücke angewendet wird.

```
sage: e = sin(x) * (x^3 + 1) * (x^5 + x^4 + 1)
sage: roots = e.roots(); len(roots)
9
sage: roots
```

```
[(0, 1),
 (-1/2*(1/18*sqrt(23)*sqrt(3) - 1/2)^(1/3)*(I*sqrt(3) + 1)
 - 1/6*(-I*sqrt(3) + 1)/(1/18*sqrt(23)*sqrt(3) - 1/2)^(1/3), 1),
 (-1/2*(1/18*sqrt(23)*sqrt(3) - 1/2)^(1/3)*(-I*sqrt(3) + 1)
 - 1/6*(I*sqrt(3) + 1)/(1/18*sqrt(23)*sqrt(3) - 1/2)^(1/3), 1),
 ((1/18*sqrt(23)*sqrt(3) - 1/2)^(1/3) + 1/3/(1/18*sqrt(23)*sqrt(3)
 - 1/2)^(1/3), 1),
 (-1/2*I*sqrt(3) - 1/2, 1), (1/2*I*sqrt(3) - 1/2, 1),
 (1/2*I*sqrt(3)*(-1)^(1/3) - 1/2*(-1)^(1/3), 1),
 (-1/2*I*sqrt(3)*(-1)^(1/3) - 1/2*(-1)^(1/3), 1), ((-1)^(1/3), 1)]
```

Da in Sage der Parameter `ring` nicht definiert ist, delegiert die Methode `roots()` der Klasse `Expression` die Berechnung der Wurzeln an das Programm Maxima, das den Ausdruck zu faktorisieren sucht, um dann eine Lösung mit Wurzelausdrücken für jeden Faktor auszugeben, dessen Grad kleiner ist als 5. Wenn der Parameter `ring` definiert ist, wird der Ausdruck in ein Objekt der Klasse `Polynomial` konvertiert, dessen Koeffizienten als Vorfahr das durch den Parameter `ring` festgelegte Objekt haben; danach wird das Resultat der Methode `Polynomial.roots()` zurückgegeben. Im weiteren Verlauf werden wir den hier verwendeten Algorithmus beschreiben (siehe Unterabschnitt 12.2.2).

Wir stellen ebenfalls Rechenbeispiele mit impliziten Wurzeln vor, zu denen wir über die Objekte `QQbar` und `AA` gelangen, welche den Körper der algebraischen Zahlen darstellen (siehe Unterabschnitt 7.3.2).

Elimination von Mehrfachwurzeln. Ist ein Polynom p mit Mehrfachwurzeln gegeben, dann kann ein Polynom mir Einfachwurzeln (d.h. mit der Vielfachheit 1) gebildet werden, das mit p identisch ist. Wenn wir also die Wurzeln eines Polynoms berechnen, können wir immer davon ausgehen, dass die Wurzeln einfach sind. Weisen wir nun die Existenz eines Polynoms mit Einfachwurzeln nach und sehen wir zu, wie es gebildet wird. Das gestattet uns, eine neue Veranschaulichung der Methode `Expression.roots()` zu geben.

Sei α eine Wurzel des Polynoms p, deren Vielfachheit m eine Zahl größer als 1 ist. Das ist eine Wurzel des abgeleiteten Polynoms p' mit der Vielfachheit $m-1$. In der Tat, wenn $p = (x-\alpha)^m q$ ist, dann haben wir $p' = (x-\alpha)^{m-1}(mq + (x-\alpha)q')$.

```
sage: alpha, m, x = var('alpha, m, x'); q = function('q')(x)
sage: p = (x - alpha)^m * q
sage: p.derivative(x)
(-alpha + x)^(m - 1)*m*q(x) + (-alpha + x)^m*diff(q(x), x)
sage: simplify(p.derivative(x)(x=alpha))
0
```

In der Konsequenz ist der ggT von p und p' das Produkt $\prod_{\alpha\in\Gamma}(x-\alpha)^{m_\alpha-1}$ mit Γ als der Menge der Wurzeln von p mit einer Vielfachheit größer als 1, und m_α ist die Vielfachheit der Wurzel α. Wenn d diesen ggT bezeichnet, dann hat der Quotient p durch d die erwarteten Eigenschaften.

Wir bemerken, dass der Grad des Quotienten von p durch d kleiner als der Grad von p ist. Wenn insbesondere der Grad kleiner ist als 5, dann ist es möglich, die Wurzeln des Polynoms mit Wurzelausdrücken anzugeben. Das folgende Beispiel veranschaulicht dies für ein Polynom 13. Grades mit rationalen Koeffizienten.

```
sage: R.<x> = PolynomialRing(QQ, 'x')
p = 128 * x^13 - 1344 * x^12 + 6048 * x^11 - 15632 * x^10 \
....: + 28056 * x^9 - 44604 * x^8 + 71198 * x^7 - 98283 * x^6 \
....: + 105840 * x^5 - 101304 * x^4 + 99468 * x^3 - 81648 * x^2 \
....: + 40824 * x - 8748
sage: d = gcd(p, p.derivative())
sage: (p // d).degree()
4
sage: roots = SR(p // d).roots(multiplicities=False)
sage: roots
[1/2*I*sqrt(3)*2^(1/3) - 1/2*2^(1/3),
-1/2*I*sqrt(3)*2^(1/3) - 1/2*2^(1/3), 2^(1/3), 3/2]
sage: [QQbar(p(alpha)).is_zero() for alpha in roots]
[True, True, True, True]
```

12.2. Numerische Lösung

In der Mathematik ist es Tradition, *diskret* und *kontinuiertlich* gegenüberzustellen. Die numerische Analyse relativiert diesen Gegensatz auf gewisse Art: ein wichtiger Aspekt der numerischen Analyse besteht tatsächlich darin, sich Fragen bezüglich der reellen Zahlen zuzuwenden, im Kern also dem Kontinuum, indem wir einen experimemtellen Standpunkt einnehmen, der sich auf den Rechner stützt, der seinerseits vom Diskreten abhängt.

Wenn es um das Lösen von nichtlinearen Gleichungen geht, stellt sich natürlich außer der Berechnung von Näherungswerten der Lösung eine Reihe von Fragen: wieviele reelle Wurzeln besitzt eine gegebene Gleichung? Wieviele imaginäre, positive oder negative?

Im Folgenden beginnen wir damit, Teilantworten zu Sonderfällen algebraischer Gleichungen zu geben. Danach beschreiben wir einige Approximationsverfahren, mit denen Näherungswerte von Lösungen einer nichtlinearen Gleichung berechnet werden können.

12.2.1. Lokalisierung der Lösungen algebraischer Gleichungen

Vorzeichenregel von Descartes. Die Regel von Descartes besagt, dass die Anzahl der positiven Wurzeln eines Polynoms mit reellen Koeffizienten kleiner ist oder gleich der Anzahl der Vorzeichenwechsel in der Folge der Koeffizienten des Polynoms.

```
sage: R.<x> = PolynomialRing(RR, 'x')
sage: p = x^7 - 131/3*x^6 + 1070/3*x^5 - 2927/3*x^4 \
....:  + 2435/3*x^3 - 806/3*x^2 + 3188/3*x - 680
sage: sign_changes = \
....: [p[i] * p[i + 1] < 0 for i in range(p.degree())].count(True)
sage: real_positive_roots = \
....: sum([alpha[1] if alpha[0] > 0 else 0 for alpha in p.roots()])
sage: sign_changes, real_positive_roots
(7, 5)
```

Seien nun p ein Polynom mit reellen Koeffizienten vom Grad d und p' das abgeleitete Polynom. Wir bezeichnen mit u und u' die Folgen der Vorzeichen der Koeffizienten der Polynome p und

p': wir haben $u_i = \pm 1$, je nachdem, ob der Koeffizient des Grades i positiv oder negativ ist. Die Folge u' ergibt sich aus u durch einfaches Abschneiden: wir bekommen $u'_i = u_{i+1}$ für $0 \leq i < d$. Daraus folgt, dass die Anzahl der Vorzeichenwechsel der Folge u höchsten gleich der Anzahl der Vorzeichenwechsel der Folge $'$ plus 1 ist.

Außerdem ist die Anzahl der positiven Wurzeln von p höchstens gleich der Anzahl positiver Wurzeln von p' plus 1: ein Intervall, dessen Grenzen Wurzeln von p sind, enthält immer eine Wurzel von p'.

Da die Regel von Descartes auch für ein Polynom 1. Grades richtig ist, zeigen die beiden vorstehenden Beobachtungen, dass sie auch für ein Polynom 2. Grades stimmt usw.

Es ist möglich, die Beziehung zwischen den Anzahlen positiver Wurzeln und der Anzahl der Vorzeichenwechsel der Folge der Koeffizienten zu präzisieren: die Differenz dieser beiden Zahlen ist immer gerade.

Isolation der reellen Wurzeln der Polynome. Wir haben gesehen, dass man bei Polynomen mit reellen Koeffizienten eine Obergrenze der im Intervall $[0, \infty[$ enthaltenen Wurzeln bestimmen kann. Noch allgemeiner, es gibt Mittel, die Anzahl der Wurzeln in einem gegebenen Intervall zu präzisieren.

Nehmen wir beispielsweise den Satz von Sturm. Seien p ein Polynom mit reellen Koeffizienten, d dessen Grad und $[a, b]$ ein Intervall. Rekursiv bilden wir eine Folge von Polynomen. Am Anfang seien $p_0 = p$ und $p_1 = p'$. Dann ist p_{i+2} das Negative des Restes der euklidschen Division von p_i durch p_{i+1}. Wenn wir diese Folge von Polynomen an den Stellen a und b auswerten, erhalten wir zwei endliche reelle Folgen $(p_0(a), \ldots, p_d(a))$ und $(p_0(b), \ldots, p_d(b))$. Der Satz von Sturm besagt: die Anzahl der Wurzeln von p im Intervall $[a, b]$ ist gleich der Anzahl der Vorzeichenwechsel der Folge $(p_0(a), \ldots, p_d(a))$ vermindert um die Anzahl der Vorzeichenwechsel der Folge $(p_0(b), \ldots, p_d(b))$ unter der Annahme, dass die Wurzeln von p einfach sind sowie $p(y) \neq 0$ und $p(b) \neq 0$.

Wir zeigen, wie dieser Satz mit Sage implementiert wird.

```
sage: def count_sign_changes(p):
....:     l = [c for c in p if not c.is_zero()]
....:     changes = [l[i]*l[i + 1] < 0 for i in range(len(l) - 1)]
....:     return changes.count(True)

sage: def sturm(p, a, b):
....:     assert p.degree() > 2
....:     assert not (p(a) == 0)
....:     assert not (p(b) == 0)
....:     assert a <= b
....:     remains = [p, p.derivative()]
....:     for i in range(p.degree() - 1):
....:         remains.append(-(remains[i] % remains[i + 1]))
....:     evals = [[], []]
....:     for q in remains:
....:         evals[0].append(q(a))
....:         evals[1].append(q(b))
....:     return count_sign_changes(evals[0]) - count_sign_changes(evals[1])
```

Hier nun eine Illustration dieser Funktion `sturm()`.

```
sage: R.<x> = PolynomialRing(QQ, 'x')
sage: p = (x - 34) * (x - 5) * (x - 3) * (x - 2) * (x - 2/3)
sage: sturm(p, 1, 4)
2
sage: sturm(p, 1, 10)
3
sage: sturm(p, 1, 200)
4
sage: p.roots(multiplicities=False)
[34, 5, 3, 2, 2/3]
sage: sturm(p, 1/2, 35)
5
```

12.2.2. Verfahren der sukzessiven Approximation

> *Der Begriff der Approximation ist von fundamentaler Bedeutung für das Verständnis der Beziehung zwischen Realität und wissenschaftlicher Modellbildung. Schon der Versuch, Aspekte der Wirklichkeit in einer Theorie zu erklären, bedeutet die Vernachlässigung gewisser für die spezielle Fragestellung nicht so wesentlicher Einzelheiten. Nur die Wirklichkeit selbst beschreibt die Wirklichkeit vollständig, jedes Modell leistet dies nur approximativ.*
>
> Walter Trockel, Ein mathematischer Countdown zur Wirtschaftswissenschaft[1]

In diesem Unterabschnitt veranschaulichen wir verschiedene Verfahren der Approximation an die Lösungen einer nichtlinearen Gleichung $f(x) = 0$. Es gibt im wesentlichen zwei Vorgehensweisen zur Berechnung solcher Näherungen. Der leistungsfähigste Algorithmus kombiniert sie beide.

Bei der ersten Vorgehensweise bilden wir eine Folge von geschachtelten Intervallen, die eine Lösung der Gleichung enthalten. Wir steuern die Genauigkeit, die Kovergenz ist gesichert, doch ist die Konvergenzgeschwindigkeit nicht immer gut.

Die zweite Vorgehensweise nimmt einen Näherungswert einer Lösung als bekannt an. Wenn das lokale Verhalten der Funktion f hinreichend gleichförmig ist, können wir einen neuen Näherungswert berechnen, der näher an der Lösung liegt. Durch Rekursion bekommen wir deshalb eine Folge von Näherungswerten. Diese Vorhensweise setzt also eine erste Approximation an die gesuchte Zahl als bekannt voraus. Außerdem hängt ihre Effizienz vom lokalen Verhalten der Funktion f ab. Von vornherein beherrschen wir die Genauigkeit der Näherungswerte nicht; aber schlimmer noch, die Konvergenz der Folge der Näherungswerte ist nicht garantiert.

Im Folgenden betrachten wir eine nichtlineare Gleichung $f(x) = 0$, wobei f eine numerische Funktion bezeichnet, die auf einem Interval $[a, b]$ definiert und stetig ist. Wir setzen voraus, dass die Funktion an den Grenzen des Intervalls nicht verschwindet, sondern verschiedene

[1] Im Original heißt es:
Approximation: Gén. au sing. Opération par laquelle on tend à se rapprocher de plus en plus de la valeur réelle d'une quantité ou d'une grandeur sans y parvenir rigoureusement.
Trésor de la Langue Française

Vorzeichen hat: anders gesagt, das Produkt $f(a)f(b)$ ist negativ. Die Stetigkeit von f sichert dann die Existenz von mindestens einer Lösung der Gleichung $f(x) = 0$.

Bei jedem Verfahren experimentieren wir mit folgender Funktion:

```
sage: f(x) = 4 * sin(x) - exp(x) / 2 + 1
sage: a, b = RR(-pi), RR(pi)
sage: bool(f(a) * f(b) < 0)
True
```

Der Hinweis ist angebracht, dass bei diesem Beispiel der Befehl `solve` nicht von Nutzen ist.

```
sage: solve(f(x) == 0, x)
[sin(x) == 1/8*e^x - 1/4]
sage: f.roots()
Traceback (most recent call last):
 ...
RuntimeError: no explicit roots found
```

Die Algorithmen zur Suche von Lösungen nichtlinearer Gleichungen können aufwendig sein: zweckmäßigerweise trifft man vor der Programmausführung einige Vorsichtsmaßnahmen. Wir versichern uns der Existenz von Lösungen, indem wir Stetigkeit und Differenzierbarkeit der Funktion prüfen, die gleich null gesetzt werden soll, sowie auch eventueller Vorzeichenwechsel. Dafür kann die Zeichnung des Graphen eine Hilfe sein (siehe Kapitel 4).

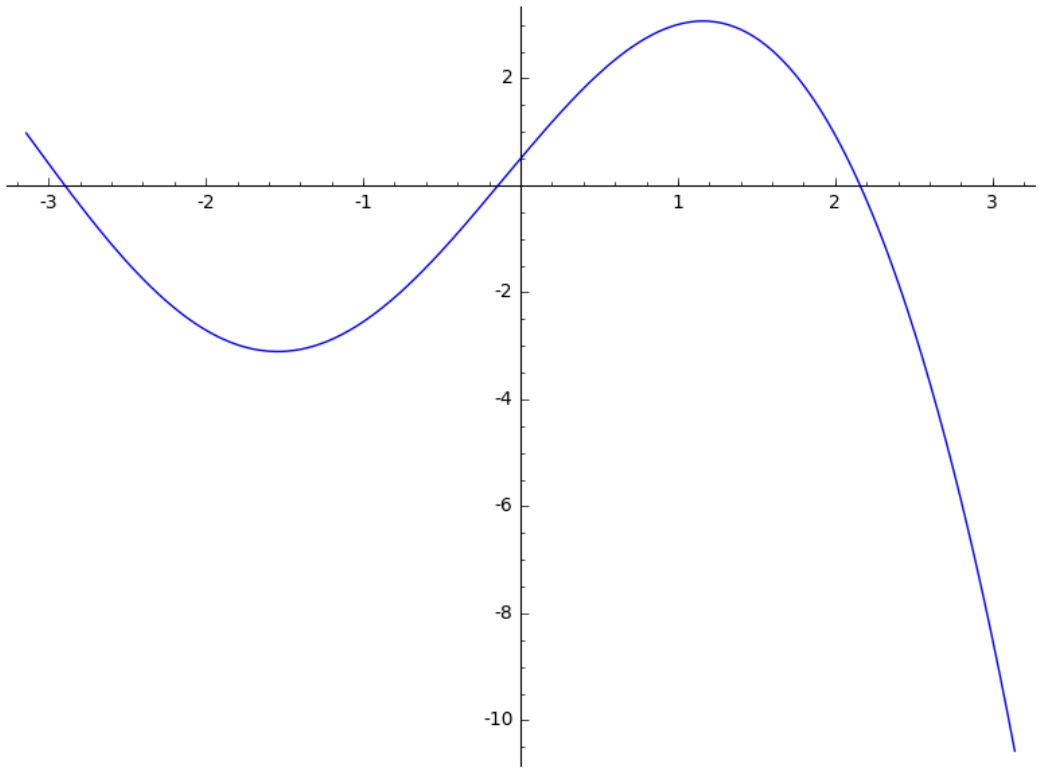


Abb. 12.2 - Graph der Funktion f.

Die Bisektion. Dieses Verfahren beruht auf der ersten Vorgehensweise: dabei ist eine Folge von geschachtelten Intervallen zu bilden, von denen jedes eine Lösung der Gleichung $f(x) = 0$ enthält.

Wir teilen das Intervall $[a, b]$ an seiner Mitte, die wir c nennen. Wir nehmen $f(c) \neq 0$ an (andernfalls haben wir schon eine Lösung gefunden). Ist $f(a)f(c)$ kleiner als null, enthält das Intervall $[a, c]$ notwendigerweise eine Lösung der Gleichung; ist $f(c)f(b)$ kleiner als null, dann enthält das Intervall $[b, c]$ eine Lösung der Gleichung. So können wir ein Intervall finden, das eine Lösung enthält und dessen Länge nur halb so groß ist wie das Intervall $[a, b]$. Durch Wiederholung dieses Schrittes erhalten wir eine Folge von Intervallen mit den erwarteten Eigenschaften. Um dieses Vorgehensweise ins Werk zu setzen, definieren wir die Python-Funktion `intervalgen`:

```
sage: def phi(s, t): return (s + t) / 2
sage: def intervalgen(f, phi, s, t):
....:     msg = 'Wrong arguments: f({0})*f({1})>=0)'.format(s, t)
....:     assert (f(s) * f(t) < 0), msg
....:     yield s
....:     yield t
....:     while True:
....:         u = phi(s, t)
....:         yield u
....:         if f(u) * f(s) < 0:
....:             t = u
....:         else:
....:             s = u
```

Die Definition dieser Funktion verdient einige Erläuterungen. Die Anwesenheit des Schlüsselwortes `yield` in der Definition von `intervalgen` macht daraus einen *Generator* (siehe Unterabschnitt 15.2.4). Bei einem Aufruf der Methode `next` eines Generators werden alle lokalen Daten gesichert, wenn der Interpreter auf das Schlüsselwort `yield` trifft, die Ausführung wird unterbrochen, und der unmittelbar rechts vom Schlüsselwort stehende Ausdruck wird zurückgegeben. Der auf die Methode `next` folgende Aufruf fährt mit der Anweisung fort, die dem Schlüsselwort `yield` folgt und den lokalen Daten, die vor der Unterbrechung gesichert worden sind. In einer Endlosschleife (`while True:`) erlaubt das Schlüsselwort `yield` daher die Prgrammierung einer rekursiv definierten Folge mit eine Syntax, die ihrer mathematischen Beschreibung nahe ist. Die Ausführung kann mit dem üblichen Schlüsselwort `return` beendet werden.

Der Parameter `phi` stellt eine Funktion dar. Für das Verfahren der Bisektion berechnet diese Funktion die Mitte eines Intervalls. Um ein anderes Verfahren der sukzessiven Approximation zu testen, das ebenfalls auf der Bildung geschachtelter Intervalle basiert, geben wir eine neue Definition der Funktion `phi` an und bilden mit der Funktion `intervalgen` wieder den entsprechenden Generator.

Die Parameter `s` und `t` der Funktion stehen für die Grenzen des ersten Intervalls. Ein Aufruf von `assert` prüft, ob die Funktion f zwischen den Grenzen des Intervalls das Vorzeichen wechselt; wir wissen, dass gegebenenfalls eine Lösung existiert.

Die beiden ersten Werte des Generators entsprechen den Parametern `s` und `t`. Der dritte Wert ist die Mitte des jeweiligen Intervalls. Danach repräsentieren die Parameter `s` und `t` die Grenzen des zuletzt berechneten Intervalls. Nach Auswertung von f in der Mitte dieses

Intervalls ändern wir eine Grenze des Intervalls so ab, dass das neue Intervall wieder eine Lösung enthält. Als Näherungswert der gesuchten Lösung bietet sich die Mitte des zuletzt berechneten Intervalls an.

Experimentieren wir mit unserem Beispiel: es folgen drei Approximationen mit dem Verfahren der Bisektion, das auf das Intervall $[-\pi, \pi]$ angewendet wird.

```
sage: a, b
(-3.14159265358979, 3.14159265358979)
sage: bisection = intervalgen(f, phi, a, b)
sage: bisection.next()
-3.14159265358979
sage: bisection.next()
3.14159265358979
sage: bisection.next()
0.000000000000000
```

Um die verschiedenen Näherungsverfahren zu vergleichen, ist es sinnvoll, über einen Mechanismus zu verfügen, der die Berechnung des Näherungswertes einer Lösung der Gleichung $f(x) = 0$ mit Hilfe von mit Sage für jedes dieser Verfahren definierten Generatoren automatisiert. Dieser Mechanismus muss die Steuerung der Rechengenauigkeit und der Maximalzahl der Iterationen ermöglichen. Das ist die Aufgabe der Funktion `iterate`, deren Definition hier folgt.

```
sage: from types import GeneratorType, FunctionType
sage: def checklength(u, v, w, prec):
....:     return abs(v - u) < 2 * prec
sage: def iterate(series, check=checklength, prec=10^-5, maxit=100):
....:     assert isinstance(series, GeneratorType)
....:     assert isinstance(check, FunctionType)
....:     niter = 2
....:     v, w = series.next(), series.next()
....:     while niter <= maxit:
....:         niter += 1
....:         u, v, w = v, w, series.next()
....:         if check(u, v, w, prec):
....:             print 'After {0} iterations: {1}'.format(niter, w)
....:             return
....:     print 'Failed after {0} iterations'.format(maxit)
```

Der Parameter `series` muss ein Generator sein. Wir speichern die letzten drei Werte dieses Generators, damit wir einen Konvergenztest machen können. Das ist die Aufgabe des Parameters `check`: eine Funktion, die die Iterationen notfalls beendet. In der Voreinstellung benutzt die Funktion `iterate` die Funktion `checklength`, die die Iterationen abbricht, wenn das letzte berechnete Intervall kleiner ist als das Doppelte des Parameters `prec`; das garantiert, dass der mit dem Verfahren der Bisektion berechnete Wert ein Näherungswert ist mit einem Fehler, der kleiner ist als `prec`.

Sobald die Anzahl der Iterationen den Parameter `maxit` übersteigt, wird eine Exzeption ausgelöst.

```
sage: bisection = intervalgen(f, phi, a, b)
sage: iterate(bisection)
After 22 iterations: 2.15847275559132
```

Übung 42. Die Funktion `intervalgen` ist so zu modifizieren, dass der Generator anhält, wenn eine der Intervallgrenzen verschwindet.

Übung 43. Programmieren Sie mit den Funktionen `intervalgen` und `iterate` die Berechnung eines Näherungswertes einer Lösung der Gleichung $f(x) = 0$ mit verschachtelten Intervallen, wobei jedes Intervall durch eine zufällige Teilung des vorhergehenden Intervalls erhalten wird.

Verfahren der falschen Position. Auch dieses Verfahren basiert auf der ersten Vorgehensweise: zu bilden ist eine Folge von verschachtelten Intervallen, von denen jedes eine Lösung der Gleichung $f(x) = 0$ enthält. Um die Intervalle zu teilen, verwenden wir diesmal jedoch eine lineare Interpolation der Funktion f.

Genauer gesagt betrachten wir den Abschnitt des Graphen von f, der von den beiden Punkten von f mit den Abszissen a und b begrenzt wird. Da $f(a)$ und $f(b)$ entgegengesetzte Vorzeichen haben, schneidet dieser Abschnitt die Abszissenachse. Er teilt daher das Intervall $[a, b]$ in zwei Intervallle. Wie beim Verfahren der Bisektion identifizieren wir das eine Lösung enthaltende Intervall, indem wir den Wert berechnen, den f an der Stelle annimmt, die beide Intervalle gemeinsam haben.

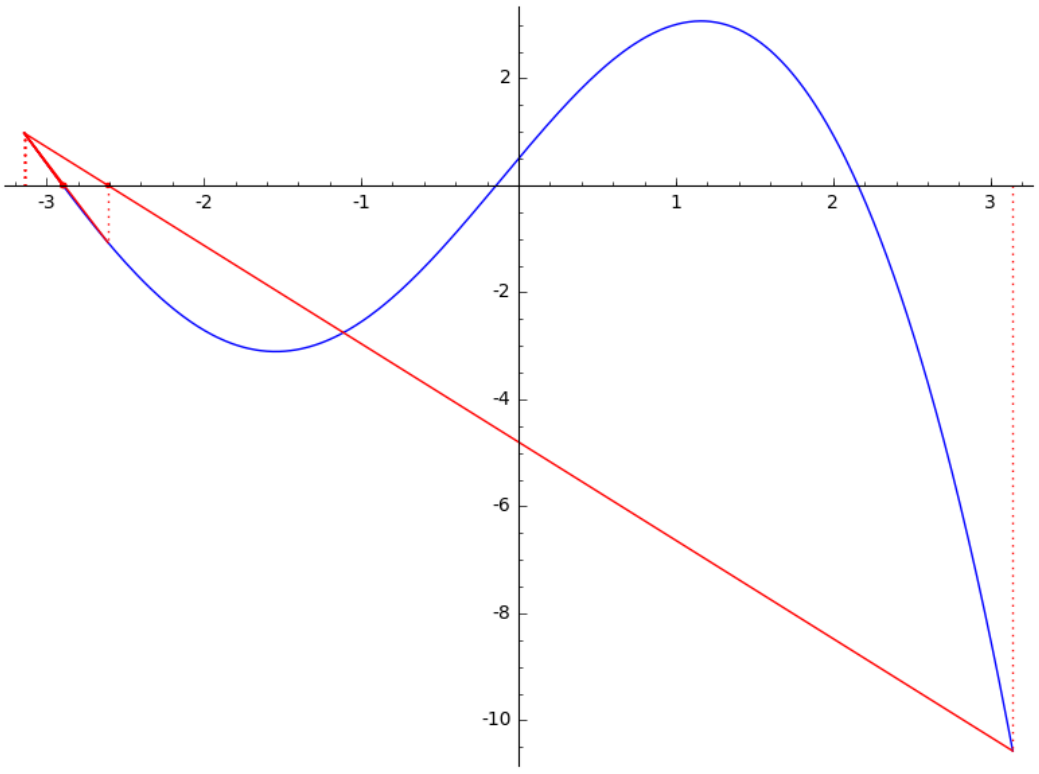


Abb. 12.3 - Verfahren der falschen Position auf $[-\pi, \pi]$.

Die Gerade durch die Punkte $(a, f(a))$ und $(b, f(b))$ hat die Gleichung:

$$y = \frac{f(b) - f(a)}{b - a}(x - a) + f(a). \tag{12.1}$$

Wegen $f(b) \neq f(a)$ schneidet diese Gerade die Abszisse an der Stelle

$$a - f(a)\frac{b - a}{f(b) - f(a)}.$$

Wir können dieses Verfahren nunmehr folgendermaßen testen:

```
sage: phi(s, t) = s - f(s) * (t - s) / (f(t) - f(s))
sage: falsepos = intervalgen(f, phi, a, b)
sage: iterate(falsepos)
After 8 iterations: -2.89603757331027
```

Wichtig ist, dass die mit den Verfaren der Bisektion wie auch der falschen Position gebildeten Folgen nicht notwendig gegen die gleiche Lösung konvergieren. Durch Verkleinerung des untersuchten Intervalls findet man auch die positive Lösung, die mit dem Verfahren der Bisektion erhalten wurde.

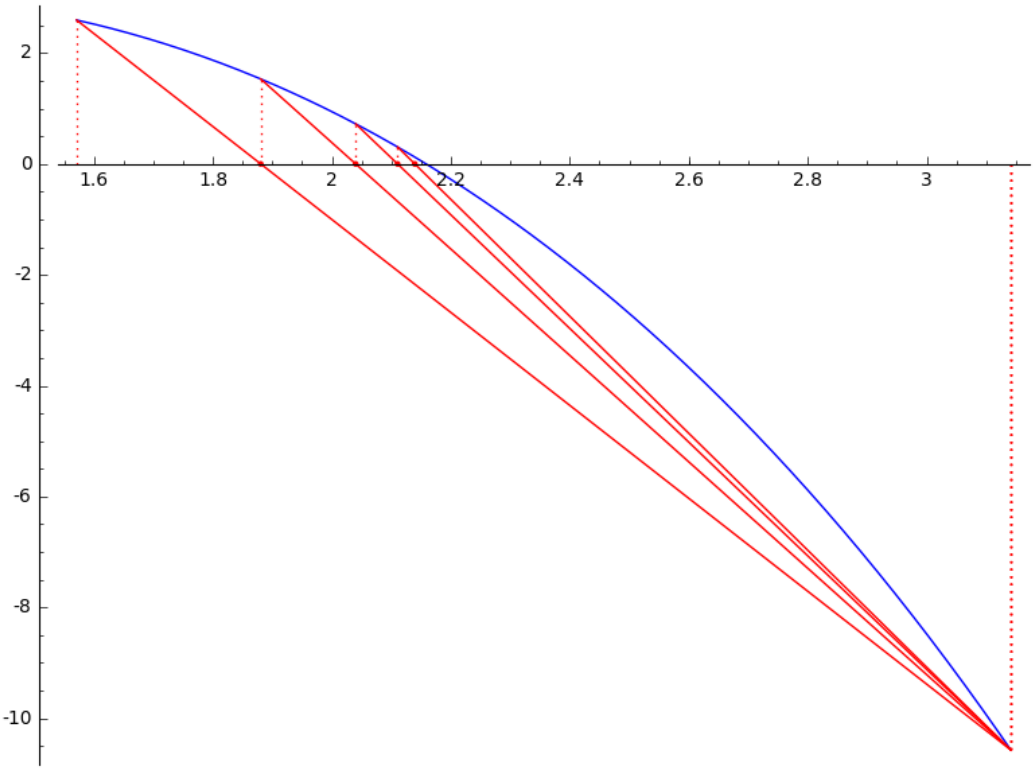


Abb. 12.4 - Verfahren der falschen Position auf $[-\pi/2, \pi]$.

```
sage: a, b = RR(pi/2), RR(pi)
sage: phi(s, t) = t - f(t) * (s - t) / (f(s) - f(t))
sage: falsepos = intervalgen(f, phi, a, b)
sage: phi(s, t) = (s + t) / 2
sage: bisection = intervalgen(f, phi, a, b)
```

```
sage: iterate(falsepos)
After 15 iterations: 2.15846441170219
sage: iterate(bisection)
After 20 iterations: 2.15847275559132
```

Das Newton-Verfahren. Wie das Verfahren der falschen Position verwendet auch das Newton-Verfahren eine lineare Approximation der Funktion f. Graphisch gesehen handelt es sich darum, eine Tangente an den Graphen von f als Näherung dieser Kurve zu betrachten.

Wir setzen jetzt voraus, dass f differenzierbar ist und dass die Ableitung f' im Intervall $[a, b]$ dasselbe Vorzeichen hat. Somit ist f hier monoton. Wir setzen auch voraus, dass f das Vorzeichen im Intervall $[a, b]$ wechselt. Die Gleichung $f(x) = 0$ hat somit in diesem Intervall eine eindeutige Lösung; diese Zahl bezeichnen wir mit α.

Sei $u_0 \in [a, b]$. Die Tangente an den Graphen von f an der Stelle u_0 hat die Gleichung

$$y = f'(u_0)(x - u_0) + f(u_0). \tag{12.2}$$

Die Koordinaten des Schnittpunktes dieser Gerade mit der Abszisse sind

$$(u_0 - f(u_0)/f'(u_0), 0).$$

Mit φ bezeichnen wir die Funktion $x \mapsto x - f(x)/f'(x)$. Sie ist unter der Bedingung definiert, dass f' im Intervall $[a, b]$ nicht verschwindet. Wir interessieren uns für die rekursiv durch $u_{n+1} = \varphi(u_n)$ definierte Folge u. Wenn die Folge u konvergiert,[2] dann erfüllt ihr Grenzwert l die Bezehung $l = l - f(l)/f'(l)$, woraus sich $f(l) = 0$ ergibt; der Grenzwert ist gleich α, und das ist die Lösung der Gleichung $f(x) = 0$. Damit das Beispiel die Bedingung der Monotonie erfüllt, reduzieren wir das untersuchte Intervall.

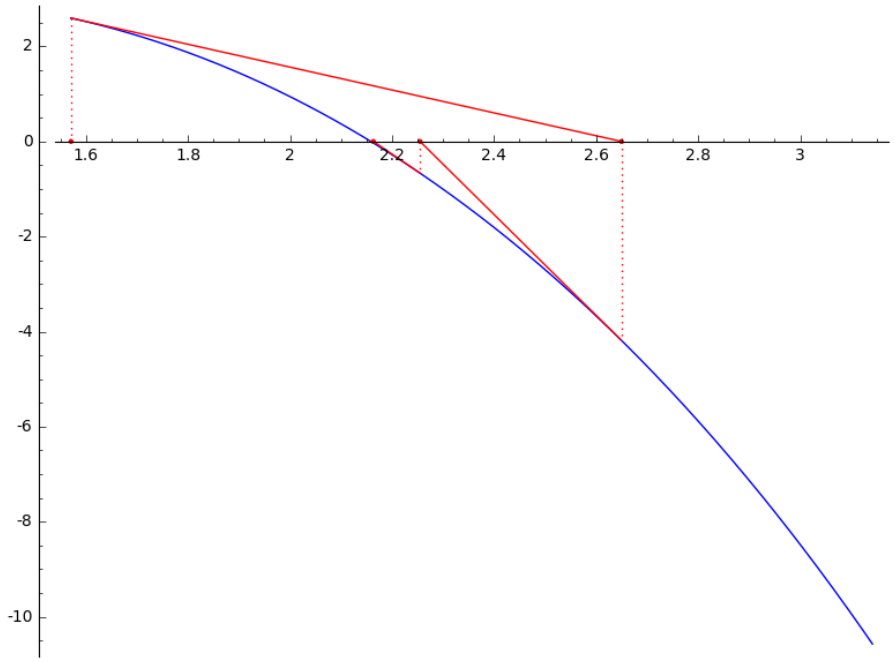


Abb. 12.5 - Newton-Verfahren.

[2] Ein Satz von Leonid Witaljewitsch Kantorowitsch liefert eine hinreichende Bedingung für die Konvergenz des Newton-Verfahrens.

```
sage: f.derivative()
x |--> 4*cos(x) - 1/2*e^x
sage: a, b = RR(pi/2), RR(pi)
```

Wir definieren einen Python-Generator `newtongen`, der eine rekursiv zu definierende Folge darstellt. Dann formulieren wir einen neuen Konvergenztest `checkconv`, der terminiert, wenn die beiden zuletzt berechneten Wert nahe genug beieinander liegen, doch ist zu beachten, dass dieser Test die Konvergenz keineswegs garantiert.

```
sage: def newtongen(f, u):
....:     while True:
....:         yield u
....:         u -= f(u) / f.derivative()(u)
sage: def checkconv(u, v, w, prec):
....:     return abs(w - v) / abs(w) <= prec
```

Wir können nun das Newton-Verfahren mit unserem Beispiel testen.

```
sage: iterate(newtongen(f, a), check=checkconv)
After 6 iterations: 2.15846852566756
```

Sekantenverfahren Beim Newton-Verfahren kann die Berechnung der Ableitung sehr aufwendig sein. Es ist bei dieser Rechnung möglich, die Ableitung durch eine lineare Interpolation zu ersetzen: wenn wir über zwei Näherungswerte für die Lösung, und damit über zwei Punkte des Graphen von f verfügen, und wenn die Gerade durch diese beiden Punkte die Abszissenachse trifft, dann betrachten wir die Abszisse des Schnittpunktes als neue Näherung. Zu Beginn und wenn beide Geraden parallel sind, berechnen wir die nächste Approximation mit dem Newton-Verfahren.

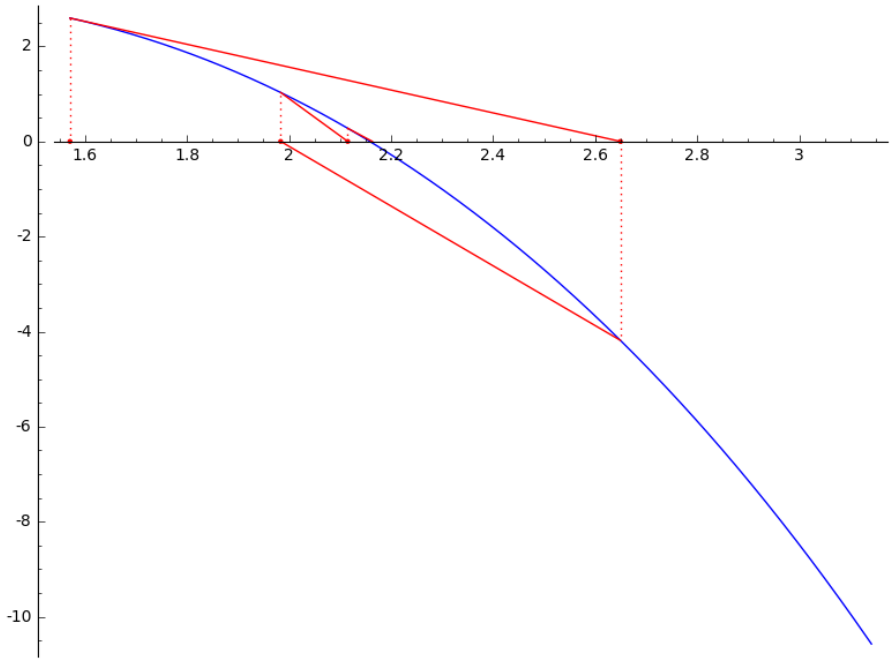


Abb. 12.6 - Sekantenverfahren.

Das führt auf dieselbe Iterationsmethode wie das Verfahren der falschen Position, aber mit anderen Punkten. Anders als das Verfahren der falschen Position bestimmt das Sekantenverfahren kein Intervall mit einer Wurzel.

Wir definieren einen Python-Generator, der das Verfahren in Gang setzt:

```
sage: def secantgen(f, a):
....:     yield a
....:     estimate = f.derivative()(a)
....:     b = a - f(a) / estimate
....:     yield b
....:     while True:
....:         fa, fb = f(a), f(b)
....:         if fa == fb:
....:             estimate = f.derivative()(a)
....:         else:
....:             estimate = (fb - fa) / (b - a)
....:         a = b
....:         b -= fb / estimate
....:         yield b
```

Nun können wir das Sekantenverfahren an unserem Beispiel ausprobieren.

```
sage: iterate(secantgen(f, a), check=checkconv)
After 8 iterations: 2.15846852557553
```

Das Muller-Verfahren. Wir können das Sekantenverfahren erweitern, indem wir f durch ein Polynom beliebigen Grades ersetzen. Beispielsweise arbeitet das Muller-Verfahren[3] mit quadratischen Approximationen.

Angenommen, wir haben schon drei Näherungslösungen r, s und t der Gleichung $f(x) = 0$ berechnet. Wir betrachten das Interpolationspolynom von Lagrange, das durch die drei Kurvenpunkte mit den Abszissen r, s und t bestimmt ist. Das ist ein Polynom zweiten Grades. Sinnvollerweise nehmen wir als neue Näherung die Wurzel des Polynoms, die am nächsten bei t liegt. Vorher werden die drei Anfangsterme der Folge auf beliebige Weise fixiert, a, b und dann $(a + b)/2$. Es muss gesagt werden, dass die Wurzeln des Polynoms - und damit die berechneten Näherungswerte - komplexe Zahlen sein können.

Die Programmierung dieses Verfahrens in Sage ist nicht schwierig; sie erfolgt nach dem gleichen Muster wie das Sekantenverfahren. Unsere Realisierung verwendet allerdings eine Datenstruktur, die an die Auflistung der Terme einer rekursiv definierten Folge besser angepasst ist.

```
sage: from collections import deque
sage: basering = PolynomialRing(CC, 'x')
sage: def quadraticgen(f, r, s):
....:     t = (r + s) / 2
....:     yield t
....:     points = deque([(r,f(r)), (s,f(s)), (t,f(t))], maxlen=3)
```

[3] Es handelt sich um David E. Muller, bekannt auch als Erfinder des Reed-Muller-Codes, nicht um den in Kapitel 11 erwähnten Jean-Michel Muller

```
....:     while True:
....:         pol = basering.lagrange_polynomial(points)
....:         roots = pol.roots(ring=CC, multiplicities=False)
....:         u = min(roots, key=lambda x: abs(x - points[2][0]))
....:         points.append((u, f(u)))
....:         yield points[2][0]
```

Das Modul `collections` aus Pythons Referenzbibliothek implementiert mehrere Datenstrukturen. In `quadraticgen` wird die Klasse `deque` benutzt, um die zuletzt berechneten Approximationen zu speichern. Ein Objekt `deque` speichert Daten bis zur Grenze `maxlen`, die bei seiner Erzeugung festgelegt wurde; hier ist die maximale Datenanzahl gleich der Rekursionstiefe der Näherungen. Wenn ein Objekt `deque` seine maximale Speicherkapazität erreicht hat, fügt die Methode `deque.append()` die neuen Daten nach dem Prinzip FIFO (first in, first out) hinzu.

Zu erwähnen ist, dass die Iterationen dieses Verfahrens keine Berechnung der Ableitungswerte erfordern. Jede Iteration verlangt nur eine Auswertung der Funktion f.

```
sage: generator = quadraticgen(f, a, b)
sage: iterate(generator, check=checkconv)
After 5 iterations: 2.15846852554764
```

Zurück zu den Polynomen. Kehren wir zu der Situation zurück, die wir zu Beginn dieses Kapitels untersucht haben. Es geht um die Berechnung der Wurzeln eines Polynoms mit reellen Koeffizienten; wir nennen diese Polynom P. Wir wollen P als unitär voraussetzen:

$$P = a_0 + a_1 x + \ldots + a_{d-1} x^{d-1} + x^d.$$

Es ist einfach zu verifizieren, dass P das charakteristische Polynom der Begleitmatrix ist (siehe Unterabschnitt 8.2.3):

$$A = \begin{pmatrix} 0 & 0 & 0 & \ldots & 0 & -a_0 \\ 1 & 0 & 0 & \ldots & 0 & -a_1 \\ 0 & 1 & 0 & \ldots & 0 & -a_2 \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & 0 & 0 & \ldots & 1 & -a_{d-1} \end{pmatrix}.$$

Folglich sind die Wurzeln des Polynoms P die Eigenwerte der Matrix A. Deshalb werden hier die Verfahren aus Kapitel 13 angewendet.

Wir haben gesehen, dass die Methode `Polynomial.roots()` bis zu drei Parameter aufnimmt, die alle optional sind: `ring, multiplicities` und `algorithm`. Wir wollen annehmen, dass ein Sage-Objekt der Klasse `Polynomial` dem Namen `p` zugeordnet ist (deshalb gibt `isinstance(p, 'Polynomial')` den Wert `True` zurück). Welchen Algorithmus der Befehl `p.roots()` benutzt, hängt dann von den Parametern `ring` und `algorithm` sowie vom Koeffizientenring des Polynoms ab, d.h. von `p.base_ring()`.

Der Algorithmus prüft, ob die in `ring` und `p.base_ring()` ausgeführten arithmetischen Operationen exakt sind. Sollte dies nicht der Fall sein, werden die Näherungswerte der Wurzeln mit der Bibliothek NumPy berechnet, falls `RDF` der `p.base_ring()` ist, und wenn `CDF`, dann mit der Bibliothek PARI (der Parameter `algorithm` ermöglicht dem Anwender eine abweichende Wahl der Bibliothek für die Berechnung). Im Quellcode von NumPy sieht man, dass das von

dieser Bibliothek für die Wurzeln benutzte Approximationsverfahren aus der Berechnung der Eigenwerte der Begleitmatrix besteht.

Der folgende Befehl gestattet die Identifikation derjenigen Objekte, bei denen die arithmetischen Operationen exakt sind (der von der Methode `Ring.is_exact()` zurückgegebene Wert ist in diesem Fall `True`).

```
sage: for ring in [ZZ, QQ, QQbar, RDF, RIF, RR, AA, CDF, CIF, CC]:
sage: print("{0:50} {1}".format(ring, ring.is_exact()))
Integer Ring                                       True
Rational Field                                     True
Algebraic Field                                    True
Real Double Field                                  False
Real Interval Field with 53 bits of precision      False
Real Field with 53 bits of precision               False
Algebraic Real Field                               True
Complex Double Field                               False
Complex Interval Field with 53 bits of precision   False
Complex Field with 53 bits of precision            False
```

Wenn der Parameter `ring` gleich `AA` oder `RIF` ist, während `p.base_ring()` gleich `ZZ`, `QQ` oder `AA` ist, ruft der Algorithmus die Funktion `real_roots()` aus dem Modul `sage.rings.polynomial.real_roots` auf. Diese Funktion konvertiert das Polynom zur Bernstein-Basis, wendet dann den Algorithmus von Casteljau an (um das in Bernstein-Basis ausgedrückte Polynom auszuwerten) und die Regel von Descartes (siehe Unterabschnitt 12.2.1), um die Wurzeln zu lokalisieren.

Ist der Parameter `ring` gleich `QQbar` oder `CIF` und `p.base_ring()` gleich `ZZ, QQ, AA` oder stellt er Gaußsche rationale Zahlen dar, delegiert der Algorithmus die Berechnungen an NumPy und PARI, deren Ergebnisse in die erwarteten Ringe konvertiert werden. In der Dokumentation zur Methode `Polynomial.roots()` kann man alle Situationen kennen lernen, die von dieser Methode abgedeckt werden.

Konvergenzgeschwindigkeit. Betrachten wir eine konvergente numerische Folge u und bezeichnen wir mit l ihren Grenzwert. Wir sagen, dass die Konvergenzgeschwindigkeit *linear* ist, wenn ein K existiert, sodass

$$\lim_{n\to\infty} \frac{|u_{n+1} - l|}{|u_n - l|} = K.$$

Die Konvergenzgeschwindigkeit der Folge u wird *quadratisch* genannt, wenn ein $K > 0$ existiert, sodass

$$\lim_{n\to\infty} \frac{|u_{n+1} - l|}{|u_n - l|^2} = K.$$

Wenden wir uns wieder dem Newton-Verfahren zu. Wir haben eine Folge u durch $u_{n+1} = \varphi(u)$ rekursiv definiert mit φ als der Funktion $x \mapsto x - f(x)/f'(x)$. Unter der Annahme, dass f zweimal differenzierbar ist, kann die Taylorgleichung für die Funktion φ und mit x in der Nähe der Wurzel α so geschrieben werden:

$$\varphi(x) = \varphi(\alpha) + (x - \alpha)\varphi'(\alpha) + \frac{(x - \alpha)^2}{2}\varphi''(\alpha) + \mathcal{O}_\alpha((x - \alpha)^3).$$

Nunmehr ist $\varphi(\alpha) = \alpha$, $\varphi'(\alpha) = 0$ und $\varphi''(\alpha) = f''(\alpha)/f'(\alpha)$. Durch Einsetzen in die vorstehende Gleichung und Rückkehr zur Definition der Folge u erhalten wir:

$$u_{n+1} - \alpha = \frac{(u_n - \alpha)^2}{2} \frac{f''(\alpha)}{f'(\alpha)} + \mathcal{O}_\infty((u_n - \alpha)^3).$$

Da das Newton-Verfahren konvergiert, ist die Geschwindigkeit der gebildeten Folge quadratisch.

Konvergenzbeschleunigung. Aus einer konvergenten Folge, deren Geschwindigkeit linear ist, können wir eine Folge bilden, deren Konvergenzgeschwindigkeit quadratisch ist. Diese Technik ist bei Anwendung auf das Newton-Verfahren unter der Bezeichnung Steffensen-Verfahren bekannt.

```
sage: def steffensen(sequence):
....:     assert isinstance(sequence, GeneratorType)
....:     values = deque(maxlen=3)
....:     for i in range(3):
....:         values.append(sequence.next())
....:         yield values[i]
....:     while True:
....:         values.append(sequence.next())
....:         u, v, w = values
....:         yield u - (v - u)^2 / (w - 2 * v + u)

sage: g(x) = sin(x^2 - 2) * (x^2 - 2)
sage: sequence = newtongen(g, RR(0.7))
sage: accelseq = steffensen(newtongen(g, RR(0.7)))
sage: iterate(sequence, check=checkconv)
After 17 iterations: 1.41422192763287
sage: iterate(accelseq, check=checkconv)
After 10 iterations: 1.41421041980166
```

Man sieht, dass die Konvergenzgeschwindigkeit asymptotisch verläuft: sie sagt über den Fehler $|u_n - l|$ für gegebenes n nichts aus.

```
sage: sequence = newtongen(f, RR(a))
sage: accelseq = steffensen(newtongen(f, RR(a)))
sage: iterate(sequence, check=checkconv)
After 6 iterations: 2.15846852566756
sage: iterate(accelseq, check=checkconv)
After 7 iterations: 2.15846852554764
```

Lösung von nichtlinearen Gleichungen	
angenäherte Wurzeln eines Polynoms	`Polynomial.roots()`
exakte Wurzeln ohne Garantie alle zu finden	`Expression.roots()`
angenäherte Wurzeln auf den reellen Zahlen	`real_roots`
angenäherte Wurzeln mit dem Brent-Verfahren	`Expression.find_root()`

Tab. 12.1 - In diesem Kapitel beschriebene Befehle

Die Methode `Expression.find_root()`. Wir interessieren uns jetzt für die allgemeinste Situation: die Berechnung eines Näherungswertes der Lösung einer Gleichung $f(x) = 0$. In Sage erfolgt diese Berechnung mit der Methode `Expression.find_root()`.

Die Parameter der Methode `Expression.find_root()` gestatten die Angabe eines Intervall, wo eine Wurzel zu suchen ist, der Rechengenauigkeit oder der Anzahl der Iterationen. Der Parameter `full_output` gibt Informationen zur Berechnung, insbesondere zur Anzahl der Iterationen und zur Anzahl der Auswertungen der Funktion.

```
sage: result = (f == 0).find_root(a, b, full_output=True)
sage: result[0], result[1].iterations
(2.1584685255476415, 9)
```

In Wirklichkeit implementiert die Methode `Expression.find_root()` keinen Suchalgorithmus für das Lösen von Gleichungen: die Rechnung wird an das Modul SciPy delegiert. Die von Sage für die Lösung eine Gleichung benutzte Funktionalität von SciPy implementiert das Brent-Verfahren, das drei uns schon bekannte Verfahren kombiniert: die Bisektion, das Sekantenverfahren und die quadratische Interpolation. Die ersten beiden Näherungswerte sind die Grenzen des Suchintervalls für die Lösung der Gleichung. Den folgenden Näherungswert erhält man durch lineare Interpolation, wie das beim Sekantenverfahren gemacht wird. Bei den dann folgenden Iterationen wird die Funktion durch quadratische Interpolation angenähert, und die Abszisse des Schnittpunktes der Interpolationskurve mit der Achse ist der neue Näherungswert, es sei denn, dieser Wert liegt nicht zwischen den beiden vorhergehenden; in diesem Fall würde mit Bisektion fortgefahren.

Die Bibliothek SciPy ermöglicht keine Rechnung mit beliebiger Genauigkeit (außer bei Rechnung mit ganzen Zahlen); außerdem beginnt die Methode `Expression.find_root`, die Grenzen in Maschinenzahlen doppelter Genauigkeit zu konvertieren. Demgegenüber funktionieren alle anderen in diesem Kapitel erläuterten Lösungsverfahren mit beliebiger Genauigkeit oder sogar symbolisch.

```
sage: a, b = pi/2, pi
sage: generator = newtongen(f, a)
sage: generator.next(); generator.next()
1/2*pi
1/2*pi - (e^(1/2*pi) - 10)*e^(-1/2*pi)
```

Übung 44. Für das Brent-Verfahren ist ein Generator zu schreiben, der mit beliebiger Genauigkeit funktioniert.

13. Numerische lineare Algebra

Hier behandeln wir die numerischen Aspekte der linearen Algebra, nachdem die symbolische lineare Algebra in Kapitel 8 präsentiert worden ist. An französischen Lehrbüchern zur numerischen Analyse der linearen Algebra kann man die Bücher von M. Schatzman [Sch91], P. Ciarlet [Cia82] oder die mehr spezialisierten von Lascaux-Théodor [LT93, LT94] heranziehen. Das auf englisch geschriebene Buch von Golub und Van Loan [GVL96] ist als Referenz ein absolutes Muss.

Numerische lineare Algebra spielt beim sogenannten *wissenschaftlichen Rechnen* eine Rolle. Diese Bezeichnung ist nicht ganz passend, um die Probleme zu beschreiben, die bei der mathematischen Untersuchung der numerischen Analyse begegnen: bei der näherungsweisen Lösung von Systemen gewöhnlicher und partieller Differentialgleichungen, bei Optimierungsaufgaben, bei der Signalverarbeitung usw.

Die numerische Lösung der Mehrzahl dieser Aufgaben, auch linearer Probleme, basiert auf Algorithmen, die mit Endlosschleifen arbeiten. Im tiefsten Grunde solcher Schleifen ist sehr oft ein lineares System zu lösen. Häufig findet das Newton-Verfahren zur Lösung nichtlinearer algebraischer Systeme Anwendung: auch dabei sind lineare Systeme zu lösen. Leistungsfähigkeit und Robustheit der Verfahren der numerischen linearen Algebra sind daher ausschlaggebend.

Dieses Kapitel umfasst drei Abschnitte: im ersten bemühen wir uns den Leser für den Einfluss der Ungenauigkeiten von Rechnungen in der linearen Algebra zu sensibilisieren; der zweite Abschnitt (13.2) behandelt, ohne erschöpfend zu sein, klassische Probleme (Lösung von Gleichungssystemen. Berechnung von Eigenwerten, kleinste Quadrate); im dritten Abschnitt (13.3) zeigen wir, wie bestimmte Probleme zu lösen sind, wenn Matrizen als dünn besetzt angesehen werden können. Dieser letzte Teil will ebenso eine Einführung in Methoden geben, die in einem aktiven Forschungsgebiet benutzt werden, wie auch eine Anleitung zu ihrer Anwendung.

13.1. Ungenaues Rechnen in der linearen Algebra

Wir interessieren uns für klassische Aufgaben der linearen Algebra (Lösung von Gleichungssystemen, Berechnung von Eigenwerten und Eigenvektoren usw.), die mit ungenauer Rechnung gelöst werden. Die erste Quelle von Ungenauigkeiten entspringt aus der Benutzung von (reellen oder komplexen) Fließpunktzahlen, also nicht nur aus der Arbeit mit Objekten, von denen man weiß, das sie nicht exakt sind, sondern auch daraus, dass sämtliche Rechnungen fehlerbehaftet sind. Die verschiedenen in Sage angebotenen Typen von Fließpunktzahlen sind in Kapitel 12 beschrieben.

Es ist beispielsweise das Gleichungssystem $Ax = b$ zu lösen, wobei A eine Matrix mit reellen Koeffizienten ist. Welchen Fehler δx macht man, wenn A mit δA und b mit δb gestört ist? In diesem Kapitel tragen wir dazu Teilantworten zusammen.

13.1.1. Normen von Matrizen und Kondition

Sei $A \in \mathbb{R}^{n\times n}$ (oder $\mathbb{C}^{n\times n}$). Wir statten $\mathbb{R}^n$ (oder $\mathbb{C}^n$) mit einer Norm aus: $||x||_\infty = \max|x_i|$ oder $||x||_1 = \sum_{i=1}^n |x_i u|$ oder auch mit der euklidschen Norm $||x||_2 = \left(\sum_{i=1}^n x_i^2\right)^{1/2}$; dann definiert die Größe

$$||A|| = \max_{||x||=1} ||Ax||$$

eine Norm auf der Menge der $n \times n$-Matrizen. Man sagt, dass es sich um eine der auf $\mathbb{R}^n$ (oder $\mathbb{C}^n$) definierten Norm *untergeordnete Norm* handelt. Die *Kondition* von A ist durch $\kappa(A) = \|A^{-1}\| \cdot \|A\|$ definiert. Das wichtigste Ergebnis besteht darin, dass die Lösung x des linearen Gleichungssystems $Ax = b$ mit δx gestört ist, wenn man A eine (kleine) Störung δA aufprägt und b eine Störung δb:

$$\frac{\|\delta x\|}{\|x\|} \leq \frac{\kappa(A)}{1 - \kappa(A)\|\delta A\|/\|A\|}\left(\frac{\|\delta A\|}{\|A\|} + \frac{\|\delta b\|}{\|b\|}\right).$$

Die Normen $\|\cdot\|_\infty$ und $\|\cdot\|_1$ sind leicht zu berechnen: $\|A\|_\infty = \max_{1\leq i\leq n}(\sum_{j=1}^n |A_{ij}|)$ und $\|A\|_1 = \max_{1\leq j\leq n}(\sum_{i=1}^n)|A_{ij}|$. Im Gegensatz dazu erhalten wir die Norm $\|\cdot\|_2$ nicht so einfach, denn $\|A\|_2 = \sqrt{\rho({}^tAA)}$, der Spektralradius ρ einer Matrix A ist der Absolutwert des absolut größten seiner Eigenwerte.

Die Frobeniusnorm ist definiert durch

$$\|A\|_F = \left(\sum_{i=1}^n \sum_{j=1}^n |a_{ij}|^2\right)^{1/2}.$$

Anders als die übrigen Normen ist sie nicht untergeordnet. Man verifiziert leicht, dass $\|A\|_F^2 = \text{Spur}({}^tAA)$ ist.

Die Berechnung der Normen einer Matrix in Sage. Matrizen besitzen eine Methode `norm(p)`. Je nach Wert des Argumentes p bekommen wir:

$p = 1$:	$\|A\|_1$,	$p = 2$:	$\|A\|_2$,
$p =$ `Infinity` :	$\|A\|_\infty$,	$p =$ `'frob'` :	$\|A\|_F$

Diese Methode ist nur anwendbar, wenn die Koeffizienten der Matrix in komplexe Zahlen `CDF` konvertiert werden können. Anzumerken ist, dass man zwar `A.norm(Infinity)` schreibt, doch `A.norm('frob')`. Mit `A.norm()` bekommen wir die Norm $\|A\|_2$ (Voreinstellung).

Fehler und Kondition - eine Illustration: zuerst mit exakter Rechnung, dann genähert. Wir wollen in Sage die Möglichkeit nutzen, exakte Rechnungen auszuführen, wenn die Koeffizienten rational sind. Wir betrachten die Hilbertmatrix

$$A_{ij} = 1/(i + j - 1),\ i, j = 1, \ldots, n.$$

Das folgende Programm berechnet die Kondition von Hilbertmatrizen in der Norm $\|\cdot\|_\infty$:

```
sage: def cond_hilbert(n):
....:     A = matrix(QQ, [[1/(i+j-1) for j in [1..n]] for i in [1..n]])
....:     return A.norm(Infinity) * (A^-1).norm(Infinity)
```

Hier nun die Ergebnisse als Funktion von n:

n	Kondition
2	27.0
4	28375.0
8	33872791095.0
16	5.06277478751e+22
32	1.35710782493e+47

Wir beobachten ein äußerst rasantes Wachstum der Kondition in Abhängigkeit von n. Man kann zeigen, dass $\kappa(A) \approx e^{7n/2}$, was offensichtlich eine Größe ist, die die sehr schnell wächst. Immer mit exakter Rechnung mit rationalen Zahlen können wir die Matrix A stören und die Lösung des originalen linearen Gleichungssystems mit der des gestörten Systems vergleichen:

```
sage: def diff_hilbert(n):
....:     x = vector(QQ,[1 for i in range(0,n)])
....:     A = matrix(QQ, [[1/(i+j-1) for j in [1..n]] for i in [1..n]])
....:     y = A*x
....:     A[n-1,n-1] = (1/(2*n-1))*(1+1/(10^5))   # verändert die Matrix
....:     sol = A\y
....:     return max(abs(float(sol[i]-x[i])) for i in range(0,n))
```

Wir erhalten

n	Fehler (`diff_hilbert`)
2	3.9998400064e-05
4	0.00597609561753
8	4.80535658002
16	67.9885057468
32	20034.3477421

sodass die Rechnungen sehr bald mit einem nicht mehr tolerierbaren Fehler erfolgen.

Rechnen wir nun mit Matrizen und Vektoren mit Fließpunktkoeffizienten. Diesmal stören wir die Matrix A nicht explizit, doch bringt die Fließpunktarithmetik kleine Störungen mit sich. Wir wiederholen obige Rechnung: das zweite mit $y = Ax$ berechnete Glied versuchen wir auch durch Lösung des linearen Gleichungssystems $As = y$ zu finden:

```
sage: def hilbert_diff(n):
....:     A = matrix(RR, [[1/(i+j-1) for j in [1..n]] for i in [1..n]])
....:     x = vector(RR, [1 for i in range(0,n)])
....:     y = A*x
....:     s = A.solve_right(y)
....:     return max(abs(float(s[i]-x[i])) for i in range(0,n))
```

Abhängig von n erhalten wir:

n	Fehler (`diff_hilbert`)
2	5.55111512313e-16
4	4.7384318691e-13
8	6.82028985288e-07
16	8.68515288279
32	378.704279105

Wir sehen, dass zum Beispiel für $n = 16$ der Fehler (mit Unendlich-Norm) so groß ist, dass alle Ziffern des Ergebnisses falsch sind (bei unserer Wahl von x mit $x_i = 1$ fallen absoluter und relativer Fehler zusammen).

Bemerkungen. Warum also mit Fließpunktzahlen rechnen? Die Frage der Leistungsfähigkeit ist nicht so vordringlich, da Bibliotheken existieren, welche die Algorithmen der linearen Algebra in rationaler Arithmetik effizient implementieren (Sage verwendet `Linbox`), Algorithmen, die ohne so leistungfsfähig zu sein wie wie ihre Fließpunkt-Entsprechungen, für bestimmte Aufgaben wie die Lösung von linearen Gleichungssystem moderater Größe vorteilhaft verwendet werden können. Doch hier kommt ein zweiter Unsicherheitsfaktor ins Spiel: bei reellen Anwendungen können die Koeffizienten nur näherungsweise bekannt sein. Beispielsweise lässt die Lösung eines nichtlinearen Gleichungssystems mit dem Newton-Verfahren Terme auftreten, die unexakt berechnet worden sind.

Schlecht konditionierte lineare Gleichungssysteme (auch ohne so extreme Fälle wie die Hilbert-Matrix) sind eher die Regel als die Ausnahme: häufig treffen wir (in Pysik, Chemie, Biologie usw.) auf Systeme gewöhnlicher Differentialgleichungen der Form $du/dt = F(u)$, wo die Jacobi-Matrix $DF(u)$, also die Matrix der partiellen Ableitungen $\partial F_i/\partial u_j$ ein schlecht konditioniertes lineares System definiert: die Eigenwerte sind in einer sehr ausgedehnten Menge verteilt, was die schlechte Kondition von $DF(u)$ nach sich zieht; diese Eigenschaft rührt daher, dass das System aus mehreren Zeitskalen zusammengesetzte Erscheinungen modelliert. Leider müssen in der Praxis lineare Systeme mit der Matrix $DF(u)$ gelöst werden.

Alle Rechnungen (Zerlegung von Matrizen, Berechnung von Eigenwerten und -vektoren, Konvergenz von iterativen Verfahren) werden durch die Kondition beeinflusst. Deshalb sollten wir diesen Begriff immer im Hinterkopf haben, wenn wir mit Fließpunkt-Darstellungen reeller Zahlen arbeiten.

13.2. Voll besetzte Matrizen

13.2.1. Lösung linearer Gleichungssysteme

Zu vermeidende Verfahren. Was man (fast) nie verwenden sollte, ist die Cramersche Regel. Überlegungen zur Rekursion zeigen. dass der Aufwand für die Berechnung der Determinante einer $n \times m$-Matrix mit der Cramerschen Regel von der Ordnung $n!$ Multiplikationen ist (und ebenso vielen Additionen). Für die Lösung eines System der Größe n sind $n+1$ Determinanten zu berechnen. Nehmen wir $n = 20$:

```
sage: n = 20; aufwand = (n+1)*factorial(n); aufwand
51090942171709440000
```

Wir bekommen den respektablen Wert von 51090942171709440000 Multiplikationen. Wenn wir annehmen, dass unser Compoter pro Sekunde $3 \cdot 10^9$ Multiplikationen ausführt (was realistisch ist), kommt die folgende Rechenzeit zustande:

```
sage: v = 3*10^9
sage: print "%3.3f"%float(aufwand/v/3600/24/365)
540.028
```

Die Berechnung braucht also etwa 540 Jahre! Selbstverständlich können Sie eine 2×2-System mit der Cramerschen Regel lösen, darüber aber nicht! Allen in der Praxis benutzten Verfahren ist polynomialer Aufwand gemein, d.h. die Ordnung n^p mit kleinem p (üblicherweise $p = 3$).

Praktikable Verfahren. Die Lösung linearer Gleichungssysteme $Ax = b$ basiert meistens auf einer Faktorisierung der Matrix A in ein Produkt von zwei Matrizen $A = M_1 M_2$, wobei M_1 und M_2 lineare Systeme definieren, die leicht lösbar sind. Zur Lösung von $Ax = b$ löst man nacheinander erst $M_1 y = b$ und dann $M_2 x = b$.

Beispielsweise können M_1 und M_2 Dreiecksmatrizen sein; in diesem Fall sind zwei Gleichungssysteme in Dreiecksgestalt zu lösen, nachdem die Faktorisierung bewerkstelligt ist. Der Aufwand für die Faktorisierung ist deutlich höher als der für die Lösung der beiden Gleichungssysteme (z.B. $\mathcal{O}(n^3)$ für die LU-Zerlegung gegenüber $\mathcal{O}(n^2)$ für die Lösung der Systeme in Dreiecksgestalt). Daher ist es bei der Lösung von mehreren Gleichungssystemen mit der gleichen Matrix zweckmäßig, die Zerlegung nur einmal vorzunehmen. Wichtig ist, dass eine Matrix zur Lösung eines Gleichungssystems *niemals* invertiert wird. Die Inversion erfordert die Zerlegung der Matrix und dann die Lösung von n Gleichungssystemen, statt eines einzigen.

13.2.2. Direkte Lösung

Dies ist die einfachste Vorgehensweise:

```
sage: A = matrix(RDF, [[-1,2],[3,4]])
sage: b = vector(RDF, [2,3])
sage: x = A\b; x
(-0.20000000000000018, 0.9000000000000001)
```

In Sage besitzen Matrizen zur Lösung linearer Gleichungssysteme eine Methode `solve_right` (die auf der QR-Zerlegung basiert); diesen Befehl rufen wir jetzt auf:

```
sage: x = A.solve_right(b)
```

Die Syntax ist praktisch die gleiche wie in Matlab, Octave oder Scilab.

13.2.3. Die LU-Zerlegung

```
sage: A = matrix(RDF, [[-1,2],[3,4]])
sage: P, L, U = A.LU()
```

Diese Methode liefert die Faktoren L und U sowie die Permutationsmatrix P, sodass $A = PLU$ ist (oder damit gleichwertig: $PA = LU$, wie der aufmerksame Leser bemerkt haben wird). Es ist die Wahl der Pivotstellen, wodurch die Bildung von P bestimmt wird. L ist eine untere Dreiecksmatrix mit einer Einser-Diagonalen und U ist eine obere Dreiecksmatrix. Dieses Verfahren ist direkt aus dem in Unterabschnitt 8.2.1 beschriebenen Gauß-Algorithmus abgeleitet. Der Übersichtlichkeit halber übergehen wir die Wahl der Pivotstellen. Dann bewirkt der Gauß-Algorithmus, dass Nullen in der ersten Spalte unter der Diagonale erscheinen, dann in der zweiten Spalte unter der Diagonalen usw. Diese Eliminierung kann man so schreiben:

$$L_{n-1} \cdots L_2 L_1 A = U.$$

Setzen wir $L = (L_{n-1} \cdots L_2 L_1)^{-1}$, erhalten wir $A = LU$ und wir verifizieren ohne Schwierigkeit, dass L in unterer Dreiecksgestalt ist mit einer Einser-Diagonalen.

Wir machen darauf aufmerksam, dass Sage die Faktorisierung von A speichert: der Befehl `A-LU_valid` gibt `True` genau dann zurück, wenn die LU-Zerlegung bereits berechnet worden ist. Besser noch, der Befehl `A.solve_right(b)` berechnet die Faktorisierung nur, wenn das notwendig ist, also wenn sie zuvor noch nicht berechnet worden ist oder wenn die Matrix A verändert wurde.

Beispiel. Wir erzeugen eine Zufallsmatrix der Größe 1000 und einen Vektor dieser Größe,

```
sage: A = random_matrix(RDF, 1000)
sage: b = vector(RDF, range(1000))
```

faktorisieren A

```
sage: %time A.LU()
CPU times: user 204 ms, sys: 16 ms, total: 220 ms
Wall time: 252 ms
```

und lösen jetzt das System $Ax = b$:

```
sage: %time x = A.solve_right(b)
CPU times: user 116 ms, sys: 8 ms, total: 124 ms
Wall time: 122 ms
```

Die Lösung geht schneller, weil sie die schon berechnete LU-Zerlegung verwendet.

13.2.4. Die Cholesky-Zerlegung reeller positiv definiter symmetrischer Matrizen

Eine symmetrische Matrix A heißt positiv definit, wenn für jeden Vektor x, der nicht der Nullvektor ist, ${}^t xAx > 0$ gilt. Zu jeder symmetrischen positiv definiten Matrix existiert eine untere Dreiecksmatrix C, sodass $A = C^t C$. Diese Faktorisierung ist die Cholesky-Zerlegung. In Sage wird sie durch Aufruf der Methode `cholesky` berechnet. Im folgenden Beispiel erzeugen wir eine Matrix A, die fast sicher positiv definit ist:

```
sage: m = random_matrix(RDF, 10)
sage: A = transpose(m)*m
sage: C = A.cholesky()
```

Wir haben anzumerken, dass es mit vertretbarem Aufwand nicht möglich ist zu testen, ob eine symmetrische Matrix positiv definit ist. Wenn wir die Methode von Cholesky auf eine Matrix anwenden, die es nicht ist, kann die Zerlegung nicht berechnet werden und es wird im Verlauf der Rechnung von `cholesky()` ein `ValueError` ausgegeben.

Um ein Gleichungssystem $Ax = b$ mit der Cholesky-Zerlegung zu lösen, gehen wir vor wie bei der LU-Zerlegung. Ist die Zerlegung einmal berechnet, führen wir `A.solve_right(b)` aus. Auch hier wird die Zerlegung nicht wiederholt.

Warum verwendet man die Cholesky-Zerlegung und nicht die LU-Zerlegung zur Lösung von Systemen mit symmetrischen positiv definiten Matrizen? Natürlich benötigen die Faktoren nur halb soviel Speicherplatz, doch es geht vor allem um die Anzahl der Operationen, wo sich die Cholesky-Zerlegung als vorteilhaft erweist. Für eine Matrix der Größe n müssen bei der Cholesky-Zerlegung n Quadratwurzeln berechnet werden, $n(n-1)/2$ Divisionen, $(n^3-n)/6$ Additionen und ebenso viele Multiplikationen. Zum Vergleich: die LU-Zerlegung benötigt auch $n(n-1)/2$ Divisionen, aber $(n^3-n)/3$ Additionen und Multiplikationen.

13.2.5. Die QR-Zerlegung

Sei $A \in \mathbb{R}^{n\times m}$ mit $n \geq m$. Hier sind zwei Matrizen Q und R zu finden, sodass $A = QR$ ist, wobei $Q \in \mathbb{R}^{n\times n}$ orthogonal ist (${}^tQQ = I$) und $R \in \mathbb{R}^{n\times m}$ in oberer Dreiecksgestalt ist. Ist die Zerlegung einmal berechnet, kann man sie natürlich zur Lösung von linearen Gleichungssystemen benutzen, wenn die Matrix A quadratisch und invertierbar ist. Wie wir aber sehen werden, ist sie vor allem eine interessante Zerlegung für die Lösung von Systemen kleinster Quadrate und für die Berechnung von Eigenwerten. Zu beachten ist, dass A nicht quadratisch sein muss. Die Zerlegung existiert, wenn der Rang maximal ist, also gleich m. Beispiel:

```
sage: A = random_matrix(RDF,6,5)
sage: Q, R = A.QR()
```

Übung 45 *(Störung eines linearen Systems).* Sei A eine invertierbare quadratische Matrix, zu der eine Zerlegung berechnet worden ist (LU, QR, *Cholesky* usw.). Seien u und v zwei Vektoren. Wir betrachten die Matrix $B = A + u^tv$ und setzen $1 + {}^tvA^{-1}u \neq 0$ voraus. Wie ist das System $Bx = f$ mit geringstem Aufwand (d.h. ohne B zu faktorisieren) zu lösen?

Zu verwenden ist die Gleichung von Sherman und Morrison (die man beweisen oder einfach übernehmen kann):

$$(A + u\,{}^tv)^{-1} = A^{-1} - \frac{A^{-1}u\,{}^tvA^{-1}}{1 + {}^tvA^{-1}u}.$$

13.2.6. Die Singulärwertzerlegung

Diese Zerlegung wird zwar selten unterrichtet, ist aber dennoch reich an Anwendungen! Sei A eine $n \times m$-Matrix mit reellen Koeffizienten. Dann existieren zwei orthogonale Matrizen $U \in \mathbb{R}^{n\times n}$ und $V \in \mathbb{R}^{m\times m}$, so dass

$${}^tU \cdot A \cdot V = \Sigma = \mathrm{diag}(\sigma_1, \sigma_2, \ldots, \sigma_p),$$

wobei $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_p \geq 0$ (mit $p = \min(m, n)$). Die Zahlen $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_p$ sind die *Singulärwerte* von A.

Die Matrizen U und V sind orthogonal ($U \cdot {}^tU = I$) und $V \cdot {}^tV = I$), woraus folgt:

$$A = U\Sigma\,{}^tV.$$

Beispiel (die Rechnungen sind offensichtlich niemals exakt):

```
sage: A = matrix(RDF, [[1,3,2],[1,2,3],[0,5,2],[1,1,1]])
sage: U, Sig, V = A.SVD()
sage: A1 = A - U*Sig*transpose(V); A1
[ 2.220446049250313e-16                    0.0                    0.0]
[3.3306690738754696e-16 -4.440892098500626e-16 -4.440892098500626e-16]
[-9.298117831235686e-16 1.7763568394002505e-15 -4.440892098500626e-16]
[ 4.440892098500626e-16 -8.881784197001252e-16 -4.440892098500626e-16]
```

Es kann gezeigt werden, dass die Singulärwerte einer Matrix A die Quadratwurzeln der Eigenwerte von ${}^tAA$ sind. Es ist leicht zu verifizieren, dass bei einer quadratischen Marix der Größe n die euklidsche Norm $\|A\|_2$ gleich σ_1 ist und dass, wenn die Matrix außerdem nicht singulär ist, die Kondition von A mit der euklidschen Norm gleich σ_1/σ_n ist. Der Rang von A ist die natürliche Zahl r, die definiert ist durch

$$\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_r \geq \sigma_{r+1} = \ldots = \sigma_p = 0.$$

13.2.7. Anwendung auf kleinste Quadrate

Wir möchten das überbestimmte Gleichungssystem $Ax = b$ lösen, wobei A eine Matrix mit reellen Koeffizienten ist, rechtwinklig mit n Zeilen und m Spalten und $n > m$. Selbstverständlich hat dieses System nicht immer eine Lösung. Wir wenden uns nun der Aufgabe zu, das Quadrat der euklidschen Norm $\|\cdot\|_2$ des Restes zu minimieren:

$$\min_x \|Ax - b\|_2^2.$$

Die Matrix A kann einen Rang kleiner als m haben.

Das Lösen normaler Gleichungen. Wenn wir bei der Minimierungsaufgabe die Ableitung nach x gleich null setzen, stellen wir ohne große Mühe fest, dass die Lösung dieser Bedingung genügt:

$${}^tAAx = {}^tAb.$$

Wenn wir voraussetzen, dass A den maximalen Rang m hat, können wir die Matrix ${}^tAA$ bilden und das System ${}^tAAx = {}^t Ab$ lösen, beispielsweise durch Berechnung der Cholesky-Zerlegung von ${}^tAA$. Das ist auch der Ursprung des *Verfahrens des Kommandeurs Cholesky*[1] Welches ist die Kondition von ${}^tAA$? Die wird die Genauigkeit der Rechnung konditionieren. Die Singulärwerte von ${}^tAA$ (Dimension $m \times m$) sind die Quadrate der Singulärwerte von A; Kondition mit euklidscher Norm ist daher σ_1^2/σ_m^2, also recht groß. Wir ziehen daher Verfahrewn vor, die entweder auf der QR-Zerlegung von A basieren oder auf der Singulärwertzerlegung[2].

[1] Ingenieur und Offizier der Artillerie (1875-1918); das Verfahren ist zur Lösung geodätischer Aufgaben erfunden worden.

[2] Der Kommandeur Cholesky hatte noch keinen Computer und sah sich wahrscheinlich nur der Lösung kleiner Systeme gegenüber, bei denen schlechte Kondition nicht wirklich problematisch ist.

Trotz allem ist das Verfahren für kleine Systeme, die nicht zu schlecht konditioniert sind, brauchbar. Hier der entsprechende Code:

```
sage: A = matrix(RDF, [[1,3,2],[1,4,2],[0,5,2],[1,3,2]])
sage: b = vector(RDF, [1,2,3,4])
sage: Z = transpose(A)*A
sage: C = Z.cholesky()
sage: R = transpose(A)*b
sage: Z.solve_right(R)
(-1.5, -0.5, 2.75)
```

Wohlgemerkt, die Cholesky-Zerlegung ist hier verborgen, und `Z.solve_right(R)` verwendet sie ohne erneute Berechnung.

Mit QR-Zerlegung. Wir setzen A mit maximalem Rang[3] voraus. Es sei $A = QR$. Dann gilt

$$\|Ax - b\|_2^2 = \|QRx - b\|_2^2 = \|Rx - {}^tQb\|_2^2.$$

Wir haben $R = \begin{bmatrix} R_1 \\ 0 \end{bmatrix}$, wobei R_1 ein Block der Größe m in oberer Dreiecksgestalt ist und ${}^tQb = \begin{bmatrix} c \\ d \end{bmatrix}$ mit c der Größe m. Daher ist $\|Ax - b\|_2^2 = \|R_1x - c\|_2^2 + \|d\|_2^2$, und wir bekommen das minimale x als Lösung des dreieckigen Systems $R_1x = c$:

```
sage: A = matrix(RDF, [[1,3,2],[1,4,2],[0,5,2],[1,3,2]])
sage: b = vector(RDF, [1,2,3,4])
sage: Q, R = A.QR()
sage: R1 = R[0:3,0:3]
sage: b1 = transpose(Q)*b
sage: c = b1[0:3]
sage: R1.solve_right(c)
(-1.499999999999999, -0.49999999999999867, 2.749999999999997)
```

Wir berechnen die Konditionszahl von ${}^tAA$ mit der Unendlich-Norm:

```
sage: Z = A.transpose()*A
sage: Z.norm(Infinity)*(Z^-1).norm(Infinity)
1992.3750000000168
```

Da das System nur klein ist, ist es nicht zu schlecht konditioniert. Die QR-Zerlegung und das Verfahren für normale Gleichungen (Seite 280) ergeben deshalb das gleiche Resultat.

Mit Singulärwertzerlegung. Auch die Singulärwertzerlegung $A = U\Sigma\,{}^tV$ ermöglicht die Berechnung einer Lösung; mehr noch, sie ist sogar dann anwendbar, wenn A nicht den maximalen Rang hat. Wenn A nicht identisch null ist, besitzt Σ $r \leq m$ positive Koeffizienten σ_i (in fallender Ordnung). Mit u_i bezeichnen wir die Spalten von U. Dann haben wir:

$$\|Ax - b\|_2^2 = \|\,{}^tUAV\,{}^tVx - {}^tUb\|_2^2.$$

[3] Diese Einschränkung kann entfallen, wenn wir QR-Zerlegung mit Pivotstellen verwenden.

Mit der Abkürzung $\lambda = {}^t Vx$ erhalten wir

$$\|Ax - b\|_2^2 = \sum_{i=1}^{p} (\sigma_i \lambda_i - {}^t u_i b)^2 + \sum_{i=p+1}^{m} ({}^t u_i b)^2.$$

Das Minimum ist daher zu erwarten, wenn wir $\lambda_i = ({}^t u_i b)/\sigma_i$ setzen mit $1 \leq i \leq p$. Wir wählen dann $\lambda_i = 0$ mit $i > p$ und erhalten schließlich die Lösung $x = V\lambda$.

Hier das Sage-Programm (wir müssen `U.column(i)` nicht transponieren):

```
sage: A = matrix(RDF, [[1,3,2],[1,3,2],[0,5,2],[1,3,2]])
sage: b = vector(RDF, [1,2,3,4])
sage: U, Sig, V = A.SVD()
sage: m = A.ncols()
sage: x = vector(RDF, [0]*m)
sage: lamb = vector(RDF, [0]*m)
sage: for i in range(0,m):
....:     s = Sig[i,i]
....:     if s < 1e-12:
....:         break
....:     lamb[i] = U.column(i)*b / s
sage: x = V*lamb; x
(0.2370370370370367, 0.4518518518518521, 0.3703703703703702)
```

Man sieht, dass die obige Matrix den Rang 2 hat (was wir mit dem Befehl `A.rank()` verifizieren können), also nicht den maximalen Rang, der 3 betragen würde. Es gibt somit mehrere Lösungen für das Problem der kleinsten Quadrate, und der oben angegebene Beweis zeigt, das x die Lösung der minimalen euklidschen Norm ist.

Betrachten wir die Singulärwerte:

```
sage: m = 3
sage: [ Sig[i,i] for i in range(0,m) ]
[8.309316833256451, 1.3983038884881154, 0.0]
```

Da die Matrix A den Rang 2 hat, ist der dritte Singulärwert zwangsläufig 0. Um eine Division durch 0 zu vermeiden, ignorieren wir Singulärwerte, die sich von 0 zu wenig unterscheiden (das ist der Grund für den Test `if s < 1e-12` im Programm).

Beispiel. Unter den schönsten Anwendungen der Singulärwertzerlegung (SVD engl. für Singular Value Decomposition, SWZ auf deutsch), die mit einem anderen Verfahren wohl kaum lösbar wäre: es seien A und $B \in \mathbb{R}^{n \times m}$ die Ergebnisse eines zweimal wiederholten Experiments. Wir fragen uns, ob B auf A *gedreht* werden kann, d.h. existiert eine orthogonale Matrix Q, sodass $A = BQ$ ist. Wird den Messungen ein Rauschen hinzugefügt, hat die Aufgabe generell offensichtlich keine Lösung. Zweckmäßigerweise wird sie dann in kleinsten Quadraten gestellt; dafür muss deshalb die orthogonale Matrix Q berechnet werden, die das Quadrat der Frobenius-Norm minimiert:

$$\|A - BQ\|_F^2,$$

Wir erinnern uns an $\|A\|_F^2 = \text{Spur}({}^t AA)$. Dann ist

$$\|A - BQ\|_F^2 = \text{Spur}({}^t AA) + \text{Spur}({}^t BB) - 2\text{Spur}({}^t Q\, {}^t BA) \geq 0,$$

und daher ist Spur(${}^tQ\,{}^tBA$) zu maximieren. Wir berechnen nun die SWZ von ${}^tBA$: wir haben ${}^tU({}^tBA)V = \Sigma$. Seien die σ_i die Singulärwerte und $O = {}^t V\,{}^tQU$. Diese Matrix ist orthogonal und deshalb sind alle Koeffizienten absolut kleiner oder gleich 1. Dann gilt:

$$\text{Spur}({}^tQ\,{}^tBA) = \text{Spur}({}^tQU\Sigma\,{}^tV) = \text{Spur}(O\Sigma) = \sum_{i=1}^{m} O_{ii}\sigma_i \leq \sum_{i=1}^{m} \sigma_i$$

und das Maximum wird für $Q = U\,{}^tV$ erwartet.

```
sage: A = matrix(RDF, [[1,2],[3,4],[5,6],[7,8]])
```

Man erhält B durch Hinzufügen eines zufälligen Rauschens zu A und durch eine Rotation um den Winkel ϑ (Rotationsmatrix R):

```
sage: th = 0.7
sage: R = matrix(RDF, [[cos(th),sin(th)],[-sin(th),cos(th)]])
sage: B = (A + 0.1*random_matrix(RDF,4,2)) * transpose(R)
sage: C = transpose(B)*A
sage: U, Sigma, V = C.SVD()
sage: Q = U*transpose(V)
```

Die zufällige Störung ist schwach, und wie erwartet ist Q nahe bei R:

```
sage: Q
[ 0.7628142512578093  0.6466176753522814]
[-0.6466176753522812  0.7628142512578091]
sage: R
[0.7648421872844885  0.644217687237691]
[-0.644217687237691 0.7648421872844885]
```

Übung 46 *(Quadratwurzel einer symmetrischen positiv semidefiniten Matrix.)* Sei A eine symmetrische positiv semidefinite Matrix (d.h. sie erfüllt ${}^txAx \geq 0$ für jeden Vektor x). Zu beweisen ist, dass man eine Matrix X berechnen kann, die ebenfalls symmetrisch und positiv semidefinit ist und zwar so, dass $X^2 = A$ ist.

13.2.8. Eigenwerte, Eigenvektoren

Bis jetzt haben wir nur direkte Verfahren verwendet (LU-, QR- und Cholesky-Zerlegung), die nach einer endlichen Anzahl von Operationen (den vier Grundrechenarten und dem Ziehen der Quadratwurzel bei der Cholesky-Zerlegung) eine Lösung liefern. Das *kann* bei der Berechnung der Eigenwerte nicht der Fall sein: Tatsächlich kann man (siehe Seite 287) jedem Polynom eine Matrix zuordnen, deren Eigenwerte die Wurzeln des Polynoms sind; wir wissen aber, dass keine explizite Formel für die Berechng der Wurzeln eines Polynoms höheren als 4. Grades existiert, eine Formel also, die ein direktes Verfahren ergäbe. Andererseits wäre die Bildung des charakteristischen Polynoms, um daraus die Wurzeln zu berechnen, extrem aufwendig (siehe Seite 277): jedenfalls stellen wir fest, dass der Algorithmus von Faddeev-Le Verrier die Berechnung des charakterischen Polynoms einer Matrix der Größe n mit $\mathcal{O}(n^4)$ Operationen ermöglicht, was trotzdem als recht aufwendig betrachtet wird. Die numerischen Verfahren für die Berechnung von Eigenwerten und -vektoren sind sämtlich iterativ.

Wir werden deshalb Folgen bilden, die gegen die Eigenwerte (und die Eigenvektoren) konvergieren und die Iterationen beenden, sobald wir der Lösung nahe genug sind[4].

Das Verfahren der iterierten Potenz. Sei A eine Matrix $\mathbb{C}^{n\times n}$. Wir legen eine beliebige Norm $\|\cdot\|$ auf $\mathbb{C}^n$ fest. Beginnend mit x_0 betrachten wir die Folge der x_k, die definiert ist durch

$$x_{k+1} = \frac{Ax_k}{\|Ax_k\|}.$$

Wenn für die Eigenwerte $|\lambda_1| > |\lambda_2| > \cdots > |\lambda_n|$ gilt, dann konvergiert die Folge gegen einen Eigenvektor, der dem betragsgrößten (dominanten) Eigenwert $|\lambda_1|$ zugeordnet ist. Zudem konvergiert die Folge $v_k =^t x_{k+1}x_k$ gegen $|\lambda_1|$. Die Voraussetzung der Trennung der Eigenwerte kann entfallen.

```
sage:  n = 10
sage: A = random_matrix(RDF, n); A = A*transpose(A)
sage: # A vérifie (presque sûrement) les hypothèses
sage: x = vector(RDF, [1 for i in range(0,n)])
sage: for i in range(0,1000):
....:     y = A*x
....:     z = y/y.norm()
....:     lam = z*y
....:     s = (x-z).norm()
....:     print i, "\ts=", s, "\tlambda=", lam
....:     if s < 1e-10: break
....:     x = z
0  s= 2.56148717112  lambda= 5.91815343752
1  s= 0.444153427849  lambda= 4.73636969543
2  s= 0.213992596115  lambda= 5.81702900649
3  s= 0.203955698809  lambda= 6.33996055612
4  s= 0.191707607395  lambda= 6.8703557299
5  s= 0.158957852503  lambda= 7.31614890304
6  s= 0.120059107245  lambda= 7.61071410257
7  s= 0.0859426802981  lambda= 7.77514327789
8  s= 0.0599388946502  lambda= 7.85860456973
9  s= 0.0413557016799  lambda= 7.89908221139
10  s= 0.028440512895  lambda= 7.91835470562
...
```

Jetzt verwenden wir

```
sage: A.eigenvalues()
[7.9357467710246175,
 0.039892714739813435,
 0.34784684826304274,
 1.2880284207988963,
 1.0559522706787432,
 2.58212362822616,
 3.416559364279167,
```

[4] In den nachstehenden Beispielen wird die Wahl des Kriteriums für den Abbruch der Iteration aus Gründen der Einfachheit ausgespart.

```
 5.537503736446404,
 4.365136545605719,
 4.954088504781742]
```

Wir haben den dominanten Eigenwert berechnet.

Das Interesse an diesem Verfahren mag begrenzt scheinen, es wächst aber an, sobald dünn besetzte Matrizen zu untersuchen sind. Es motiviert auch das, was nun folgt, und das ist von großem Nutzen.

Das Verfahren der inversen Potenz mit Translation. Wir setzen eine *Approximation* μ eines Eigenwertes λ_j mit λ_j (μ und $\lambda_j \in \mathbb{C}$) als bekannt voraus. Wie berechnet man einen λ_j zugordneten Eigenvektor?

Wir treffen die Annahme, dass $\forall k \notin j, 0 < |\mu - \lambda_j| < |\mu - \lambda_k|$ gilt, und somit ist λ_j ein einfacher Eigenwert. Wir betrachten dann den Ausdruck $(A - \mu I)^{-1}$, dessen größter Eigenwert $(\lambda_j - \mu)^{-1}$ ist, und wenden das Verfahren der iterierten Potenz auf diese Matrix an.

Nehmen wir als Beispiel:

```
sage: A = matrix(RDF, [[1,3,2],[1,2,3],[0,5,2]])
```

Mit Aufruf der Methode `A.eigenvalues()` finden wir die Eigenwerte `6.392947916489187, 0.5605194761119395, -1.9534673926011217`. Wir werden den zum zweiten Eigenwert gehörigen Eigenvektor suchen und dazu von einem Näherungswert ausgehen:

```
sage: mu = 0.50519
sage: AT = A - mu*identity_matrix(RDF, 3)
sage: x = vector(RDF, [1 for i in range(0,A.nrows())])
sage: P, L, U = AT.LU()
sage: for i in range(1,10):
....:     y = AT.solve_right(x)
....:     x = y/y.norm()
....:     lamb = x*A*x
....:     print x, lamb
(0.9607985552565431, 0.18570664546988727, -0.2058620364352333)
1.08914936279
(0.927972943625375, 0.1044861051800865, -0.3576994125289455)
0.563839629189
(0.9276916133832703, 0.10329827357690813, -0.35877254233619243)
0.560558807639
(0.9276845318283404, 0.10329496332009931, -0.35879180586954856)
0.560519659839
(0.9276845648427788, 0.1032947552966817, -0.3587917803972978)
0.560519482021
(0.9276845629124328, 0.10329475732310033, -0.3587917848049623)
0.560519476073
(0.9276845629447148, 0.10329475725254855, -0.3587917847418061)
0.560519476114
(0.9276845629438836, 0.1032947572539018, -0.3587917847435661)
0.560519476112
```

```
(0.9276845629439012, 0.10329475725386913, -0.35879178474352963)
0.560519476112
```

Wir fügen einige Bemerkungen an:

- wir berechnen das Inverse der Matrix $A - \mu I$ nicht, sondern verwenden ihre LU-Zerlegung, die (mit `solve_right`) ein für alle Mal berechnet wird;
- wir profitieren für die verbesserte Schätzung des Eigenwertes von den Iterationen;
- die Konvergenz erfolgt sehr schnell; man kann gut zeigen (mit der obigen Annahme sowie der Wahl eines Anfangsvektors, der zu dem Eigenvektors q_j, der zu λ_j gehört, nicht orthogonal ist), dass wir als Iterierte $x^{(i)}$ und $\lambda^{(i)}$

$$\|x^{(i)} - q_j\| = O\left(\left|\frac{\mu - \lambda_j}{\mu - \lambda_K}\right|^i\right)$$

und

$$\|\lambda^{(i)} - \lambda_j\| = O\left(\left|\frac{\mu - \lambda_j}{\mu - \lambda_K}\right|^{2i}\right)$$

bekommen, wobei λ_K der zu μ zweitnächste Eigenwert ist;

- die Konditionszahl von $(A - \mu I)$ (gebunden an das Verhältnis von größtem unf kleinstem Eigenwert von $(A - \mu I)$) ist schlecht; man kann jedoch zeigen (siehe [Sch91], wo Parlett zitiert wird), dass die Fehler trotz allem unbedeutend sind!

Das QR-Verfahren. Sei A eine quadratische nichtsinguläre Matrix. Wir betrachten die Folge $A_0, A_1, A_2, \ldots, A_k, A_{k+1}, \ldots$. In der Rohform des QR-Verfahrens wird der Übergang von A_k nach A_{k+1} so bewerkstelligt:

1. wir berechnen die QR-Zerlegung von A_k: $A_k = Q_k R_k$,
2. wir berechnen $A_{k+1} = R_k Q_k$

Programmieren wir dieses Verfahren mit einer reellen symmetrischen Matrix für A:

```
sage: m = matrix(RDF, [[1,2,3,4],[1,0,2,6],[1,8,4,-2],[1,5,-10,-20]])
sage: Aref = transpose(m)*m
sage: A = copy(Aref)
sage: for i in range(0,20):
....:     Q, R = A.QR()
....:     A = R*Q
....:     print A.str(lambda x: RealField(30)(x).str())
[    347.58031    -222.89331    -108.24117  -0.067928252]
[   -222.89331     243.51949     140.96827   0.081743964]
[   -108.24117     140.96827     90.867499 -0.0017822044]
[ -0.067928252   0.081743964 -0.0017822044   0.032699348]
...
[     585.03056 -3.2281469e-13 -6.8752767e-14 -9.9357605e-14]
[-3.0404094e-13      92.914265 -2.5444701e-14 -3.3835458e-15]
[-1.5340786e-39  7.0477800e-25      4.0229095  2.7461301e-14]
[ 1.1581440e-82 -4.1761905e-68  6.1677425e-42    0.032266909]
```

Wir beobachten eine schnelle Konvergenz gegen eine quasi-diagonale Matrix. Die Koeffizienten auf der Diagonalen sind die Eigenwerte von A. Wir verifizieren

```
sage: Aref.eigenvalues()
[585.0305586200212, 92.91426499150643, 0.03226690899408103, 4.022909479477674]
```

Man kann die Konvergenz beweisen, wenn die hermitesche Matrix positiv definit ist. Rechnet man mit einer nichtsymmetrischen Matrix, ist es sinnvoll, auf $\mathbb{C}$ zu arbeiten, wo die Eigenwerte *a priori* komplex sind, und wenn das Verfahren konvergiert, tendieren die dreieckigen Teile unterhalb von A_k gegen null, während die Diagonale zu den Eigenwerte tendiert.

Das QR-Verfahren benötigt etliche Verbesserungen, um effizient zu sein, insbesondere weil die sukzessiven QR-Zerlegungen aufwendig sind; bei den gebräuchlichen Verfeinerungen wird im allgemeinen mit der Reduktion der Matrix in eine einfachere Form begonnen (Hessenberg-Form: obere Dreiecksgestalt mit einer Subdiagonalen), was die QR-Zerlegung deutlich weniger aufwendig macht; dann müssen zur Beschleunigung der Konvergenz die geschickt gewählten Translationen $A := A + \sigma I$ ausgeführt werden (siehe z.B [GVL96]). Wir verraten noch, dass dieses Verfahren von Sage angewendet wird, wenn man mit voll besetzten Matrizen (`CDF` oder `RDF`) zu tun hat.

In der Praxis. Die obigen Programme sind hier als pädagogische Beispiele angeführt. Man wird daher im Rahmen des Möglichen die von Sage bereitgestellten Methoden verwenden und die optimierten Routinen der Bibliothek Lapack benutzen. Die Schnittstellen erlauben entweder die Berechnung nur der Eigenwerte oder der Eigenwerte und Eigenvektoren:

```
sage: A = matrix(RDF, [[1,3,2],[1,2,3],[0,5,2]])
sage: A.eigenmatrix_right()
(
[   6.39294791648918                  0.0                  0.0]
[                0.0   0.560519476111939                  0.0]
[                0.0                  0.0 -1.9534673926011215],

[ 0.5424840601106511  0.9276845629439008 0.09834254667424457]
[ 0.5544692861094349 0.10329475725386986  -0.617227053099068]
[ 0.6310902116870117 -0.3587917847435306   0.780614827194734]
)
```

Dieses Beispiel berechnet die Diagonalmatrix der Eigenwerte und die Matrix der Eigenvektoren der Eigenwerte (Eigenvektoren sind die Spalten).

Beispiel (Berechnung der Wurzeln eines Polynoms). Ist ein Polynom (mit reellen oder komplexen Koeffizienten) $p(x) = x^n + a_{n-1}x^{n-1} + \ldots + a_1 x + a_0$ gegeben, dann kann leicht verifiziert werden, dass die Eigenwerte der Begleitmatrix M, die durch $M_{i+1,i} = 1$ und $M_{i,n-1} = -a_i$ definiert ist, die Wurzeln von p sind (siehe Unterabschnitt 8.2.3), was somit ein Verfahren zur Ermittlung der Wurzeln von p liefert:

```
sage: def pol2companion(p):
....:     n = len(p)
....:     m = matrix(RDF,n)
....:     for i in range(1,n):
....:         m[i,i-1]=1
....:     for i in range(0,n):
```

```
....:         m[i,n-1]=-p[i]
....:     return m

sage: q = [1,-1,2,3,5,-1,10,11]
sage: comp = pol2companion(q); comp
[ 0.0 0.0 0.0 0.0 0.0 0.0 0.0 -1.0]
[ 1.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0]
[ 0.0 1.0 0.0 0.0 0.0 0.0 0.0 -2.0]
[ 0.0 0.0 1.0 0.0 0.0 0.0 0.0 -3.0]
[ 0.0 0.0 0.0 1.0 0.0 0.0 0.0 -5.0]
[ 0.0 0.0 0.0 0.0 1.0 0.0 0.0 1.0]
[ 0.0 0.0 0.0 0.0 0.0 1.0 0.0 -10.0]
[ 0.0 0.0 0.0 0.0 0.0 0.0 1.0 -11.0]
sage: racines = comp.eigenvalues(); racines
[0.3475215101190289 + 0.5665505533984981*I,
 0.3475215101190289 - 0.5665505533984981*I,
 0.34502377696179265 + 0.43990870238588275*I,
 0.34502377696179265 - 0.43990870238588275*I,
 -0.5172576143252197 + 0.5129582067889322*I,
 -0.5172576143252197 - 0.5129582067889322*I,
 -1.3669971645459291, -9.983578180965276]
```

In diesem Beispiel wird das Polynom durch die Liste q seiner Koeffizienten von 0 bis $n-1$ dargestellt. Das Polynom $x^2 - 3x + 2$ wird so durch die Liste $q = [2, -3]$ abgebildet.

13.2.9. Polynomiale Angleichung: die Rückkehr des Teufels

Stetige Version. Wir möchten die Funktion $f(x)$ auf dem Intervall $[\alpha, \beta]$ durch ein Polynom $P(x)$ höchstens n-ten Grades annähern. Wir formulieren das Problem in kleinsten Quadraten.

$$\min_{a_0,\ldots,a_n \in \mathbb{R}} J(a_0, \ldots, a_n) = \int_\alpha^\beta (f(x) - \sum_{i=0}^n a_i x^i)^2 dx.$$

Durch Ableitung von $J(a_0, \ldots, a_n)$ nach den Koeffizienten a_i finden wir, dass die $a_0, \ldots, a_n$ Lösungen eines linearen Gleichungssystems $Ma = F$ sind mit $M_{i,j} = \int_\alpha^\beta x^i x^j dx$ und $F_j = \int_\alpha^\beta x^j f(x) dx$. Wenn wir den Fall $\alpha = 0$ und $\beta = 1$ betrachten, sehen wir sofort, dass $M_{i,j}$ die Hilbert-Matrix ist! Es gibt aber ein Mittel: es reicht hin, eine Basis der orthogonalen Polynome zu verwenden (beispielsweise die Basis der Legendre-Polynome, wenn $\alpha = -1$ ist und $\beta = 1$): dann wird die Matrix M zur Einheitsmatrix.

Diskrete Version. Wir betrachten m Beobachtungen $y_1, \ldots, y_m$ eines Phänomens an den Punkten $x_1, \ldots, x_m$. Wir möchten ein Polynom n-ten Grades $\sum_{i=0}^n a_i x^i$ anpassen, das an höchstens $m-1$ dieser Punkte übereinstimmt. Wir minimieren also das Funktional:

$$J(a_0, \ldots, a_n) = \sum_{j=1}^m \left(\sum_{i=0}^n a_i x_j^i - y_j \right)^2.$$

So geschrieben wird die Aufgabe eine Matrix ergeben, die einer Hilbert-Matrix sehr nahe kommt, und das System wird schwer lösbar sein. Wir bemerken jetzt aber, dass $\langle P, Q \rangle =$

$\sum_{j=1}^{m} P(x_j) \cdot Q(x_j)$ ein Skalarprodukt von Polynomen des Grades $n \leq m-1$ definiert. Wir können deshalb zunächst n Polynome mit abgestuften Graden bilden, die für das Skalarprodukt orthonormiert sind, und dann das lineare System diagonalisieren. Wenn wir uns daran erinnern,[5] dass das Verfahren von Gram-Schmidt sich für die Berechnung der orthogonalen Polynome zu einer Rekursion mit drei Termen vereinfacht, suchen wir das Polynom $P_{n+1}(x)$ in der Form $P_{n+1}(x) = xP_n(x) - \alpha_n P_{n-1}(x) - \beta_n P_{n-2}(x)$: genau das macht die Funktion `orthopoly` hierunter (wir stellen hier die Polynome durch die Liste ihrer Koeffizienten dar: zum Beispiel repräsentiert $[1, -2, 3]$ das Polynom $1 - 2x + 3x^2$).

Die Auswertung eines Polynoms mit dem Horner-Schema wird nun so programmiert:

```
sage: def eval(P,x):
....:     if len(P) == 0:
....:         return 0
....:     else:
....:         return P[0]+x*eval(P[1:],x)
```

Nun können wir das Skalarprodukt der beiden Polynome programmieren:

```
sage: def pscal(P,Q,lx):
....:     return float(sum(eval(P,s)*eval(Q,s) for s in lx))
```

und die Operation $P \leftarrow P + aQ$ für die Polynome P und Q:

```
sage: def padd(P,a,Q):
....:     for i in range(0,len(Q)):
....:         P[i] += a*Q[i]
```

Ein etwas ernsthafteres Programm muss eine Exzeption auswerfen, sobald es unpassend verwendet wird; in unserem Fall veranlassen wir die Exzeption wenn $n \geq m$ ist:

```
sage: class BadParamsforOrthop(Exception):
....:     def __init__(self, degreplusun, npoints):
....:         self.deg = degreplusun
....:         self.np = npoints
....:     def __str__(self):
....:         return "degre: " + str(self.deg) + \
....:                " nb. points: " + repr(self.np)
```

Die folgende Funktion berechnet die n orthogonalen Polynome:

```
sage: def orthopoly(n,x):
....:     if n > len(x):
....:         raise BadParamsforOrthop(n-1, len(x))
....:     orth = [[1./sqrt(float(len(x)))]]
....:     for p in range(1,n):
....:         nextp = copy(orth[p-1])
....:         nextp.insert(0,0)
....:         s = []
....:         for i in range(p-1,max(p-3,-1),-1):
....:             s.append(pscal(nextp, orth[i], x))
```

[5] Der Beweis ist nicht schwierig!

```
....:         j = 0
....:         for i in range(p-1,max(p-3,-1),-1):
....:             padd(nextp, -s[j], orth[i])
....:             j += 1
....:         norm = sqrt(pscal(nextp, nextp, x))
....:         nextpn = [nextp[i]/norm for i in range(len(nextp))]
....:         orth.append(nextpn)
....:     return orth
```

Wenn die orthogonalen Polynome $P_0(x), \ldots, P_n(x)$ berechnet sind, ist die Lösung gegeben durch $P(x) = \sum_{i=1}^{n} \gamma_i P_i(x)$ mit

$$\gamma_i = \sum_{j=1}^{m} P_i(x_j) y_j,$$

was wir offensichtlich auf die Basis der Monome $1, x, \ldots, x^n$ zurückführen können.

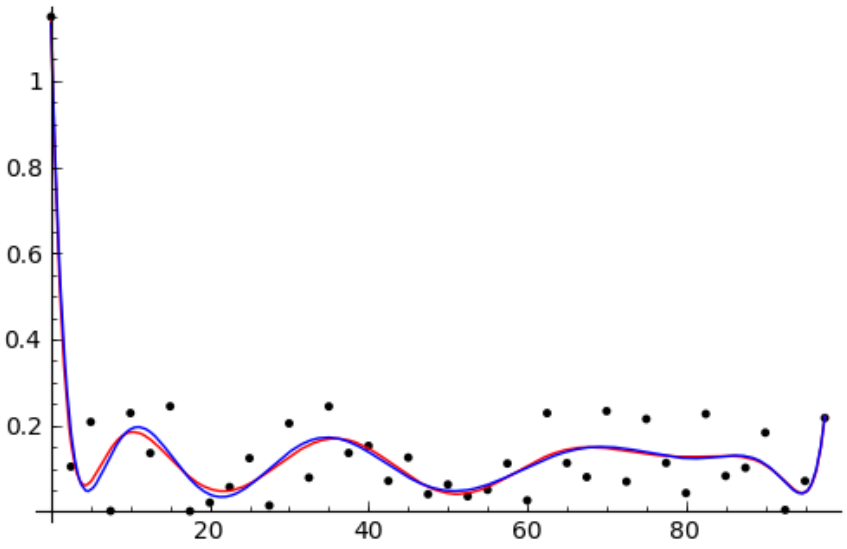


Abb. 13.1 - Polynomiale Anpassung

Beispiel ($n = 15$):

```
sage: L = 40
sage: X = [100*float(i)/L for i in range(40)]
sage: Y = [float(1/(1+25*X[i]^2)+0.25*random()) for i in range(40)]
sage: n 15; orth = orthopoly(n, X)
```

Berechnen wir nun die Lösung mit der Basis der orthogonalen Polynome:

```
sage: coeff = [sum(Y[j]*eval(orth[i],X[j]) for j in
....:         range(0,len(X))) for i in range(0,n)]
```

Jetzt können wir dieses Resultat in die Basis der Monome $1, x, \ldots, x^n$ transformieren, um beispielsweise den Graphen zeichnen zu können:

```
sage: polmin = [0 for i in range(0,n)]
sage: for i in range(0,n):
```

```
....:     padd(polmin, coeff[i], orth[i])
sage: p = lambda x: eval(polmin, x)
sage: plot(p(x), x, 0, X[len(X)-1])
```

Wir setzen hier weder die Berechnung der naiven Anpassung mit den Monomen x^i als Basis auseinander, noch ihre graphische Darstellung. Wir erhalten die Abbildung 13.1. Beide Kurven, die eine entspricht der Anpassung mit orthogonalen Polynomen als Basis, die andere dem naiven Verfahren, liegen dicht beieinander, doch wenn wir ihren Rest berechnen (der Wert des Funktionals J), finden wir 0.1202 für die Anpassung mit orthogonalen Polynomen und 0.1363 für die naive Anpassung.

13.2.10. Implementierung und Leistung

Rechnungen mit Maztrizen mit Koeffizienten aus `RDF` werden mit der Fließpunkt-Arithmetik des Prozessors ausgeführt, diejenigen mit Matzrizen über `RR` mit Hilfe der Bibliothek GNU MPFR. Außerdem bedient sich Sage im ersten Fall der Python-Module NumPy/SciPy, die auf die Bibliothek Lapack zugreifen (die in Fortran codiert ist) und diese Bibliothek verwendet wiederum BLAS[6] und ATLAS, die für jede Maschine optimiert sind. So bekommen wir bei der Berechnung der Produkte zweier Matrizen der Größe 1000 die Angaben

```
sage: a = random_matrix(RR, 1000)
sage: b = random_matrix(RR, 1000)
sage: %time a*b
CPU times: user 421.44 s, sys: 0.34 s, total: 421.78 s
Wall time: 421.79 s

sage: c = random_matrix(RDF, 1000)
sage: d = random_matrix(RDF, 1000)
sage: %time c*d
CPU times: user 0.18 s, sys: 0.01 s, total: 0.19 s
Wall time: 0.19 s
```

und die Rechenzeiten unterscheiden sich um den Faktor 2000!

Wir bemerken auch die Geschwindigkeit der Rechnungen mit Matrizen mit Koeffizienten aus `RDF`: wir sehen sofort, dass das Produkt zweier quadratischer Matrizen der Größe n^3 Multiplikationen erfordert (und ebenso viele Additionen); hier werden also 10^9 Additionen und Multiplikationen in 0.18 Sekunden ausgeführt. Das sind ungefähr 10^{10} Operationen, die pro Sekunde ausgeführt werden oder auch 10 Gigaflops[7]. Die Zentraleinheit der Testmaschine ist mit 3.1 GHz getaktet, es wird also *mehr* als eine Operation je Takt ausgeführt, was nur möglich wird durch quasi direkten Aufruf der entsprechenden Routine der Bibliothek ATLAS[8]. Wir weisen darauf hin, dass auch ein Algorithmus existiert, der den Aufwand bei der Multiplikation zweier Matrizen kleiner macht als n^3: das Verfahren von Strasser. Es ist in der Praxis (für Gleitpunktrechnungen) aus Gründen der numerischen Stabilität nicht implementiert. Der Leser kann sich anhand der obigen Programme überzeugen, dass die Rechnung in Sage proportional zu n^3 ist.

[6] Basic Linear Algebra Subroutines

[7] flop für floating point operation

[8] Diese Bibliothek arbeitet blockweise, wobei die Größe der Blöcke während der Compilierung automatisch angepasst wird. Sie ist teilweise für die beträchtliche Dauer der Erstellung von Sage aus den Quellen verantwortlich.

13.3. Dünn besetzte Matrizen

Dünn besetzte Matrizen treten in wissenschaftlichen Rechnungen häufig auf: dünne Besetzung (*sparsity* auf englisch) ist eine gut erforschte Eigenschaft, die das Lösen von umfangreichen Problemen ermöglicht, die mit voll besetzten Matrizen nicht angegangen werden könnten.

Eine vorläufige Definition: wir sagen, dass eine Menge von Matrizen $\{M_n\}_n$ (der Größe n) eine Familie dünn besetzter Matrizen ist, wenn die Anzahl der von null verschiedenen Koeffizienten von der Ordnung $\mathcal{O}(n)$ ist.

Selbstverständlich werden solche Matrizen rechnerintern mit Datenstrukturen dargestellt, bei denen nur die von null verschiedenen Elemente gespeichert werden. Durch Berücksichtigung der dünnen Besetzung von Matrizen will man natürlich Speicherplatz sparen und daraufhin sehr große Matrizen handhaben können, aber auch die Rechenzeiten deutlich senken.

13.3.1. Vorkommen dünn besetzter Matrizen

Randwertaufgaben. Das häufigste Vorkommen ist die Diskretisierung partieller Differentialgleichungen. Betrachten wir beispielsweise die Poisson-Gleichung (stationäre Wärmeleitung):

$$-\Delta u = f$$

mit $u = u(x, y)$, $f = f(x, y)$,

$$\Delta u := \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}.$$

Die Gleichung ist im Quadrat $[0, 1]^2$ mit der Bedingung $u = 0$ an den Rändern des Quadrats aufgestellt. Das eindimensionale Analogon ist das Problem

$$-\frac{\partial^2 u}{\partial x^2} = f \tag{13.1}$$

mit $u(0) = u(1) = 0$.

Eins der einfachsten Verfahren für eine Näherungslösung dieser Gleichung ist das Finite-Differenzen-Verfahren. Wir zerschneiden das Intervall $[0, 1]$ in eine endliche Anzahl N von Intervallen konstanter Schrittweite h. Mit u_i bezeichnen wir den Näherungswert von u im Punkt $x_i h$. Wir nähern die Ableitung von u durch $(u_{i+1} - u_i)/h$ an und die zweite Ableitung mit

$$\frac{(u_{i+1} - u_i)/h - (u_i - u_{i-1})/h}{h} = \frac{u_{i+1} - 2u_i + u_{i-1}}{h^2}.$$

Man sieht sofort, dass die $u_0, \ldots, u_N$, die u in den Punkten ih annähern, ein lineares Gleichungssystem bilden, das nur drei von null verschiedene Terme je Zeile aufweist (und deshalb können wir sagen, dass die Matrix symmetrisch positiv definit ist.)

Bei zwei Dimensionen können wir ein Netz mit der Maschenweite h über das Einheitsquadrat legen und wir bekommen ein System mit fünf Diagonalen ($4/h^2$ auf der Hauptdiagonalen, zwei Diagonalen darüber und zwei darunter mit den Koeffizienten $-1/h^2$). Bei drei Dimensionen gehen wir in einem Würfel entsprechen vor und bekommen ein System, wo jede Zeile sieben von null verschiedene Koeffizienten hat. Somit haben wir jeweils dünn besetzte Matrizen.

Zufallsweg durch einen großen dünn besetzten Graphen. Wir betrachten einen Graphen, in dem jeder Knoten mit einer kleinen Anzahl von Knoten (klein in Bezug auf die Gesamtzahl der Knoten) verbunden ist. Wir können uns beispielsweise einen Graphen vorstellen, dessen Knoten Internetseiten sind: jede Seite ist nur mit einer kleinen Anzahl von Seiten verlinkt (was die Kanten des Graphen ausmacht), doch handelt es sich mit Sicherheit um einen großen Graphen. Ein Zufallsweg durch diesen Graphen wird durch eine stochastische Matrix beschrieben, also eine Matrix mit reellwertigen Koeffizienten, die alle zwischen 0 und 1 (einschließlich) liegen und deren Summe in jeder Zeile gleich eins ist. Wir zeigen, dass eine solche Matrix einen dominanten Eigenwert von eins hat. Die stationäre Verteilung des Zufallsweges ist der Eigenvektor von links, der dem dominanten Eigenwert zugeordnet ist, d.h. der Vektor, der $Ax = x$ verifiziert. Eine der spektakulärsten Anwendungen ist der *Pagerank*-Algorithmus von Google, wo der Vektor x dazu dient, die Suchergebnisse zu gewichten.

13.3.2. Sage und dünn besetzte Matrizen

Sage gestattet mit dünn besetzten Matrizen zu arbeiten, indem bei der Erzeugung der Matrix `sparse = True` angegeben wird. Es handel sich um eine interne Speicherung als Diktionär. Andererseits werden Berechnungen bei großen dünn besetzten Matrizen mit Fließpunkt-Koeffizienten (`RDF` oder `CDF`) von Sage mit der Bibliothek SciPy ausgeführt, die für dünn besetzte Matrizen eigene Klassen bereitstellt. Beim gegenwärtigen Stand der Dinge gibt es kein Interface zwischen Sages `sparse`-Matrizen und den dünn besetzten Matrizen von SciPy. Zweckmäßigerweise verwendet man direkt die SciPy-Objekte.

Die von SciPy für die Darstellung dünn besetzter Matrizen angebotenen Klassen sind:

- eine Struktur in Gestalt einer Liste von Listen (die aber mit der von Sage verwendeten nicht identisch ist) geeignet für die Erzeugung und Modifizierung von Matrizen, die `lil_matrix`;
- immutable Strukturen, die nur die von null verschiedenen Koeffizienten speichern und praktisch Standard für die dünn besetzte lineare Algebra sind (Formate `csr` und`csv`).

13.3.3. Lösung linearer Systeme

Für ein- und zweidimensionale Systeme mittlerer Größe, wie wir sie oben besprochen haben, können wir ein direktes Verfahren benutzen, das auf der LU-Zerlegung basiert. Ohne große Probleme kann man sich davon überzeugen, dass bei der LU-Zerlegung einer dünn besetzten Matrix A die Faktoren L und U im allgemeinen zusammen mehr von null verschiedene Terme enthalten als A. Wir müssen Algorithmen der Umnummerierung von Unbekannten verwenden, um die Größe des erforderlichen Speichers zu begrenzen, wie in der von Sage auf transparente Weise benutzten Bibliothek `SuperLU`:

```
sage: from scipy.sparse.linalg.dsolve import *
sage: from scipy.sparse import lil_matrix
sage: from numpy import array
sage: n = 200
sage: n2 = n*n
sage: A = lil_matrix((n2, n2))
sage: h2 = 1./float((n+1)^2)
sage: for i in range(0,n2):
....:     A[i,i]=4*h2+1.
```

```
....:     if i+1<n2: A[i,int(i+1)]=-h2
....:     if i>0:    A[i,int(i-1)]=-h2
....:     if i+n<n2: A[i,int(i+n)]=-h2
....:     if i-n>=0: A[i,int(i-n)]=-h2
sage: Acsc = A.tocsc()
sage: b = array([1 for i in range(0,n2)])
sage: solve = factorized(Acsc) # LU-Zerlegung
sage: S = solve(b)             # Lösung
```

Nach Erzeugung der Matrix als `lil_matrix` (Achtung: dieses Format verlangt Indizes vom Python-Typ `int`) muss sie in das Format `csc` konvertiert werden. Dieses Programm ist nicht besonders leistungsfähig: die Bildung der `lil_matrix` geht langsam und die Strukturen `lil_matrix` sind nicht besonders effizient. Dafür sind die Konversion in eine `csc`-Matrix und die Zerlegung schnell, und die sich ergebende Lösung ist es noch mehr.

Die wichtigsten Befehle	
Lösung eines linearen Gleichungssystems	`x = A\b`
LU-Zerlegung	`P, L, U = A.LU()`
Cholesky-Zerlegung	`C = A.cholesky()`
QR-Zerlegung	`Q, R = A.QR()`
Singulärwertzerlegung	`U, Sig, V = A.SVD()`
Eigenwerte und -vektoren	`Val, Vect = A.eigenmatrix_right()`

Tab. 13.1 - Zusammenfassung

Iterative Verfahren. Das Prinzip dieser Verfahren ist die Bildung einer Folge, die gegen die Lösung des Systems $Ax = b$ konvergiert. Moderne iterative Verfahren verwenden den Krylow-Vektorraum K_n, der durch $b, Ab, \ldots, A^n b$ aufgespannt wird. Von den bekanntesten Verfahren nenn wir:

- das Verfahren des konjugierten Gradienten: es kann nur mit Systemen verwendet werden, deren Matrix A symmetrisch und positiv definit ist. In diesem Fall ist $\|x\|_A = \sqrt{{}^t xAx}$ eine Norm und die Iterierte x_n wird durch Minimierung der Abweichung $\|x - x_n\|_A$ zwischen der Lösung x und x_n berechnet mit $x_n \in K_n$ (es gibt explizite, leicht zu programmierende Formeln, siehe z.B. [LT94], [GVL96]);
- das GMRES-Verfahren (Akronym für Generalized Minimal RESidual): es hat den Vorteil, bei nichtsymmetrischen linearen Systemen eingesetzt werden zu können. Bei der n-ten Iteration ist es die euklidsche Norm $\|Ax_n - b\|_2$, die für $x_n \in K_n$ minimiert wird. Man sieht, es handelt sich um eine Aufgabe der kleinsten Quadrate.

In der Praxis sind diese Verfahren nur *vorkonditioniert* effizient: statt $Ax = b$ zu lösen, löst man $MAx = Mb$, wobei M eine Matrix ist, sodass MA besser konditioniert ist als A. Die Untersuchung und das Auffinden von effizienten Vorkonditionierern ist ein aktuelles Gebiet und reich an Entwicklungen der numerischen Analyse. Als Beispiel hier die Lösung des oben mit dem Verfahren des konjugierten Gradienten untersuchten Systems, wobei der Vorkonditionierer M diagonal ist und die Inverse der Diagonalen von A. Das ist ein einfacher, aber wenig effizienter Vorkonditionierer:

```
sage: b = array([1 for i in range(0,n2)])
sage: m = lil_matrix((n2, n2))
sage: for i in range(0,n2):
....:     m[i,i] = 1./A[i,i]
```

```
sage: msc = m.tocsc()
sage: from scipy.sparse.linalg import cg
sage: x = cg(A, b, M = msc, tol=1.e-8)
```

13.3.4. Eigenwerte, Eigenvektoren

Das Verfahren der iterierten Potenz. Das Verfahren der iterierten Potenz ist besonders an den Fall sehr großer dünn besetzter Matrizen angepasst; tatasächlich reicht es aus zu wissen, wie Matrix-Vektor-Produkte (und Skalarprodukte) gebildet werden, um diesen Algorithmus implementieren zu können. Als Beispiel greifen wir den Zufallsweg durch einen dünn besetzten Graphen wieder auf und berechnen mit dem Verfahren der iterierten Potenz die stationäre Verteilung:

```
sage: from scipy import sparse
sage: from numpy.linalg import *
sage: from numpy import array
sage: from numpy.random import rand
sage: def power(A,x):              # iterierte Potenz
....:     for i in range(0,1000):
....:         y = A*x
....:         z = y/norm(y)
....:         lam = sum(x*y)
....:         s = norm(x-z)
....:         print i,"s= "+str(s)+" lambda= "+str(lam)
....:         if s < 1e-3:
....:             break
....:         x = z
....:     return x
sage: n = 1000
sage: m = 5
sage: # Bilden eine stochstischen Matrix der Größe n
sage: # mit m von null verschiedenen Koeffizienten je Zeile
sage: A1 = sparse.lil_matrix((n, n))
sage: for i in range(0,n):
....:     for j in range(0,m):
....:         l = int(n*rand())
....:         A1[l,i] = rand()
sage: for i in range(0,n):
....:     s = sum(A1[i,0:n])
....:     A1[i,0:n] /= s
sage: At = A1.transpose().tocsc()
sage: x = array([rand() for i in range(0,n)])
sage: # Berechnen des dominanten Eigenwertes und
sage: # des zugehörigen Eigenvektors
sage: y = power(At, x)
0 s= 17.637122289 lambda= 255.537958336
1 s= 0.374734872232 lambda= 0.91243996321
2 s= 0.215216267956 lambda= 0.968970901784
3 s= 0.120794893336 lambda= 0.98672095617
```

```
4 s= 0.071729229056 lambda= 0.990593746559
...
```

Führen wir dieses Beispiel aus, können wir uns damit amüsieren, die Rechenzeiten für die verschiedenen Teile zu messen. Wir werden dann feststellen, dass fast die gesamte Rechenzeit für die Erstellung der Matrix verbraucht wird; die Berechnung der Transponierten ist nicht sehr zeitaufwendig; die Iterationen beanspruchen eigentlich nur eine vernachlässigbar kleine Zeit (etwa 2% der gesamten Rechenzeit auf dem Testrechner). Das Speichern großer Matrizen als Listen ist nicht sehr effizient, und solche Probleme sollten besser mit kompilierten Sprachen und angepassten Datenstrukturen behandelt werden.

13.3.5. Gegenstand des Nachdenkens: Lösung von sehr großen nicht-linearen Systemen

Das Verfahren der iterierten Potenz und die auf dem Krylow-Vektorraum basierenden Verfahren teilen eine wertvolle Eigenschaft: es genügt zu wissen, wie Matrix-Vektor-Produkte berechnet werden, um sie implementieren zu können. Wir müssen noch nicht einmal die Matrix kennen, es reicht aus zu wissen, was die Matrix mit einem Vektor macht. Wir können daher auch dann Rechnungen ausführen, wenn wir die Matrix nicht genau kennen oder sie nicht berechnen können. Die Methoden von SciPy akzeptieren tatsächlich *lineare Operatoren* als Argumente. Hier nun eine Anwendung, über die man nachdenken (und die man implementieren) kann.

Sei $F : \mathbb{R}^n \to \mathbb{R}^n$. Wir wollen $F(x) = 0$ lösen. Das Verfahren der Wahl ist das Newton-Verfahren, bei dem wir ausgehend von x_0 die Iterierten $x_{n+1} = x_n - J(x_n)^{-1} \cdot F(x_n)$ berechnen. Die Matrix $J(x_n)$ ist die Jacobi-Matrix von F nach x_n, die Matrix der partiellen Ableitungen von F nach x_n. In der Praxis wird man nacheinander $J(x_n)y_= F(x_)$ und dann $x_{n+1} = x_n - y_n$ lösen. Wir müssen also ein lineares Gleichungssystem lösen. Ist F ein etwas kompliziert zu berechnendes Objekt, dann sind seine Ableitungen im allgemeinen noch viel schwieriger zu berechnen und zu kodieren, und die Rechnung kann sich als praktisch undurchführbar erweisen. Dann nimmt man seine Zuflucht zur numerischen Ableitung: ist e_j der Vektor auf $\mathbb{R}^n$, der überall 0 ist außer der j-ten Komponente, die gleich 1 ist, dann liefert $(F(x + he_j) - F(x))/h$ eine (gute) Näherung der j-ten Spalte der Jacobi-Matrix. Wir müssen also $n + 1$ Auswertungen von F vornehmen, um J näherungsweise zu erhalten, was sehr aufwendig ist, wenn n groß ist. Und wenn wir ein iteratives Verfahren vom Krylow-Typ zur Lösung des Systems $J(x_n)y_n$ einsetzen? Wir sehen dann, dass $J(x_n)V \approx (F(x_n + hV) - F(x_n))/h$ für hinreichend kleines h wird, und das vermeidet die Berechnung der *gesamten* Matrix. In SciPy muss also nur ein „linearer Operator“ definiert werden wie es die Funktion $V \to (F(x_n + hV) - F(x_n))/h$ ist. Solche Verfahren werden aktuell zur Lösung großer, nichtlinearer Gleichungssysteme eingesetzt. Ist die „Matrix“ nicht symmetrisch, wird man beispielsweise GMRES benutzen.

14. Numerische Integration und Differentialgleichungen

Dieses Kapitel behandelt die numerische Berechnung von Integralen (Abschnitt 14.1) sowie die numerische Lösung von gewöhnlichen Differentialgleichungen (Abschnitt 14.2) mit Sage. Wir wiederholen die theoretischen Grundlagen der Integrationsverfahren, danach besprechen wir die verfügbaren Funktionen und ihren Gebrauch im einzelnen (Unterabschnitt 14.1.1).

Die symbolische Integration mit Sage ist bereits behandelt worden (Unterabschnitt 2.3.8), und wird hier nur kursorisch erwähnt als eine Möglichkeit für die Berechnung des numerischen Wertes eines Integrals. Dieses Vorgehen (erst symbolisch, dann rechnerisch) bildet, sofern es möglich ist, eine der Stärken von Sage und ist vorzuziehen, denn die Anzahl der auszuführenden Rechenschritte und damit die Größe der Rundungsfehler ist gemeinhin kleiner als bei der numerischen Integration.

Wir geben eine knappe Einführung in die klassischen Verfahren für die Lösung von Differentialgleichungen, sodann wird die Behandlung eines Beispiels (Unterabschnitt 14.2.1) das Inventar der verfügbaren Funktionen anführen.

14.1. Numerische Integration

Wir interessieren uns für die numerische Berechnung von Integralen reeller Funktionen; für eine Funktion $f : I \to \mathbb{R}$, worin I ein Intervall von $\mathbb{R}$ ist, wollen wir

$$\int_I f(x)dx$$

berechnen. Wir berechen beispielsweise

$$\int_1^3 \exp(-x^2)\ln(x)dx.$$

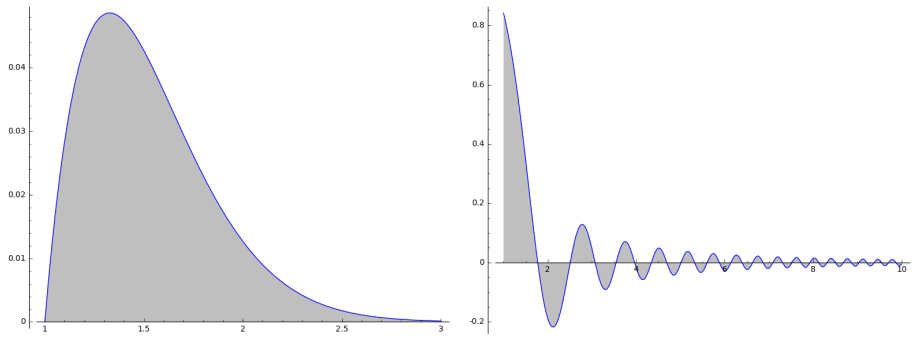


Abb. 14.1 - Die Funktionen $x \mapsto \exp(-x^2)\ln(x)$ und $x \mapsto \frac{\sin(x^2)}{x^2}$.

```
sage: x = var('x'); f(x) = exp(-x^2)*log(x)
sage: N(integrate(f, x, 1, 3))
0.035860294991267694
sage: plot(f, 1, 3, fill='axis')
```

Die Funktion `integrate` berechnet das symbolische Integral des Ausdrucks; um einen Zahlenwert zu erhalten, muss das explizit verlangt werden.

Es ist im Prinzip auch möglich, Integrale auf einem Intervall zu berechnen, das nicht begrenzt ist:

```
N(integrate(sin(x^2)/(x^2), x, 1, infinity))
0.285736646322853 + 6.93889390390723e-18*I
plot(sin(x^2)/(x^2), x, 1, 10, fill='axis')
```

Es existieren in Sage mehrere Verfahren zur numerischen Berechnung eines Integrals, und wenn sich ihre Implementierungen auch technisch unterscheiden, beruhen sie alle auf einem der beiden folgenden Prinzipien:

- polynomiale Interpolation (insbesondere das Verfahren von Gauß-Kronrod);
- Transformation der Funktion (doppelt exponentielles Verfahren).

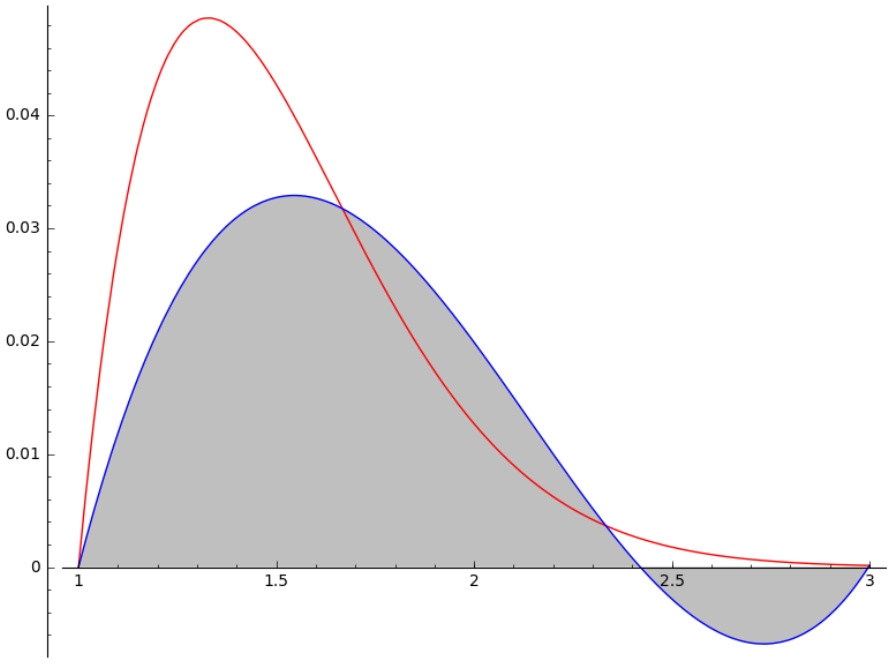
Interpolierende Verfahren. Bei diesen Verfahren wertet man die zu integrierende Funktion f an einer bestimmten Zahl n von zweckmäßig gewählten Punkten $x_1, x_2, \ldots, x_n$ aus und berechnet einen Näherungswert des Integrals von f auf $[a, b]$ mit

$$\int_a^b f(x)dt \approx \sum_{i=1}^{n} w_i f(x_i).$$

Die Koeffizienten w_i werden die Gewichte des Verfahrens genannt. Sie werden durch die Bedingung bestimmt, dass das Verfahren für jedes Polynom f vom Grade $n-1$ genau sein muss. Für die festen Punkte (x_i) sind die Gewichte (w_i) durch diese Bedingung eindeutig bestimmt. Die *Ordnung* des Verfahrens wird als der höchste Grad der exakt integrierten Polynome bestimmt; diese Ordnung ist daher mindestens $n-1$, kann aber auch größer sein.

So wählt auch die Familie der Integrationsverfahren von Newton-Cotes (zu denen auch die Rechteck- die Trapez- und die Regel von Simpson gehören) äquidistante Integrationspunkte auf dem Intervall $[a, b]$:

```
sage: fp = plot(f, 1, 3, color='red')
sage: n = 4
sage: interp_points = [(1+2*u/(n-1), N(f(1+2*u/(n-1)))) for u in xrange(n)]
sage: A = PolynomialRing(RR, 'x')
sage: pp = plot(A.lagrange_polynomial(interp_points), 1, 3, fill='axis')
sage: show(fp+pp)
```



Bei den interpolierenden Verfahren kann man deshalb davon ausgehen, dass wir das Lagrange-Interpolationspolynom der gegebenen Funktion berechnen und dass der Wert dieses Integrals als Näherungswert des gesuchten Integrals ausgegeben wird. Diese beiden Etappen werden in Wirklichkeit in einer Formel zusammengefasst, welche die „Regel“ der Quadratur genannt wird. Das Interpolationspolynom von Lagrange wird aber nicht explizit berechnet, doch beeinflusst die Wahl der Stützstellen die Güte der Näherung erheblich, und die Wahl gleichverteilter Stützstellen stellt die Konvergenz nicht sicher, wenn deren Anzahl steigt (Phänomen von Runge). Dieses Integrationsverfahren kann daher zu Erscheinungen führen, wie sie in Abb. 14.2 zu sehen sind.

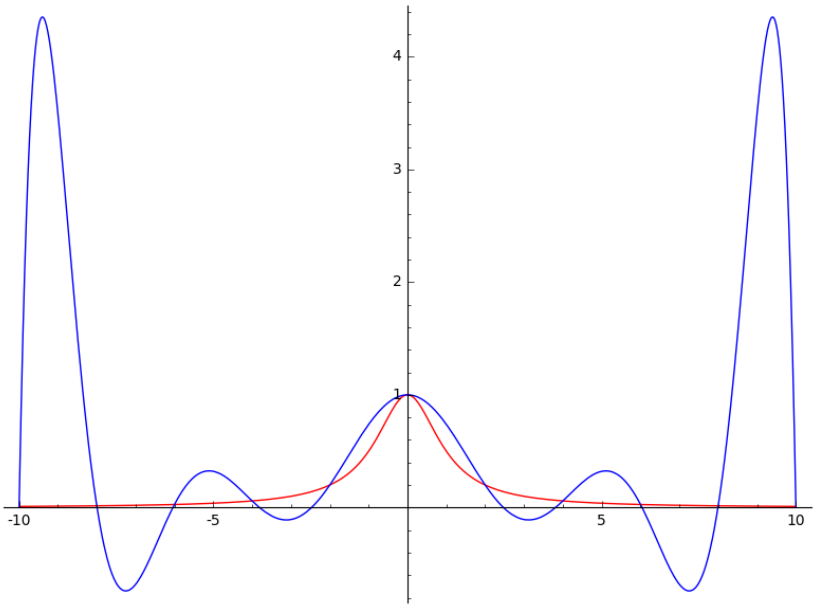


Abb. 14.2 - Interpolation durch ein Polynom 10. Grades (blau) der Funktion $x \mapsto 1/(1+x^2)$ (rot) mit 11 äquidistanten Stützstellen im Intervall $[-10, 10]$. Das Phänomen von Runge zeigt sich an den Rändern.

Wenn wir von Sage verlangen, ein Integral auf einem beliebigen Intervall numerisch zu berechnen, wird das Integrationsverfahren nicht direkt auf das gesamte Intervall angewendet; wir unterteilen den Integrationsbereich in hinreichend kleine Intervalle, damit das elementare Verfahren ein hinreichend genaues Resultat liefert (wir sprechen von Komposition der Verfahren). Als Strategie für die Unterteilung kann man sich beispielsweise dem Verlauf der zu integrierenden Funktion dynamisch anpassen: wenn wir $I_a^b(f)$ den Wert des vom Integrationsverfahren berechneten Wertes von $\int_a^b f(x)\mathrm{d}x$ nennen, vergleichen wir

$$I_0 = I_a^b(f)$$

mit

$$I_1 = I_a^{(a+b)/2}(f) + I_{(a+b)/2}^b(f)$$

und wir beenden die Unterteilung, sobald $|I_0 - I_1|$ klein ist gegenüber der bei der Rechnung verwendeten Genauigkeit. Hier ist der Begriff der Ordnung eines Verfahrens wichtig: bei einem Verfahren n-ter Ordnung teilt die Zweiteilung des Integrationsbereiches den theoretischen Fehler ohne Berücksichtigung des Rundungsfehlers durch 2^n.

Ein in Sage zur Verfügung stehendes besonderes Interpolationsverfahren ist das Verfahren von Gauß-Legendre. Bei diesem Verfahren werden die n Integrationspunkte als Wurzeln des Legendre-Polynoms n-ten Grades bestimmt (mit einem Definitionsbereich, der in das Intervall der vorliegenden Integration, $[a, b]$, korrekt übersetzt ist). Die Eigenschaften der für das Skalarprodukt orthogonalen Legendre-Polynome

$$\langle f, g\rangle = \int_a^b f(x)g(x)\mathrm{d}x$$

bewirken, dass das Integrationsverfahren die Integrale bis $2n-1$-ten Grades genau berechnet und nicht nur bis zum $n-1$-ten Grad, wie man erwarten könnte. Zudem sind die entsprechenden Gewichte der Integration stets positiv, was das Verfahren gegenüber den numerischen Problemen des Typs *Auslöschung*[1] weniger verwundbar macht.

Um mit den Interpolationsverfahren zum Ende zu kommen ist das Verfahren von Gauß-Kronrod an $2n+1$ Punkten eine „Verbesserung“ des Verfahrens von Gauß-Legendre an n Punkten:

- unter den $n+1$ Punkten befinden sich die n Punkte des Verfahrens von Gauß-Legendre,
- das Verfahren ist exakt für jedes Polynom höchstens $3n+1$-ten Grades.

Ganz unvoreingenommen können wir sehen, dass die $3n+2$ Unbekannten ($2n+1$ Gewichte und $n+1$ hinzugenommene Punkte) *a priori* dadurch bestimmt sind, das verlangt wird, dass das Verfahren mindesten $3n+1$-ter Ordnung sein soll (was $3n+2$ Bedingungen ergibt). Aber Vorsicht: die mit dem Verfahren von Gauß-Kronrod verbundenen Gewichte an den n Punkten von Gauß-Legendre müssen nicht diejenigen sein, die ihnen bei dem Verfahren von Gauß-Legendre zugeordnet werden.

Das Interesse an einem solchen verbesserten Verfahren erwächst, sobald wir überlegen, dass der Hauptaufwand bei einem Integrationsalgorithmus die Anzahl der Auswertungen der zu integrierenden Funktion f ist (besonders dann, wenn die Punkte und Gewichte tabelliert

[1] Diese Erscheinung zeigt sich, wenn eine Summe absolut signifikant kleiner ist als die Summanden: jeder Rundungsfehler kann dann größer werden als das Endergebnis, wobei die Genauigkeit völlig verloren geht; siehe auch Unterabschnitt 11.3.3.

sind). Da das Verfahren von Gauß-Kronrod im Prinzip genauer ist als das Verfahren von Gauß-Legendre, können wir sein Ergebnis I_1 verwenden, um das Ergebnis I_0 des letzteren zu bestätigen und eine Abschätzung des Fehlers von $|I_1 - I_0|$ zu bekommen, was die Anzahl der Aufrufe von f verringert. Diese für das Verfahren von Gauß-Legendre eigentümliche Strategie können wir mit der allgemeineren Strategie der Unterteilung vergleichen, die wir auf Seite 300 gesehen haben.

Doppelt exponentielle Verfahren. Die doppelt exponentiellen Verfahren (DE) beruhen einerseits auf einem Wechsel der Variablen, der auf $\mathbb{R}$ einen beschränkten Integrationsbereich transformiert und andererseits auf der mit dem Trapezverfahren auf $\mathbb{R}$ erhaltenen sehr guten Genauigkeit für analytische Funktionen.

Für eine auf $\mathbb{R}$ integrierbare Funktion f und eine Schrittweite h besteht das Trapezverfahren aus der Berechnung von

$$I_h = h \sum_{i=-\infty}^{+\infty} f(hi)$$

als Näherungswert von $\int_{-\infty}^{+\infty} f(x)\mathrm{d}x$. Die doppelt exponentielle Transformation ist 1973 von Takahasi und Mori entwickelt worden und wird aktuell von Programmen für die numerische Integration verwendet. Eine Einführung in die Transformation und ihre Entdeckung wird in [Mor05] mitgeteilt; hier soll das Wesentliche wiedergegeben werden. Man findet darin insbesondere eine Erklärung für die frappierende Genauigkeit, die mit dem Trapezverfahren (das in gewissem Sinne optimal ist) bei analytischen Funktionen erreicht wird.

Zur Berechnung von

$$I = \int_{-1}^{1} f(x)\mathrm{d}x$$

kann eine Transformation $x = \varphi(t)$ benutzt werden, wobei φ auf $\mathbb{R}$ analytisch ist und die Bedingungen

$$\lim_{t\to-\infty} \varphi(t) = -1 \quad \text{und} \quad \lim_{t\to\infty} \varphi(t) = 1$$

erfüllt. Dann ist

$$I = \int_{-\infty}^{\infty} f(\varphi(t))\varphi'(t)\mathrm{d}t.$$

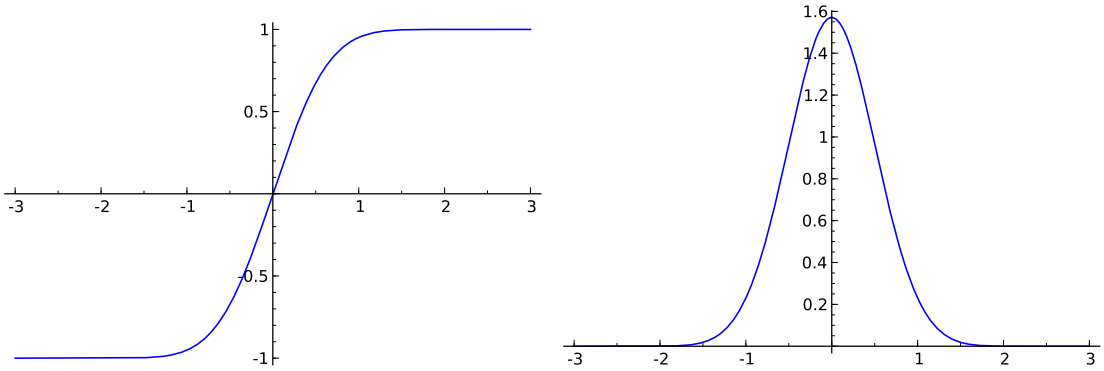


Abb. 14.3 - Die im doppelt exponentiellen Verfahren verwendete Transformation $\varphi(t) = \tanh(\frac{\pi}{2}\sinh t)$ (links) und die Abnahme von $\varphi'(t)$ (rechts).

Wir wenden die Trapezformel auf den letzten Ausdruck an und berechnen

$$I_h^N = h \sum_{k=-N}^{N} f(\varphi(kh))\varphi'(kh)$$

mit einer bestimmten Schrittweite h, wobei wir die Summe auf die Terme von $-N$ bis N beschränken. Die vorgeschlagene Transformation ist

$$\varphi(t) = \tanh\left(\frac{\pi}{2}\sinh t\right).$$

Das ergibt die Formel

$$I_h^N = h \sum_{k=-N}^{N} f\left(\tanh\left(\frac{\pi}{2}\sinh kh\right)\right) \frac{\frac{\pi}{2}\cosh kh}{\cosh^2(\frac{\pi}{2}\sin kh)}.$$

Die Formel verdankt ihren Namen der doppelt exponentiellen Abnahme von

$$\varphi'(t) = \frac{\frac{\pi}{2}\cosh t}{\cosh^2(\frac{\pi}{2}\sinh t)},$$

wenn $|t| \to \infty$ geht (siehe Abb. 14.3). Das Prinzip der Transformation besteht darin, das Wesentliche des Beitrags der zu integrierenden Funktion um 0 herum zu konzentrieren, von wo die starke Abnahme rührt, wenn $|t|$ wächst. Es gilt einen Kompromiss zu finden zwischen der Festlegung der Parameter und der verwendeten Transformation φ: eine schnellere Abnahme als die doppelt exponentielle vermindert den durch den Abbruch entstehenden Fehler, vergrößert jedoch den Fehler durch die Diskretisierung.

Nach der Entwicklung der DE-Transformation wird dieses Verfahren allein oder zusammen mit anderen Transformationen angewendet, je nach der Art des Integranden, seiner Singularitäten und des Integrationsbereiches. Ein solches Beispiel ist die Kardinalsinus-Zerlegung „sinc“:

$$f(x) \approx \sum_{k=-N}^{N} f(kh)S_{k,h}(h)$$

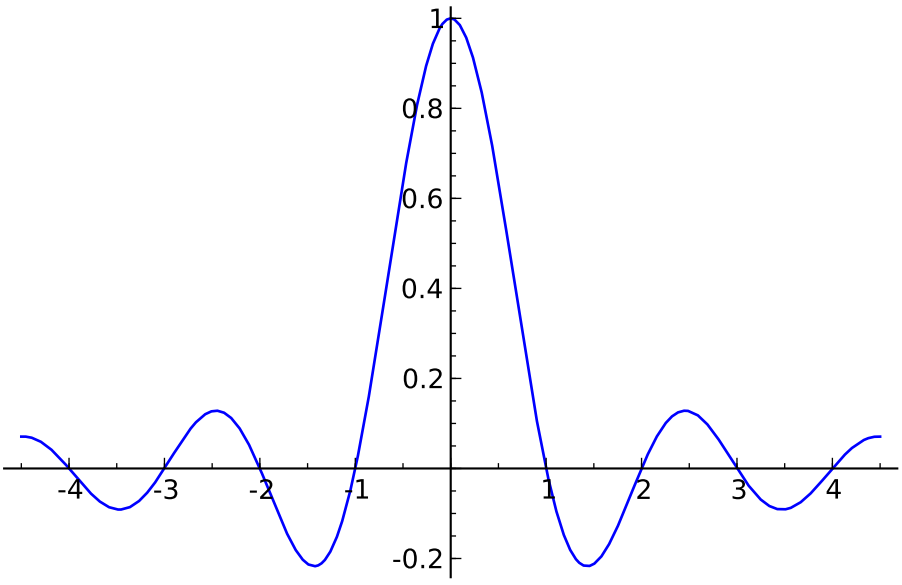


Abb. 14.4 - Die Kardinalsinus-Funktion

mit

$$S_{k,h}(x) = \frac{\sin(\pi(x-kh)/h)}{\pi(x-kh)/h},$$

was zusammen mit dem doppelt exponentiellen Verfahren in [TSM05] zur Verbesserung der vorstehenden Formeln verwendet wird, die nur eine einfach exponentielle Transformation $\varphi(t) = \tanh(t/2)$ benutzen. Die Funktion sinc ist definiert durch

$$\text{sinc} = \begin{cases} 1 & \text{für } x = 0, \\ \dfrac{\sin(\pi x)}{\pi x} & \text{sonst.} \end{cases}$$

Ihren Graphen zeigt die Abbildung 14.4.

Die Auswahl der zu benutzenden Transformation bestimmt zu großen Teilen die Qualität des Ergebnisses bei Vorhandensein von Singularitäten an den Grenzen (bei Singularitäten im Inneren des Intervalls gibt es hingegen keine gute Lösung). Später werden wir sehen, dass in der betrachteten Version von Sage nur die Integrationsfunktionen von PARI doppelt exponentielle Transformationen benutzen mit der Möglichkeit, das Verhalten an den Grenzen zu präzisieren.

14.1.1. Funktionen für die Integration

Wir werden uns nun die verschiedenen Arten der Berechnung eines numerischen Integrals anhand einiger Beispiele der Berechnung von Integralen genauer ansehen:

$$I_1 = \int_{17}^{42} exp(-x^2)\log(x)\mathrm{d}x, \quad I_2 = \int_0^1 x\log(1+x)\mathrm{d}x = \frac{1}{4},$$

$$I_3 = \int_0^1 \sqrt{1-x^2}\mathrm{d}x = \frac{\pi}{4},$$

$$I_4 = \int_0^1 \max(\sin(x),\cos(x))\mathrm{d}x = \int_0^{\frac{\pi}{4}} \cos(x)\mathrm{d}x + \int_{\frac{\pi}{4}}^1 \sin(x)\mathrm{d}x$$

$$= \sin\frac{\pi}{4} + \cos\frac{\pi}{4} - \cos 1 = \sqrt{2} - \cos 1,$$

$$I_5 = \int_0^1 \sin(\sin(x))\mathrm{d}x, \quad I_6 = \int_0^x \sin(x)\exp(\cos(x))\mathrm{d}x = e - \frac{1}{e},$$

$$I_7 = \int_0^1 \frac{1}{1+10^{10}x^2}\mathrm{d}x = 10^{-5}\arctan(10^5), \quad I_8 = \int_0^{1.1} \exp(-x^{100})\mathrm{d}x,$$

$$I_9 = \int_0^{10} x^2\sin(x^3)\mathrm{d}x = \frac{1}{3}(1-\cos(1000)), \quad I_{10} = \int_0^1 \sqrt{x}\mathrm{d}x = \frac{1}{3}.$$

Wir behandeln die Funktionen zur Integration hier nicht erschöpfend - das findet man in der Online-Hilfe - vielmehr nur deren eher normale Verwendung.

`N(integrate(...))`. Das erste Verfahren, das wir mit Sage zur numerischen Berechnung verwenden können, ist `N(integrate(...))`:

```
sage: N(integrate(exp(-x^2)*log(x), x, 17, 42))
 2.5657285006962035e-127
```

Es ist nicht garantiert, dass die Integrale auf diese Weise wirklich *numerisch* berechnet werden, in der Tat verlangt die Funktion `integrate` eine symbolische Integration. Wenn diese möglich ist, wird Sage den erhaltenen symbolischen Ausdruck nur auswerten:

```
sage: integrate(log(1+x)*x, x, 0, 1)
1/4
sage: N(integrate(log(1+x)*x, x, 0, 1))
0.250000000000000
```

Die Funktion `numerical_integral` verlangt im Gegenteil *explizit* eine numerische Integration der als Parameter gegebenen Funktion. Dafür benutzt sie die numerische Bibliothek GSL (GNU Scientific Library), welche das Verfahren von Gauß-Kronrod für eine feste Zahl n von Integrationspunkten implementiert. Die Punkte und Gewichte sind vorberechnet, und die Genauigkeit ist auf die Genauigkeit der Fließpunktzahlen der Maschine begrenzt (53 Bits der Mantisse). Das Resultat ist ein Paar, das sich aus dem berechneten Ergebnis und einer Abschätzung des Fehlers zusammensetzt:

```
sage: numerical_integral(exp(-x^2)*log(x), 17, 42)
(2.5657285006962035e-127, 3.3540254049238093e-128)
```

Die Abschätzung des Fehlers ist keine garantierte Schranke für den eingetretenen Fehler, sondern ein Wert, der einfach auf die Schwierigkeit der Berechnung des gegebenen Integrals verweist. Bei obigem Beispiel stellen wir fest, dass aufgrund der gegebenen Fehlerabschätzng alle Ziffern des Ergebnisses (außer der ersten) zweifelhaft sind.

Die Argumente von `numerical_integral` erlauben vor allem

- die Anzahl der verwendeten Punkte festzulegen (sechs Einstellungen von `rule=1` für 15 Punkte bis `rule=6` für 61 Punkte, den voreingestellten Wert);
- eine adaptive Unterteilung zu verlangen (Festlegung voreingestellt) oder eine direkte Anwendung des Verfahrens ohne Zusätze auf dem Integrationsbereich (durch Hinzufügung von `algorithm='qng'`);
- die Anzahl der Aufrufe des Integranden zu begrenzen.

Wird GSL an einer adaptiven Integration gehindert, kann das zu einem Verlust an Genauigkeit führen:

```
sage: numerical_integral(exp(-x^100), 0, 1.1)
(0.994325851191501, 4.077573041369464e-09)
sage: numerical_integral(exp(-x^100), 0, 1.1, algorithm='qng')
(0.9943275385765318, 0.016840666914688864)
```

Findet die Funktion `integrate` den analytischen Ausdruck nicht, der dem geforderten Integral entspricht, gibt sie den symbolischen Ausdruck unverändert zurück:

```
sage: integrate(exp(-x^2)*log(x), x, 17, 42)
integrate(e^(-x^2)*log(x), x, 17, 42)
```

und die numerische Rechnung, die mit `N` angestoßen wird, verwendet `numerical_integral`. Dies erklärt insbesondere, weshalb der Genauigkeitsparameter in diesem Fall ignoriert wird:

```
sage: N(integrate(exp(-x^2)*log(x), x, 17, 42), 200)
2.5657285006962035e-127
```

doch wir bekommen

```
sage: N(integrate(sin(x)*exp(cos(x)), x, 0, pi), 200)
2.3504023872876029137647637011912016303114359626681917404591
```

denn hier ist die symbolische Integration möglich.

`sage.calculus.calculus.nintegral`. Bei den symbolisch definierten Funktionen ist es möglich, von Maxima zu verlangen, ein Integral numerisch zu berechnen:

```
sage: sage.calculus.calculus.nintegral(sin(sin(x)), x, 0, 1)
(0.4306061031206906, 4.78068810228705e-15, 21, 0)
```

doch ist es ebenso möglich, die Methode `integral` bei einem Objekt des Typs `Expression` direkt aufzurufen:

```
sage: g(x) = sin(sin(x))
sage: g.nintegral(x, 0, 1)
(0.4306061031206906, 4.78068810228705e-15, 21, 0)
```

Maxima setzt die numerische Integrationsbibliothek QUADPACK ein, die wie GSL auf die Fließpunktzahlen der Maschine beschränkt ist. Die Funktion `nintegral` verwendet eine Strategie der adaptiven Unterteilung des Integrationsbereiches und wir können präzisieren:

- die für die Ausgabe gewünschte Genauigkeit;

- die maximale Anzahl der Teilintervalle, die bei der Rechnung betrachtet werden.

Die Ausgabe ist ein Tupel:

1. der Näherungswert des Integrals,
2. eine Abschätzung des absoluten Fehlers,
3. die Anzahl der Aufrufe des Integranden,
4. ein Fehlercode (0, wenn kein Problem aufgetreten ist; wegen der anderen Werte können Sie mit `sage.calculus.calculus.nintegralk?` die Dokumentation befragen).

`gp('intnum(...)')`. In Sage steht das Programm PARI zur Verfügung und enthält auch den Befehl `intnum` zur numerischen Integration von Funktionen:

```
sage: gp('intnum(x=17, 42, exp(-x^2)*log(x))')
2.5657285005610514829176211363206621657 E-127
```

Die Funktion `intnum` arbeitet nach dem doppelt exponentiellen Verfahren.

Es ist möglich, die globale Genauigkeit des Interpreters PARI und damit die Genauigkeit des Ergebnisses festzulegen:

```
sage: gp('intnum(x=0, 1, sin(sin(x)))')
0.43060610312069060491237735524846578643
sage: old_prec = gp.set_precision(50)
sage: gp('intnum(x=0, 1, sin(sin(x)))')
0.43060610312069060491237735524846578643360804182200
```

Ein größerer Nachteil dieser Methode ist, dass die zu integrierende Funktion wegen der Syntax von PARI als Zeichenkette angegeben werden muss. Deshalb kann nicht jede beliebige Funktion integriert werden.

Die Funktion `intnum` gestattet die Angabe des Verhaltens des Integranden an den Grenzen des Integrationsbereiches. Das folgende Beispiel illustriert den Einfluss, den das auf die Genauigkeit des Ergebnisses haben kann. Integrieren wir $x \mapsto x^{-1/2}$ zunächst ohne diese Angabe:

```
sage: p = gp.set_precision(old_prec) # die voreingestellte Genauigkeit
sage: gp('intnum(x=0, 1, x^(-99/100))')
73.62914262423378369
```

Mit Angabe der Art der Singularität, dass sich nämlich die Funktion bei $x = 0$ wie $x \mapsto x^{-99/100}$ verhält, kommt richtig

```
sage: gp('intnum(x=[0, -99/100], 1, x^(-99/100))')
100.00000000000000000000000000000000000000
```

Der Anwender ist für die Richtigkeit der Angabe des Verhaltens verantwortlich; wird irrtümlich beispielsweise gesagt, dass sich die Funktion bei $x = 0$ wie $x \mapsto x^{-1/42}$ verhält, wird der Fehler erheblich bleiben:

```
sage: gp('intnum(x=[0, -1/42], 1, x^(-99/100))')
74.47274932028288503
```

mpmath.quad*. Die Bibliothek `mpmath` ist eine in Python geschriebene Bibliothek für beliebig genaues numerisches Rechnen. Sie kann mit reellen und komplexen Fließpunktzahlen rechnen, mit Matrizen und Intervallen von reellen Fließpunktzahlen.

Sie verfügt über Funktionen zur numerischen Integration (die wichtigste ist `quad`) und kann in Sage importiert werden:

```
sage: import mpmath
sage: mpmath.mp.prec = 53
sage: mpmath.quad(lambda x: mpmath.sin(mpmath.sin(x)), [0, 1])
mpf('0.43060610312069059')
```

Es gibt keine Garantie, dass der Ausgabewert die geforderte Genauigkeit aufweist, wie wir anhand des folgenden Beispiels sehen können:

```
sage: mpmath.mp.prec = 113
sage: mpmath.quad(lambda x: mpmath.sin(mpmath.sin(x)), [0, 1])
mpf('0.43060610312069060491237735524846580 9')
```

```
sage: mpmath.mp.prec = 114
sage: mpmath.quad(lambda x: mpmath.sin(mpmath.sin(x)), [0, 1])
mpf('0.43060610312069060491237735524846578 5')
```

Es ist möglich, die geforderte Genauigkeit als Anzahl der Dezimalstellen anzugeben (`mpmath.mp.dps`) oder als Anzahl der Bits (`mpmath.mp.prec`). Zwecks Übereinstimmung mit der Funktion `N` von Sage verwenden wir nur die Genauigkeit in Bits.

Die Funktion `mpmath.quad` kann entweder das Verfahren von Gauß-Legendre aufrufen oder die doppelt exponentielle Transformation (letztere ist voreingestellt). Mit den Funktionen `mpmath.quadg1` und `mpmath.wuadts`[2] kann die zu verwendende Methode direkt angegeben werden.

Eine wichtige Einschränkung bei der Benutzung der Funktionen von `mpmath` zur Integration ist, dass sie nicht mit beliebigen in Sage definerten Funktionen zurechtkommen:

```
sage: mpmath.quad(sin(sin(x)), [0, 1])
Traceback (most recent call last):
...
TypeError: no canonical coercion from
<type 'sage.libs.mpmath.ext_main.mpf'> to Symbolic Ring
```

Die Situation ist jedoch nicht so problematisch wie bei PARI, welches auf eine Interaktion in Textform angewiesen ist. Tatsächlich können nämlich für Auswertung und Konversion nötige Funktionen hinzugefügt werden, um beliebige Funktionen mit `mpmath.quad` integrieren zu können[3]:

```
sage: g(x) = max_symbolic(sin(x), cos(x))
sage: mpmath.mp.prec = 100
```

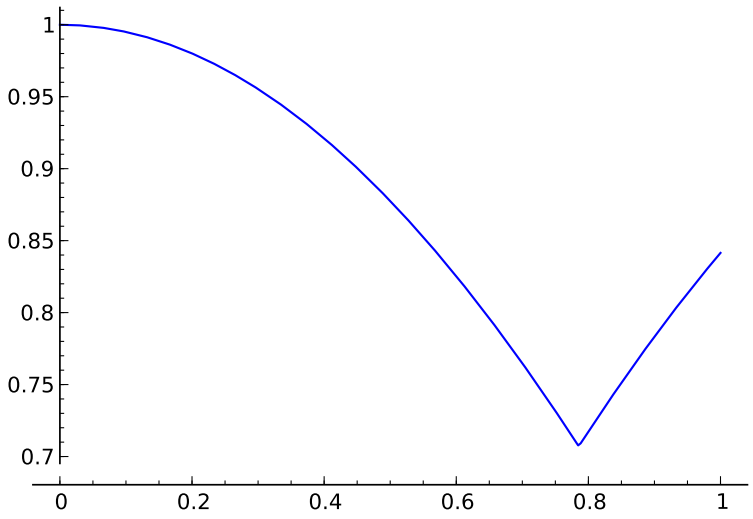


Abb. 14.5 - Die Funktion $x \mapsto \max(\sin(x), \cos(x))$. Die Irregularität bei $\pi/4$ führt zu einer sehr problematischen Integration

[2] Die verwendete Transformation $\varphi : t \mapsto \tanh\left(\frac{\pi}{2}\sinh(t)\right)$ wurde schon vorgestellt.

[3] Der Leser, der wissen möchte, weshalb wir `max_symbolic` verwendet haben, wird es mit `max` probieren und dann die Dokumentation von `max_symbolic` konsultieren.

```
sage: mpmath.quadts(lambda x: g(N(x, 100)), [0, 1])
mpf('0.87391241626303543595797908625 2')
```

Wir stellen fest, dass die Integration von nichtregulären Funktionen (wie das Beispiel I_4 weiter oben) einen Genauigkeitsverlust zur Folge haben kann, selbst dann, wenn wir ein Ergebnis mit hoher Genauigkeit forden:

```
sage: mpmath.mp.prec = 170
sage: mpmath.quadts(lambda x: g(N(x, 190)), [0, 1])
mpf('0.873910907574009752053930059819624763440541483541887 94')
sage: N(sqrt(2) - cos(1), 100)
0.87391125650495533140075211677
```

Nur fünf Stellen sind hier korrekt. Man kann `mpmath` dennoch helfen, indem eine Unterteilung des Integrationsbereiches vorgenommen wird (hier im irregulären Punkt, siehe Abb. 14.5):

```
sage: mpmath.quadts(lambda x: g(N(x, 170)), [0, mpmath.pi / 4, 1])
mpf('0.873911256504955331400752116766721474837361454759025 51')
```

Nichtstetige Funktionen stellen für die Integrationsverfahren eine klassische „Falle“ dar. Trotzdem kann eine Strategie der automatischen Anpassung durch Unterteilung den Schaden begrenzen.

Übung 47 *(Berechnung der Koeffizienten von Newton-Cotes).* Wir versuchen, die Koeffizienten des Verfahrens von Newton-Cotes in n Punkten zu berechnen, das in Sage nicht verfügbar ist. Zur Vereinfachung überlegen wir, dass der Integrationsbereich $I = [0, n-1]$ ist, demnach die Integrationspunkte $x_1 = 0$, $x_2 = 1, \ldots$, $x_n = n-1$. Die Koeffizienten (w_i) des Verfahrens sind derart, dass die Gleichung

$$\int_0^{n-1} f(x)\mathrm{d}x = \sum_{i=0}^{n-1} w_i f(i) \tag{14.1}$$

für jedes Polynom f höchstens $n-1$-ten Grades exakt ist.

1. Wir betrachten für $i \in \{0, \ldots, n-1\}$ das Polynom $P_i(X) = \prod_{j=0, j\neq i}^{n-1} (X - x_j)$. Die Gleichung (14.1) ist auf P_i anzuwenden, um w_i als Funktion von P_i zu erhalten.
2. Daraus ist eine Funktion `NCRule` herzuleiten, welche den n Punkten auf dem Intervall $[0, n-1]$ die Koeffizienten des Verfahrens von Newton-Cotes zuordnet.
3. Zeigen Sie, wie diese Gewichte auf einen beliebigen Integrationsbereich $[a, b]$ anzuwenden sind.
4. Schreiben Sie eine Funktion `QuadNC`, die das Integral einer Funktion auf einer Teilmenge von $\mathbb{R}$ berechnet, die durch einen Parameter gegeben ist. Vergleichen Sie das Resultat mit den Ergebnissen der in Sage zur Verfügung stehenden Funktionen für die Integrale I_1 bis I_{10}.

14.2. Numerische Lösung gewöhnlicher Differentialgleichungen

Wir interessieren uns in diesem Abschnitt für die numerische Lösung von gewöhnlichen Differentialgleichungen. Die in Sage für die Lösung verfügbaren Funktionen sind in der Lage, Gleichungssysteme der Form

$$\begin{cases} \frac{dy_1}{dt}(t) &= f_1(t, y_1(t), y_2(t), \ldots, y_n(t)) \\ \frac{dy_2}{dt}(t) &= f_2(t, y_1(t), y_2(t), \ldots, y_n(t)) \\ &\vdots \\ \frac{dy_n}{dt}(t) &= f_n(t, y_1(t), y_2(t), \ldots, y_n(t)) \end{cases}$$

mit den Anfangsbedingungen $y_1(0), y_2(0), \ldots, y_n(0)$ zu behandeln.

Diese Formeln machen es möglich, dass wir uns auch für Probleme mit Ordnungen über 1 interessieren, indem wir zusätzliche Variablen einführen (siehe das in 14.2.1 entwickelte Beispiel). Sie gestatten jedoch nicht, das durch die ρ-Funktion von Dickman

$$\begin{cases} u\rho'(u) + \rho(u-1) &= 0 \quad \text{für } u \geq 1, \\ \rho(u) &= 1 \quad \text{für } 0 \leq u \leq 1 \end{cases}$$

beschriebene Gleichungssystem auszudrücken. Die Werkzeuge zur Lösung von gewöhnlichen Differentialgleichungen sind an eine solche Gleichung nämlich nicht angepasst (sog. *retardierte* Differentialgleichung).

Die sog. „Einschritt-Verfahren“ zur numerischen Lösung machen alle von demselben allgemeinen Prinzip Gebrauch: für einen Schritt h und bekannte Werte von $y(t_0)$ und $y'(t_0)$ berechnet man einen Näherungswert von $y(t_0+h)$ und geht dabei von einem Schätzwert für $y'(t)$ aus, der auf dem Intervall $[t_0, t_0+h]$ genommen wird. Beispielsweise besteht das einfachste Verfahren in der Näherung

$$\begin{aligned} \forall t \in [t_0, t_0+h], \quad y'(t) &\approx y'(t0), \\ \int_{t_0}^{t_0+h} y'(t)dt &\approx hy'(t_0), \\ y(t_0+h) &\approx y(t_0) + hy'(t_0). \end{aligned}$$

Wenn hier y' durch eine konstante Funktion angenähert wird, erinnert das an das numerische Integrationsverfahren mit Rechtecken. Das vorliegende Verfahren ist von 1. Ordnung, d.h. dass der nach einem Rechenschritt erhaltene Fehler unter der Annahme, dass f hinnreichend gleichmäßig verläuft, im Bereich $\mathcal{O}(h^2)$ liegt. Allgemein ist ein Verfahren von der Ordnung p, wenn der bei einem Schritt erhaltene Fehler im Bereich $\mathcal{O}(h^{p+1})$ bleibt. Der für $t_1 = t_0 + h$ erhaltene Wert dient dann als Ausgangspunkt für den nächsten Schritt und so lange weiter wie gewünscht.

Dieses nach Euler benannte Verfahren 1. Ordnung ist nicht wegen seiner Genauigkeit berühmt (genauso wie die Integration mit Rechtecken), und es gibt Verfahren höherer Ordnung, beispielsweise das Verfahren von Runge-Kutta 2. Ordnung zur Lösung der Gleichung

$y' = f(t, y)$:

$$\begin{aligned} k_1 &= hf(t_n, y(t_n)) \\ k_2 &= hf(t_n + \frac{1}{2}h, y(t_n) + \frac{1}{2}k_1) \\ y(t_{n+1}) &\approx y(t_n) + k_2 + \mathcal{O}(h^3) \end{aligned}$$

Bei diesem Verfahren wird versucht, $y'(t_n + h/2)$ auszuwerten, um einen besseren Schätzwert für $y(t_n + h)$ zu erhalten.

Man findet auch Mehrschritt-Verfahren (zum Beispiel die Verfahren von Gear), die darin bestehen, $y(t_n)$ ausgehend von schon für l Schritte erhaltenen Werten $(y(t_{n-1}), y(t_{n-2}), \ldots, y(t_{n-L}))$ zu berechnen. Diese Verfahren starten notwendigerweise mit einer Initialierungsphase bis eine ausreichende Zahl von Schritten ausgeführt ist.

Genau wie das Verfahren zur numerischen Integration von Gauß-Kronrod existieren hybride Verfahren zur Lösung von Differentialgleichungen. So berechnet das Verfahren von Dormand und Prince mit den gleichen Stützstellen einen Wert 4. und 5. Ordnung, wobei letztere zur Abschätzung des Fehlers der ersteren dient. Wir sprechen hier von einem adaptiven Verfahren.

Wir unterscheiden noch die sogenannten expliziten Verfahren von den impliziten: bei einem expliziten Verfahren ist der Wert $y(t_{n+1})$ durch eine Formel gegegeben, die nur bekannte Werte benutzt; bei einem impliziten Verfahren muss eine Gleichung gelöst werden. Nehmen wir als Beispiel das implizite Euler-Verfahren:

$$y(t_{n-1}) = y(t_n) + hf(t_{n+1}, y(t_{n+1})).$$

Wir stellen fest, dass der gesuchte Wert $y(t_{n+1})$ auf beiden Seiten der Gleichung vorkommt; ist die Funktion f hinreichend komplex, muss ein nichtlineares algebraisches Systen gelöst werden, typischerweise mit dem Newton-Verfahren (siehe Unterabschnitt 12.2.2).

A priori erwarten wir umso genauere Ergebnisse zu bekommen, je kleiner wir den Integrationsschritt h machen; außer dem Aufwand für zusätzliche Rechnungen, den das kostet, wird der erhoffte Gewinn an Genauigkeit teilweise wieder aufgezehrt durch eine größere Anzahl von Rundungsfehlern, die das Ergebnis auf lange Sicht belasten.

14.2.1. Beispiellösung

Betrachten wir den van-der-Pol-Oszillator mit dem Parameter μ, der der folgenden Differentialgleichung genügt:

$$\frac{\mathrm{d}^2x}{\mathrm{d}t^2}(t) - \mu(1 - x^2)\frac{\mathrm{d}x}{\mathrm{d}t}(t) + x(t) = 0.$$

Setzen wir $y_0(t) = x(t)$ und $y_1(t) = \frac{\mathrm{d}x}{\mathrm{d}t}$, erhalten wir dieses System 1. Ordnung:

$$\begin{cases} \dfrac{\mathrm{d}y_0}{\mathrm{d}t} = y_1 \\ \dfrac{\mathrm{d}y_1}{\mathrm{d}t} = \mu(1 - y_0^2)y_1 - y_0 \end{cases}$$

Um es zu lösen, werden wir ein „Löser“-Objekt verwenden, das wir mit dem Befehl `ode_solver` bekommen:

```
sage: T = ode_solver()
```

Ein Löser-Objekt dient zur Aufnahme der Parameter und zur Definition des zu lösenden Systems; es verschafft Zugriff auf die in der Bibliothek GSL vorhandenen und im Zusammenhang mit der numerischen Integration bereits erwähnten Funktionen zur numerischen Lösung von Differentialgleichungen.

Die Gleichungen des Systems werden als Funktion angegeben:

```
sage: def f_1(t, y, params): return [y[1],params[0]*(1-y[0]^2)*y[1]-y[0]]
sage: T.function = f_1
```

Der Parameter y steht für den Vektor der unbekannten Funktionen und wir sollten den Vektor der rechten Seite des System als Funktion von t und eines optionalen Parameters (hier `params[0]`, der für μ steht) zurückgeben.

Bestimmte Algorithmen von GSL benötigen zusätzlich die Jacobi-Matrix des Systems (die Matrix mit den Einträgen $\frac{\partial f_i}{\partial y_j}$ an der Stelle (i, j) und deren letzte Zeile $\frac{\partial f_j}{\partial t}$ enthält):

```
sage: def j_1(t, y, params):
....:     return [[0,1],
....:             [-2*params[0]*y[0]*y[1]-1, params[0]*(1-y[0]^2)],
....:             [0,0]]
sage: T.jacobian = j_1
```

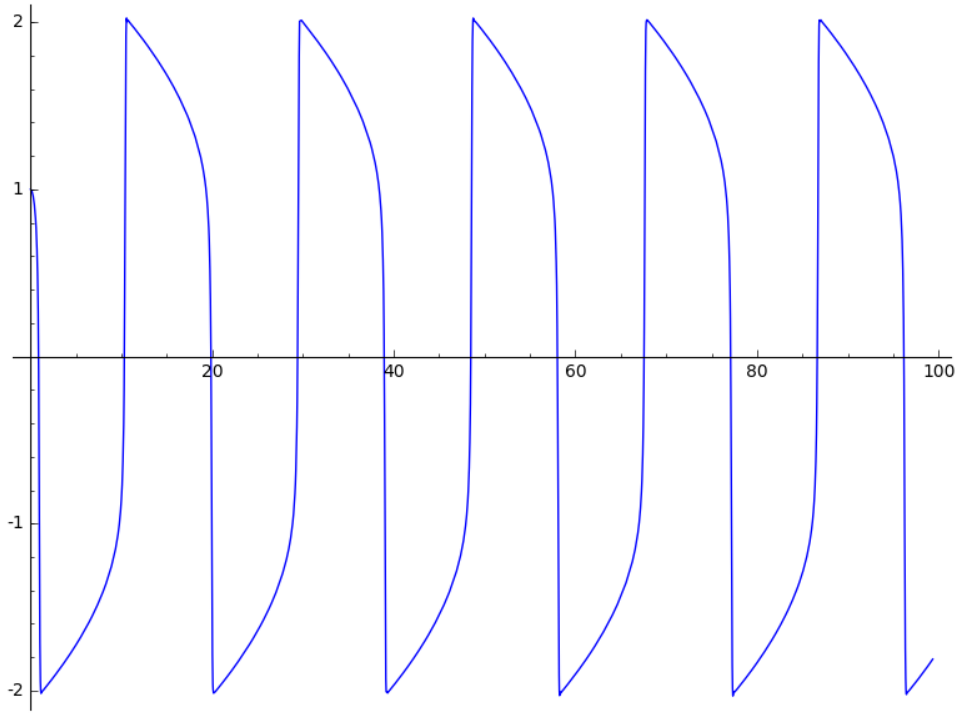
Nun kann eine numerische Lösung verlangt werden. Wir legen den Algorithmus und das Intervall fest, auf dem die Lösung zu berechnen ist und die Anzahl der gewünschten Schritte (wodurch h bestimmt wird):

```
sage: T.algorithm = "rk8pd"
sage: T.ode_solve(y_0=[1,0], t_span=[0,100], params=[10],
....:             num_points=1000)
sage: f = T.interpolate_solution()
```

Hier haben wir für die Lösung auf $[0, 100]$ den Algorithmus von Runge-Kutta Dormand-Prince genommen; die Anfangsbedingungen und die Werte der Parameter (hier nur einer) sind ebenfalls gegeben: hier steht `y_0=[1,0]` für $y_0(0) = 1$, $y_1(0) = 0$, d.h. $x(0) = 1$, $x'(0) = 0$.

Der Graph der Lösung zeigt (mit `plot(f, 0, 2)`), dass die Ableitung bei $t = 0$ gleich null ist),

```
sage: plot(f, 0, 100)
```



14.2.2. Verfügbare Funktionen

Unter den Löser-Objekten von GSL haben wir das Verfahren `rk8pd` bereits erwähnt. Weitere Verfahrenen sind:

`rkf45`: Runge-Kutta-Fehlberg, ein adaptives Verfahren 5. und 4. Ordnung,

`rk2`: adaptives Runge-Kutta 3. und 2. Ordnung,

`rk4`: das klassische Runge-Kutta-Verfahren 4. Ordnung,

`rk2imp`: ein implizites Runge-Kutta-Verfahren 2. Ordnung mit Auswertung in Intervallmitte,

`rk4imp`: eim implizites Runge-Kutta-Verfahren 4. Ordnung mit Auswertung an den „Gauß-Punkten“,[4]

`bsimp`: das implizite Verfahren von Burlisch-Stoer,

`gear1`: das implizite Einschritt-Verfahren von Gear,

`gear2`: das implizite Zweischritt-Verfahren von Gear.

Den an Einzelheiten zu diesen Verfahren interessierten Leser verweisen wir auf [AP98] oder [CM84].

Es ist anzumerken, dass die Beschränkung von GSL auf Maschinenzahlen - und damit auf eine feste Genauigkeit -, die bei der numerischen Integration schon angesprochen worden ist, auch für die Lösungsverfahren von Differentialgleichungen gültig bleibt.

[4] Es handelt sich um die Wurzeln des Legendre-Polynoms 2. Grades auf dem Intervall $[t, t+h]$. Sie sind nach dem Integrationsverfahren von Gauß-Legendre benannt.

Maxima stellt mit seiner eigenen Syntax ebenfalls numerische Routinen für die Lösung bereit:

```
sage: t, y = var('t, y')
sage: desolve_rk4(t*y*(2-y), y, ics=[0,1], end_points=[0, 1], step=0.5)
[[0, 1], [0.5, 1.12419127424558], [1.0, 1.461590162288825]]
```

Die Funktion `desolve_rk4` benutzt das Runge-Kutta-Verfahren 4. Ordnung (genau wie `rk4` in GSL) und erhält als Parameter

- die rechte Seite der Gleichung $y'(t) = f(t, y(t))$, hier $y' = ty(2 - y)$;
- den Namen der unbekannten Funktionsvariablen, hier y;
- die Anfangsbedingungen `ics`, hier $t = 0$ und $y = 1$;
- das Intervall für die Lösung `end_points`, hier $[0, 1]$;
- die Schrittweite `step`, hier 0.5.

Wir besprechen hier nicht den ähnlichen Befehl `desolve_system_rk4`, der schon im 4. Kapitel erwähnt wurde und der auf ein System von Differentialgleichungen angewendet wird. Auch Maxima wird durch die Maschinengenauigkeit begrenzt.

Sucht man Lösungen, die mit beliebig großer Genauigkeit berechnet werden, kann man sich `odefun` aus dem Modul `mpmath` zuwenden.

```
sage: import mpmath
sage: mpmath.mp.prec = 53
sage: sol = mpmath.odefun(lambda t, y: y, 0, 1)
sage: sol(1)
mpf('2.7182818284590451')
sage: mpmath.mp.prec = 100
sage: sol(1)
mpf('2.7182818284590452353602874802307')
sage: N(exp(1), 100)
2.7182818284590452353602874714
```

Die Argumente der Funktion `mpmath.odefun` sind:

- die rechte Seite des Gleichungssystems als Funktion $(t, y) \mapsto f(t, y(t))$, hier $y' = y$ wie bei der Funktion `ode_solver`. Die Dimension des Systems wird automatisch aus der Dimension des Rückgabewertes der Funktion hergeleitet;
- die Anfangsbedingungen t_0 und $y(t_0)$, hier $y(0) = 1$.

Beispielsweise ist für das System der Dimension 2

$$\begin{cases} y_1' = -y_2 \\ y_2' = y_1 \end{cases}$$

mit den Lösungen $(\cos(t), \sin(t))$ und den Anfangsbedingungen $y_1(0) = 1$ und $y_2(0) = 0$:

```
sage: mpmath.mp.prec = 53
sage: f = mpmath.odefun(lambda t, y: [-y[1], y[0]], 0, [1, 0])
sage: f(3)
```

```
[mpf('-0.98999249660044542'), mpf('0.14112000805986721')]
sage: (cos(3.), sin(3.))
(-0.989992496600445, 0.141120008059867)
```

Die Funktion `mpmath.odefun` wendet das Taylorverfahren an. Für einen Grad vom p nimmt man

$$y(t_{n+1}) = y(t_n) + h\frac{\mathrm{d}y}{\mathrm{d}t}(t_n) + \frac{h^2}{2!}\frac{\mathrm{d}^2y}{\mathrm{d}t^2}(t_n) + \ldots + \frac{h^p}{p!}\frac{\mathrm{d}^py}{\mathrm{d}t^p}(t_n) + \mathcal{O}(h^{p+1}).$$

Die Frage nach den Ableitungen ist die wichtigste. Dafür berechnet `odefun` die Näherungswerte

$$[\widetilde{y}(t_n + h), \ldots, \widetilde{y}(t_n + ph)] \approx [y(t_n + h), \ldots, y(t_n + ph)]$$

mit p Schritten des weniger genauen Euler-Verfahrens. Daraufhin berechnen wir

$$\widetilde{\frac{\mathrm{d}y}{\mathrm{d}t}}(t_n) \approx \frac{\widetilde{y}(t_n + h) - \widetilde{y}(t_n)}{h}, \quad \widetilde{\frac{\mathrm{d}y}{\mathrm{d}t}}(t_n + h) \approx \frac{\widetilde{y}(t_n + 2h) - \widetilde{y}(t_n + h)}{h}$$

und dann

$$\widetilde{\frac{\mathrm{d}^2y}{\mathrm{d}t^2}}(t_n) \approx \frac{\widetilde{\frac{\mathrm{d}y}{\mathrm{d}t}}(t_n + h) - \widetilde{\frac{\mathrm{d}y}{\mathrm{d}t}}(t_n)}{h},$$

und so fort bis man die Schätzwerte der Ableitungen von $y(t_n)$ bis zur Ordnung p erhält.

Wir müssen aufpassen, sobald die Fließpunkt-Genauigkeit von `mpmath` verändert wird. Um dieses Problem zu illustrieren, greifen wir die Lösung der Differentialgleichung $y' = y$ wieder auf, die mit der weiter oben gegebenen Funktion `exp` verifiziert wird:

```
sage: mpmath.mp.prec = 10
sage: sol = mpmath.odefun(lambda t, y: y, 0, 1)
sage: sol(1)
mpf('2.7148')
sage: mpmath.mp.prec = 100
sage: :sol(1)
mpf('2.7135204235459511323824699502438')
```

Die Näherung von exp(1) ist sehr schlecht, und das, obwohl mit einer Genauigkeit von 100 Bits gerechnet wurde! Die Lösung `sol` (ein „Interpolant“ im Jargon von `mpmath`) ist mit nur 10 Bits Genauigkeit berechnet worden und ihre Koeffizienten sind nach der Änderung der Genauigkeit nicht neu berechnet worden, was das Ergebnis erlärt.

Teil IV.

Kombinatorik

15. Abzählende Kombinatorik

Dieses Kapitel spricht haupsächlich die Behandlung der folgenden kombinatorischen Probleme mit Sage an: das Abzählen (wieviele Elemente gibt es in einer Menge S?), die Angabe (Berechnen aller Elemente von S oder die Iteration über sie), das zufäliige Ziehen (Zufallswahl eines Elements aus S gemäß einer Vorschrift, z.B. Gleichverteilung) . Diese Fragen stellen sich natürlich bei der Berechnung der Wahrscheinlichkeiten (wie groß ist die Wahrscheinlichkeit beim Poker einen Straight Flush oder vier Asse zu erhalten?), in der statistischen Physik, aber auch beim symbolischen Rechnen (Anzahl der Elemente eines endlichen Körpers) oder bei der algorithmischen Analysis. Die Kombinatorik deckt ein viel weiteres Gebiet ab (partielle Ordnungen, Theorie der Abbildungen usw.). Wir begnügen uns mit Hinweisen auf von Sage angebotenen Möglichkeiten. Graphen werden in Kapiel 16 behandelt.

Ein Charakteristikum rechnender Kombinatorik ist die Fülle an Typen von Objekten und Mengen, die man bearbeiten möchte. Es wäre nicht möglich, sie alle zu beschreiben oder gar zu implementieren. Nach einigen Beispielen (Abschnitt 15.1) veranschaulicht dieses Kapitel die zugrunde liegende Methodik: Bausteine zum Fundament zu liefern, um die üblichen kombinatorischen Mengen zu beschreiben (in Abschnitt 15.2), um Werkzeuge zu ihrer Kombination zwecks Bildung neuer Mengen zu liefern (in Abschnitt 15.3) und auch generische Algorithmen, mit denen eine große Klasse von Aufgaben behandelt werden kann (in Abschnitt 15.4). Zunächst kann dieses Kapitel quer gelesen werden, die Zusammenfassungen von Unterabschnitt 15.1.2 und Abschnitt 15.3 aber schon genauer.

Das ist ein Gebiet, wo Sage mehr Funktionalitäten hat als die meisten anderen Systeme zum symbolischen Rechnen und in voller Entwicklung begriffen ist; gleichzeitig ist es immer noch sehr jung mit vielen unnötigen Brüchen und Einschränkungen.

15.1. Erste Beispiele

15.1.1. Poker und Wahrscheinlichkeiten

Wir beginnen mit der Lösung eines klassischen Problems: des Abzählens bestimmter Kombinationen von Karten beim Poker, um deren Wahrscheinlichkeit abzuleiten.

Beim Poker ist eine Karte durch eine Farbe (Herz, Karo, Pik, Kreuz) und einen Wert (2, 3, ..., 10, Bube, Dame, König, As) charakterisiert. Das Spiel wird mit allen Karten gespielt, die möglich sind; es handelt sich um das cartesische Produkt der Menge der Farben und der Menge der Werte:

$$\text{Karten} = \text{Farben} \times \text{Werte} = \{(f, w) \mid f \in \text{Farben und } w \in \text{Werte}\}$$

Diese Mengen bilden wir in Sage:

```
sage: Farben = Set(['Karo','Herz','Pik','Kreuz'])
sage: Werte = Set([2..10] + ['Bube', 'Dame', 'Koenig', 'As'])
sage: Karten = cartesian_product([Farben, Werte])
```

Beim Poker gibt es 4 Farben und 13 mögliche Werte, also $4 \times 13 = 52$ Karten.

```
sage: Farben.cardinality()
4
sage: Werte.cardinality()
verb|13|
sage: Karten.cardinality()
52
```

Wir ziehen eine Karte:

```
sage: Karten.random_element()
['Kreuz', 6]
```

Nun ziehen wir zwei Karten:

```
sage: Set([Karten.random_element(), Karten.random_element()])
{('Herz', 2), ('Pik', 4)}
```

Zurück zu unserem Vorhaben. Wir betrachten hier eine vereinfachte Form von Poker, wo jeder Spieler unmittelbar fünf Karten zieht, die sein Blatt oder seine *Hand* bilden. Alle Karten sind verschieden und die Reihenfolge, in der sie gezogen werden, spielt keine Rolle; eine Hand ist also eine Untermenge der Größe 5 der Menge aller Karten. Um eine Hand zu ziehen, beginnen wir mit der Bildung der Menge aller möglichen Hände, dann entnehmen wir daraus zufällig eine Hand.

```
sage: Haende = Subsets(Karten, 5)
sage: Haende.random_element()
{('Herz', 4), ('Karo', 9), ('Pik', 8), ('Kreuz', 9), ('Herz', 7)}
```

Die Gesamtzahl der Hände ist durch die Anzahl der Teilmengen der Größe 5 einer Menge der Größe 52, d.h. durch den Binomialkoeffizienten $\binom{52}{5}$ gegeben:

```
sage: binomial(52,5)
2598960
```

Man kann sich um das Berechnungsverfahren auch keine Sorgen machen und einfach nach der Größe der Menge der Hände fragen:

```
sage: haende.cardinality()
2598960
```

Der Rang oder die Wertigkeit einer Hand beim Poker hängt von der Art der Kombination dieser Karten ab. Eine dieser Kombinationen ist der *Flush*; das ist eine Hand mit lauter Karten derselben Farbe (grundsätzlich sind hier Straight Flush und Royal Flush auszuschließen; diese sind Gegenstand einer Übung weiter unten). Eine solche Hand ist also charakterisiert durch eine bestimmte Farbe von vier möglichen und durch fünf Werte von 13 möglichen. Wir bilden die Menge aller Flushes, um deren Anzahl zu berechnen:

```
sage: Flushes = cartesian_product([Subsets(Werte, 5), farben])
sage: Flushes.cardinality()
5148
```

Die Wahrscheinlichkeit, beim Ziehen einer Hand zufällig einen Flush zu bekommen, ist daher:

```
sage: Flushes.cardinality()/Haende.cardinality()
33/16660
```

oder ungefähr zwei Promille:

```
sage: 1000.0*Flushes.cardinality()/Haende.cardinality()
1.98079231692677
```

Machen wir eine kleine numerische Simulation. Die folgende Funktion prüft, ob eine gegebene Hand ein Flush ist:

```
sage: def is_Flush(Hand):
....:     return len(set(Farbe for (Farbe, Wert) in Hand)) == 1
```

Wir ziehen jetzt zufällig 10000 Hände und berechnen die Anzahl der erhaltenen Flushes (das dauert etwa 10 Sekunden):

```
sage: n = 10000; nFlushes = 0
sage: for i in range(n):
....:     Hand = Haende.random_element()
....:     if is_Flush(hand):
....:         nFlushes += 1
....:     print n, nFlushes
10000 18
```

Übung 48. Eine Hand mit vier Karten gleichen Wertes heißt Vierling. Bilden Sie eine Menge solcher Vierlinge. (Hinweis: verwenden Sie `Arrangements` zum zufälligen Ziehen eines Paares mit verschiedenen Werten, und wählen Sie dann eine Farbe für den ersten Wert). Berechnen Sie die Anzahl möglicher Vierlinge, listen Sie sie auf und bestimmen Sie anschließend die Wahrscheinlichkeit für das Ziehen einer Hand mit einem Vierling.

Übung 49. Eine Hand mit lauter Karten gleicher Farbe und aufeinander folgenden Werten ist ein Straight Flush oder ein Royal Flush, aber kein Flush. Berechnen Sie die Anzahl möglicher Straight Flushes und Royal Flushes und leiten Sie dann die korrekte Wahrscheinlichkeit dafür her, einen Flush zu erhalten, wenn eine Hand zufällig gezogen wird.

Übung 50. Berechnen Sie die Wahrscheinlichkeit jeder Kartenkombination beim Poker (siehe `https://de.wikipedia.org/wiki/Hand_(Poker)`) und vergleichen Sie sie mit den Ergebnissen von Simulationen.

15.1.2. Abzählen von Bäumen durch erzeugende Reihen

In diesem Unterabschnitt diskutieren wir das Beispiel vollständiger Binärbäume und veranschaulichen in diesem Zusammenhang viele Abzähltechniken, unter denen die formalen Potenzreihen eine natürliche Rolle spielen. Diese Techniken sind ziemlich allgemein und können immer dann angewendet werden, wenn das in Rede stehende kombinatorische Objekt eine rekursive Definition (rekursive Grammatik) zulässt (siehe Unterabschnitt 15.4.4 wegen einer automatisierten Behandlung). Das Ziel ist keine formale Darstellung dieser Verfahren; die Rechnungen sind streng, die Begründungen werden aber meistens weggelassen.

Ein *vollständiger Binärbaum* ist entweder ein Blatt `F` oder ein Knoten mit zwei Binärbäumen (siehe Abb. 15.1).

Übung 51. Suchen Sie ohne Rechnerhilfe alle vollständigen Binärbäume mit $n = 1, 2, 3, 4, 5$ Blättern (siehe auch Übung 59, um sie mit Sage zu finden.)

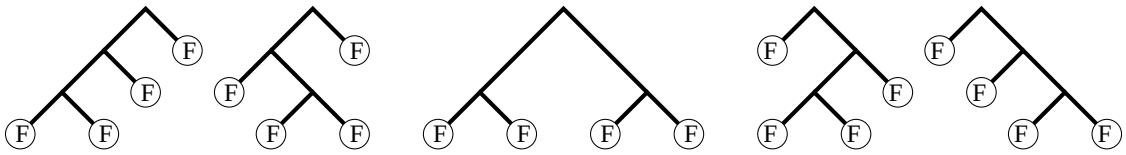


Abb.15.1 - Die fünf vollständigen Binärbäume mit vier Blättern.

Unser Ziel ist, die Anzahl c_n der vollständigen Binärbäume mit n Blättern zu suchen (in diesem Abschnitt steht „Bäume“ immer für vollständige Binärbäume, wenn nicht ausdrücklich etwas anderes gesagt ist). Das ist eine typische Situation, in der wir nicht nur an einer einzelnen Menge interessiert sind, sondern an einer Familie von Mengen, die typischerweise mit $n \in \mathbb{N}$ parametrisiert werden.

Entsprechend der Lösung von Übung 51 sind die ersten Terme durch $c_1, \ldots, c_5 = 1, 1, 2, 5, 14$ gegeben. Die einfache Tatsache diese Zahlen zu kennen, ist bereits sehr wertvoll. Sie erlaubt nämlich die Suche in einer Goldmine von Informationen: auf der Seite *Online Encyclopedia of Integer Sequences* `http://oeis.org/`, die nach ihrem wichtigsten Autor gemeinhin „Sloane“ genannt wird. Sie enthält mehr als 282892 ganzzahlige Folgen:

```
sage: oeis([1,1,2,5,14])
0: A000108: Catalan numbers: C(n) = binomial(2n,n)/(n+1) =
(2n)!/(n!(n+1)!). Also called Segner numbers.
1: A120588: G.f. satisfies: 3*A(x) = 2 + x + A(x)^2, with a(0) = 1.
2: A080937: Number of Catalan paths (nonnegative, starting and ending at
0, step +/-1) of 2*n steps with all values <= 5.
```

Das Ergebnis lässt erkennen, dass die Bäume mit einer der berühmtesten Folgen abgezählt werden, den Catalan-Zahlen. Bei Durchsicht der Ausgabe der Enzyklopädie sehen wir, das dies tatsächlich so ist: die wenigen eingebenen Zahlen bilden einen Fingerabdruck unserer Objekte, der uns in einer umfangreichen Literatur in wenigen Sekunden ein genaues Ergebnis finden lässt.

Abzählen mit erzeugenden Reihen. Unser Aufgabe ist es, dieses Resultat nun auch mit Sage zu finden. Sei C_n die Menge der Bäume mit n Blättern, also $c_n = |C_n|$. Wie üblich definieren wir $C_0 = \emptyset$ und $c_0 = 0$. Die Menge aller Bäume ist dann die disjunkte Vereinigung der Mengen C_n:

$$C = \biguplus_{n\in\mathbb{N}} C_n.$$

Nachdem wir die Menge C aller Bäume benannt haben, können wir die rekursive Definition in eine mengentheoretische Gleichung übersetzen:

$$C \approx \{L\} \quad \uplus \quad C \times C.$$

In Worten: ein Baum t (der definitionsgemäß in C liegt) ist entweder ein Blatt (also in $\{L\}$) oder ein Knoten mit den beiden Bäumen t_1 und t_2, die wir deshalb mit dem Paar (t_1, t_2) (im cartesischen Produkt $C \times C$) identifizieren können.

Die Gründungsidee der algebraischen Kombinatorik, die von Euler in einem Brief an Goldbach von 1751 zur Behandlung eines ähnlichen Problems eingeführt worden ist [Vie07], besteht darin, alle Zahlen c_n simultan zu behandeln, indem sie als Koeffizienten einer formalen Potenzreihe kodiert werden, welche die *erzeugende Funktion* der c_n genannt wird:

$$C(z) = \sum_{n\in\mathbb{N}} c_n z^n,$$

wobei z eine symbolische Unbestimmte ist (weshalb wir uns um die Frage der Konvergenz nicht sorgen müssen). Die Schönheit dieser Idee besteht darin, dass die Mengenoperationen ($A \uplus B$, $A \times B$) ganz natürlich in algebraische Operationen mit den Reihen ($A(z) + B(z)$, $A(z) \cdot B(z)$) übersetzt werden, und zwar so, dass die Mengengleichung, die von C erfüllt wird, in eine algebraische Gleichung mit $C(z)$ überführt wird:

$$C(z) = z + C(z) \cdot C(z).$$

Nun können wir diese Gleichung mit Sage lösen. Dazu führen wir die beiden Variablen C und z ein und definieren die Gleichung

```
sage: C, z = var('C, z'); sys = [C = z + C*C]
```

Es gibt zwei Lösungen, die zufällig geschlossene Formen haben:

```
sage: sol = solve(sys, C, solution_dict=True); sol
[{C: -1/2*sqrt(-4*z + 1) + 1/2}, {C: 1/2*sqrt(-4*z + 1) + 1/2}]
sage: s0 = sol[0][C]; s1 = sol[1][C]
```

und deren Taylor-Entwicklung beginnt mit

```
sage: s0.series(z, 6)
1*z + 1*z^2 + 2*z^3 + 5*z^4 + 14*z^5 + Order(z^6)
sage: s1.series(z,6)
1 + (-1)*z + (-1)*z^2 + (-2)*z^3 + (-5)*z^4 + (-14)*z^5 + Order(z^6)
```

Die zweite Lösung ist offensichtlich unbrauchbar, wohingegen die erste die erwarteten Koeffizienten liefert. Daher setzen wir

```
sage: C = s0
```

Wir können nun die nächsten Terme berechnen:

```
sage: C.series(z, 11)
1*z + 1*z^2 + 2*z^3 + 5*z^4 + 14*z^5 + 42*z^6 +
132*z^7 + 429*z^8 + 1430*z^9 + 4862*z^10 + Order(z^11)
```

oder im Nu den 100. Koeffizienten:

```
sage: C.series(z, 101).coefficient(z, 100)
22750883079422934966181954039568885395604168260154104734 0
```

Leider muss alles nochmals berechnet werden, falls wir einmal den 101.Koeffizienten brauchen. Faule Potenzreihen (siehe Unterabschnitt 7.5.3) machen hier Sinn, zumal wir sie direkt aus einem Gleichungssystem definieren können, ohne es zu lösen und auch deshalb, weil wir für die Antwort keine geschlossene Form brauchen. Wir beginnen mit der Definition von faulen Potenzreihen

L.<z> = LazyPowerSeriesRing(QQ)

Dann erzeugen wir eine „freie“ Potenzreihe, der wir einen Namen geben und die wir dann durch eine rekursive Gleichung definieren:

```
sage: C = L()
sage: C._name = 'C'
sage: C.define(z + C*C)

sage: [C.coefficient(i) for i in range(11)]
[0, 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862]
```

Wir können nun einen beliebigen Koeffizienten erfragen, ohne C neu definieren zu müssen:

```
sage: C.coefficient(100)
22750883079422934966181954039568885395604168260154104734 0

sage: C.coefficient(200)
12901315806442911400122290766967667513434953055272888249981085159890141 9\
01334831904553458085084773552827575012218894 0
```

Rekursion und geschlossene Form. Wir kommen nun auf die geschlossenen Form von $C(z)$ zurück:

```
sage: z = var('z'); C = s0; C
-1/2*sqrt(-4*z + 1) + 1/2
```

Der n-te Koeffizient der Taylorentwicklung von $C(z)$ ist durch $\frac{1}{n!}C^{(n)}(0)$ gegeben. Wir betrachten dazu die sukzessiven Ableitungen $C^{(n)}(z)$:

```
sage: derivative(C, z, 1)
1/sqrt(-4*z + 1)
sage: derivative(C, z, 2)
2/(-4*z + 1)^(3/2)
sage: derivative(C, z, 3)
12/(-4*z + 1)^(5/2)
```

Das lässt die Existenz einer einfachen expliziten Formel vermuten, nach der wir jetzt suchen werden. Die folgende kleine Funktion gibt $d_n = n!c_n$ zurück:

```
sage: def d(n): return derivative(C, n).subs(z=0)
```

Mit Verwendung sukzessiver Quotienten

```
sage: [ (d(n+1) / d(n)) for n in range(1,17) ]
[2, 6, 10, 14, 18, 22, 26, 30, 34, 38, 42, 46, 50, 54, 58, 62]
```

bemerken wir, dass d_n der rekursiven Bezehung $d_{n+1} = (4n-2)d_n$ genügt, aus der wir herleiten, dass c_n die Gleichung $c_{n+1} = \frac{4n-2}{n+1}c_n$ erfüllt. Wir vereinfachen und finden, dass c_n die $(n-1)$-te catalansche Zahl ist:

$$c_n = \text{Catalan}\,(n-1) = \frac{1}{n}\binom{2(n-1)}{n-1}.$$

Das prüfen wir:

```
sage: def c(n): return 1/n*binomial(2*(n-1),n-1)
sage: [c(k) for k in range(1, 11)]
[1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862]
sage: [catalan_number(k-1) for k in range(1, 11)]
[1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862]
```

Wir können nun auch die viel späteren Koeffizienten berechnen; hier berechnen wir c_{100000}, das mehr als 60000 Ziffern hat:

```
sage: time cc = c(100000)
Time: CPU 0.35 s, Wall: 0.37 s
sage: ZZ(cc).ndigits()
60198
```

Systematische Behandlung mit algebraischen Differentialgleichungen. Das Verfahren, das wir verwendet haben, wird auf alle rekursiv definierten Objekte verallgemeinert: die Gleichungssysteme für Mengen werden in Gleichungssysteme für die erzeugende Reihe übersetzt; das gestattet die rekursive Berechnung ihrer Koeffizienten. Wenn die Gleichungssysteme für Mengen hinreichend einfach sind (wenn beispielsweise nur cartesische Produkte und disjunkte Vereinigungen vorkommen), erhalten wir eine algebraische Gleichung in $C(z)$. Diese hat nicht immer eine Lösung in geschlossener Form; indessen können wir aus ihr durch Eingrenzung eine *lineare* Differentialgleichungen herleiten, die ihrerseits in eine Rekursionsgleichung fester Länge für die Koeffizienten c_n umgeformt wird (die Reihe wird nun *D-endlich* genannt). Schließlich geht die Berechnung der Koeffizienten nach diesen Vorberechnungen sehr schnell. Alle diese Schritte sind rein algorithmisch, und es ist vorgesehen, alle in Maple (die Pakete gfun und combstruct) oder MuPAD-Combinat (die Bibliothek decomposableObjects) vorhandenen Implementierungen nach Sage zu übertragen.

Für den Moment veranschaulichen wir das allgemeine Vorgehen für den Fall vollständiger Binärbäume. Die erzeugende Funktion $C(x)$ ist Lösung einer algebraischen Gleichung $P(z, C(z)) = 0$, wobei $P = P(x, y)$ ein Polynom mit Koeffizienten aus $\mathbb{Q}$ ist. Im vorliegenden Fall ist $P = y^2 - y + x$. Wir differenzieren diese Gleichung symbolisch nach z:

```
sage: x, y, z = var('x, y, z')
sage: P = function('P')(x, y); C = function('C')(z)
sage: equation = P(x=z, y=C) == 0
sage: diff(equation, z)
diff(C(z), z)*D[1](P)(z, C(z)) + D[0](P)(z, C(z)) == 0
```

oder in besser lesbarer Form

$$\frac{\mathrm{d}C(z)}{\mathrm{d}z}\frac{\partial P}{\mathrm{y}}(z,C(z)) + \frac{\partial P}{\partial x}(z,C(z)) = 0.$$

Daraus leiten wir her:

$$\frac{\mathrm{d}C(z)}{\mathrm{d}z} = -\frac{\frac{\partial P}{\partial x}}{\frac{\partial P}{\partial y}}(z,C(z)).$$

Bei vollständigen Binärbäumen ergibt das:

```
sage: P = y^2 - y + x; Px = diff(P, x); Py = diff(P, y)
sage: - Px / Py
-1/(2*y - 1)
```

Nun ist aber $P(x,y) = 0$. Wir können deshalb diesen Bruch modulo P berechnen und so die Ableitung von $C(z)$ als *Polynom in* $C(z)$ *mit Koeffizienten aus* $\mathbb{Q}(z)$. Dazu bilden wir den Quotientenring $R = \mathbb{Q}(x)[y]/(P)$:

```
sage: Qx = QQ['x'].fraction_field(); Qxy = Qx['y']
sage: R = Qxy.quo(P); R
Univariate Quotient Polynomial Ring in ybar
over Fraction Field of Univariate Polynomial Ring in x
over Rational Field with modulus y^2 - y + x
```

Bemerkung: `ybar` ist der Name der Variablen y im Quotienten; wegen weiterer Informationen zu Quotientenringen siehe Unterabschnitt 7.2.2. Wir setzen nun die Berechnung dieses Bruches in R fort

```
sage: fraction = - R(Px) / R(Py); fraction
(1/2/(x - 1/4))*ybar - 1/4/(x - 1/4)
```

Wir heben das Ergebnis nach $\mathbb{Q}(x)[y]$ und ersetzen dann z und $C(z)$, um einen Ausdrück für $\frac{\mathrm{d}}{\mathrm{d}z}C(z)$ zu bekommen:

```
sage: fraction = fraction.lift(); fraction
(1/2/(x - 1/4))*y - 1/4/(x - 1/4)
sage: fraction(x=z, y=C)
2*C(z)/(4*z - 1) - 1/(4*z - 1)
```

oder besser lesbar

$$\frac{\partial C(z)}{\partial z} = \frac{1}{1-4z} - \frac{2}{1-4z}C(z).$$

In diesem einfachen Fall können wir aus diesem Ausdruck eine lineare Differentialgleichung mit Koeffizienten aus $\mathbb{Q}[z]$ unmittelbar herleiten:

```
sage: equadiff = diff(C,z) == fraction(x=z, y=C); equadiff
diff(C(z), z) == 2*C(z)/(4*z - 1) - 1/(4*z - 1)
```

```
sage: equadiff = equadiff.simplify_rational()
sage: equadiff = equadiff * equadiff.rhs().denominator()
sage: equadiff = equadiff - equadiff.rhs()
sage: equadiff
(4*z - 1)*diff(C(z), z) - 2*C(z) + 1 == 0
```

oder besser lesbar

$$(1-4z)\frac{\partial C(z)}{\partial z} + 2C(z) - 1 = 0.$$

Es ist trivial, diese Gleichung in geschlossener Form zu verifizieren:

```
sage: Cf = sage.symbolic.function_factory.function('C')
sage: bool(equadiff.substitute_function(Cf, s0))
True
```

Im allgemeinen fahren wir mit der Berechnung der sukzessiven Ableitungen von $C(z)$ fort. Diese Ableitungen sind auf den Quotientenring $\mathbb{Q}(z)[C]/P$ beschränkt, der die endliche Dimension $\deg P$ auf $\mathbb{Q}(z)$ hat. Dafür finden wir zwischen den ersten $\deg P$ Ableitungen von $C(z)$ eventuell eine lineare Beziehung. Setzen wir das auf einen gemeinsamen Nenner, erhalten wir eine lineare Differentialgleichung vom Grad $\leq \deg P$ mit Koeffizienten aus $\mathbb{Q}[z]$. Durch Herausziehen des Koeffizienten von z^n in der Differentialgleichung bekommen wir die verlangte rekursive Beziehung zwischen den Koeffizienten. Hier treffen wir wieder auf die Beziehung, die wir - auf der geschlossenen Form basierend - breits gefunden hatten:

$$c_{n+1} = \frac{4n-2}{n+1} c_n$$

Nach Festlegung der korrekten Anfangsbedingungen wird es möglich, die Koeffizienten von $C(z)$ rekursiv zu ermitteln:

```
sage: def C(n): return n if n <= 1 else (4*n-6)/n * C(n-1)
sage: [ C(i) for i in range(10) ]
[0, 1, 1, 2, 5, 14, 42, 132, 429, 1430]
```

Wird n zu groß für eine explizite Berechnung von c_n kann eine asymptotisch äquivalente Folge von Koeffizienten c_n gesucht werden. Auch hier gibt es wieder generelle Techniken. Das zentrale Werkzeug ist komplexe Analysis, speziell die Untersuchung der erzeugenden Funktion in der Nähe ihrer Singularitäten. Bei der vorliegenden Aufgabe befindet sich die Singularität bei $z_0 = 1/4$ und wir bekämen $c_n \approx \frac{4^{n-1}}{n^{3/2}\sqrt{\pi}}$.

Zusammenfassung. Wir erkennen hier ein allgemeines Phänomen der Computeralgebra: die beste Datenstruktur zur Beschreibung eines komplizierten mathematischen Objektes (eine reelle Zahl, eine Folge, eine formale Potenzreihe, eine Funktion, eine Menge) ist oft eine Gleichung, die das Objekt definiert (oder ein Gleichungssystem, typischerweise mit Anfangsbedingungen). Eine Lösung in geschlossener Form zu finden zu versuchen, ist nicht immer interessant: einerseits existiert eine solche geschlossene Fom nur selten (z.B. bei der Lösung von Polynomen durch Radikale), und andererseits enthält die Gleichung alle für die algorithmische Berechnung der Eigenschaften des betrachteten Objektes notwendige Information in sich (z.B. eine numerische Näherung, die ersten Terme oder Elemente, ein asymptotisches Äquivalent) oder für die Rechnung mit dem Objekt selbst (z.B. Arithmetik mit Potenzreihen). Deshalb suchen wir nach der Gleichung, die das Objekt beschreibt und zu dem zu lösenden Problem am besten passt; siehe auch Unterabschnitt 2.2.2.

Wie wir an unserem Beispiel gesehen haben, ist Eingrenzung (beispielsweise in einem endlichdimensionalen Vektorraum) ein grundlegendes Werkzeug für die Untersuchung solcher Gleichungen. Der Begriff der Eingrenzung ist bei Eliminationsverfahren breit anwendbar (lineare Algebra, Gröbner-Basen und ihre Verallgemeinerungen bei algebraischen Differentialgleichungen). Dasselbe Werkzeug spielt bei der automatischen Summierung und der automatischen Prüfung von Identitäten eine zentrale Rolle (Algorithmen von Gosper oder Zeilberger und ihre Verallgemeinerungen [PWZ96]; siehe auch die Beispiele in Unterabschnitt 2.3.1 und in Übung 54).

Alle diese Techniken und ihre vielfältigen Verallgemeinerungen sind Gegenstand sehr intensiver Forschungen: automatische und analytischbe Kombinatorik mit größeren Anwendungen in der Analyse von Algorithmen [FS09]. Es ist wahrscheinlich und wünschenswert, dass sie nach und nach in Sage implementiert werden.

15.2. Die üblichen abzählbaren Mengen

15.2.1. Beispiel: Teilmengen einer Menge

Stellen wir uns eine Menge E der Größe n vor und betrachten ihre Teilmengen der Größe k. Wir wissen, dass diese Teilmengen durch Binomialkoeffizienten $\binom{n}{k}$ abgezählt werden. Wir können deshalb die Anzahl der Teilmengen der Größe $k = 2$ von $E = \{1, 2, 3, 4\}$ mit der Binomialfunktion berechnen:

```
sage: binomial(4, 2)
6
```

Alternativ können wir die Menge $\mathcal{P}_2(E)$ der Teilmengen der Größe 2 von E *konstruieren* und dann nach ihrer Kardinalität fragen;

```
sage: S = Subsets([1,2,3,4], 2); S.cardinality()
6
```

Liegt S einmal vor, können wir auch die Liste der Elemente bekommen, ein Element zufällig auswählen oder ein typisches Element verlangen.

```
sage: S.list()
[{1, 2}, {1, 3}, {1, 4}, {2, 3}, {2, 4}, {3, 4}]
sage: S.random_element()
{1, 4}
sage: S.an_element()
{2, 3}
```

Genauer, das Objekt S modelliert die Menge $\mathcal{P}_2(E)$, die mit einer festen Ordnung (hier die lexikographische Ordnung) ausgestattet ist. Daher ist es möglich, nach ihrem 5. Element zu fragen, wobei zu beachten ist, dass in Python das erste Element mit 0 indiziert ist: als Abkürzung kann man hier auch schreiben:

```
sage: S.unrank(4)
{2, 4}
sage: S[4]
{2, 4}
```

Das sollte aber mit Vorsicht gemacht werden, weil manche Mengen eine andere Indizierung haben als $(0, \ldots)$.

Umgekehrt können wir die Position eines Elementes in dieser Ordnung berechnen:

```
sage: s = S([2,4]); S.rank(s)
4
```

Beachten Sie, dass `S` *nicht* die Liste ihrer Elemente ist. Wir können beispielsweise die Menge $\mathcal{P}(\mathcal{P}(\mathcal{P}(E)))$ bilden und deren Kardinalität $2^{2^{2^4}}$ berechnen:

```
sage: E = Set([1,2,3,4])
sage: S = Subsets(Subsets(Subsets(E))); S.cardinality()
200352993040684646497907235156025575044782547556975141926501 6...736
```

was ungefähr $2 \cdot 10^{19728}$ ist:

```
sage: S.cardinality().ndigits()
19729
```

und nach dem 237102124. Element fragen:

```
sage: S.unrank(237102123)
{{{2, 4}, {1, 4}, {}, {1, 3, 4}, {1, 2, 4}, {4}, {2, 3}, {1, 3}, {2}},
{{1, 3}, {2, 4}, {1, 2, 4}, {}, {3, 4}}}
```

Physikalisch ist es unmöglich alle Elemente von `S` explizit hinzuschreiben, denn es gibt viel mehr davon als Partikel im ganzen Universum vorhanden sind (schätzungsweise 10^{82}).

Bemerkung: in Python wäre es nur natürlich, mit `len(S)` nach der Kardinalität von `S` zu fragen. Das geht hier aber nicht, weil Python verlangt, dass das Ergebnis von `len` eine Zahl vom Typ `int` zu sein hat; das könnte zu einem Überlauf führen und bei unendlichen Mengen auch nicht den Wert `Infinity` zurückgeben.

```
sage: len(S)
Traceback (most recent call last):
...
OverflowError: Python int too large to convert to C long
```

15.2.2. Partitionen natürlicher Zahlen

Wir nehmen uns jetzt ein anderes klassisches Problem vor: gegeben ist eine natürliche Zahl n; auf wieviele Arten kann n in Form einer Summe $n = i_1 + i_2 + \ldots + i_l$ geschrieben werden, wobei $i_1, \ldots, i_l$ natürliche Zahlen sind? Zwei Fälle sind zu unterscheiden:

- die Reihenfolge der Summanden ist unwichtig; dann nennen wir $n = i_1 + i_2 + \ldots + i_l$ eine *Partition* von n;
- auf die Reihenfolge der Summanden kommt es an; dann nennen wir die Summe eine *Komposition* von n.

Wir beginnen mit den Partitionen von $n = 5$; wie zuvor bilden wir zuerst die Menge dieser Partitionen:

```
sage: P5 = Partitions(5); P5
Partitions of the integer 5
```

dann fragen wir nach ihrer Kardinalität:

```
sage: P5.cardinality()
7
```

Wir sehen uns diese 7 Partitionen an; da es auf die Reihenfolge nicht ankommt, werden die Einträge vereinbarungsgemäß in absteigender Reihenfolge angeordnet.

```
sage: P5.list()
[[5], [4, 1], [3, 2], [3, 1, 1], [2, 2, 1], [2, 1, 1, 1], [1, 1, 1, 1, 1]]
```

Die Berechnung der Anzahl der Partitionen verwendet die Formel von Rademacher (siehe `https://de.wikipedia.org/wiki/Partitionsfunktion`), implementiert in C und bestens optimiert, was sie sehr schnell macht:

```
sage: Partitions(100000).cardinality()
27493510569775696512677516320986352688173429315980054758203125984302147328114964173055050741660736621590157844774296248940493063070200461792764493033510116079342457190155718943509725312466108452006369558934464248716828789832182345009262853831404597021307130674510624419227311238999702284408609370935531629697851569569892196108480158600569421098519
```

Partitionen natürlicher Zahlen sind Objekte der Kombinatorik, die mit vielen Operationen ausgestattet sind. Sie werden im Vergleich zu einfachen Listen als reichere Objekte zurückgegeben.

```
sage: P7 = Partitions(7); p = P7.unrank(5); p
[4, 2, 1]
```

```
sage: type(p)
<class 'sage.combinat.partition.Partitions_n_with_category.element_class'>
```

Zum Beispiel können sie graphisch als Ferrers-Diagramm dargestellt werden.

```
sage: print p.ferrers_diagram()
****
**
*
```

Wir überlassen es dem Leser, die verfügbaren Operationen durch Introspektion zu erkunden.

Wir können eine Partition auch direkt erzeugen mit

```
sage: Partition([4,2,1])
[4, 2, 1]
sage: P7([4,2,1])
[4, 2, 1]
```

Möchte man die möglichen Werte der Teile $i_1, \ldots, i_l$ beschränken, zum Beispiel beim Wechseln von Geld, kann man `WeightedIntegerValues` benutzen. Die folgende Rechnung beispielsweise

```
sage: WeightedIntegerVectors(8, [2,3,5]).list()
[[0, 1, 1], [1, 2, 0], [4, 0, 0]]
```

zeigt, dass 8 Dollar mit Banknoten zu 2$, 3$ und 5$ entweder in 3$ plus 5$ oder in 2$ plus 2 mal 3$ oder 4 mal 2$ getauscht werden können.

Kompositionen von natürlichen Zahlen werden ebenso behandelt:

```
sage: C5 = Compositions(5); C5
Compositions of 5 sage: C5.cardinality()
16
sage: C5.list()
[[1, 1, 1, 1, 1],  [1, 1, 1, 2], [1, 1, 2, 1], [1, 1, 3],
 [1, 2, 1, 1],  [1, 2, 2], [1, 3, 1], [1, 4], [2, 1, 1, 1],
 [2, 1, 2], [2, 2, 1], [2, 3], [3, 1, 1], [3, 2], [4, 1], [5]]
```

Die Anzahl 16 scheint bedeutsam zu sein und lässt die Existenz einer Formel vermuten. Wir schauen auf die Anzahl der Kompositionen von n im Bereich von 0 bis 9:

```
sage: [ Compositions(n).cardinality() for n in range(10) ]
[1, 1, 2, 4, 8, 16, 32, 64, 128, 256]
```

Wenn wir die Anzahl der Kompositionen von 5 hinsichtlicht der Länge betrachten, finden wir eine Zeile des pascalschen Dreiecks:

```
sage: x = var('x'); sum( x^len(c) for c in C5 )
x^5 + 4*x^4 + 6*x^3 + 4*x^2 + x
```

Das obige Beispiel benutzt eine Funktionalität, die wir bislang noch nicht gesehen haben: Da `C5` iterierbar ist, kann es wie eine Liste in einer `for`-Schleife oder einer Raffung[1]engl. comprehension) verwendet werden (Unterabschnitt 15.2.4).

Übung 52. Beweisen Sie die von obigen Beispielen nahegelegte Formel für die Anzahl der Kompositionen von n und die Anzahl der Kompositionen von n der Länge k; untersuchen Sie durch Introspektion, ob Sage diese Formeln zur Berechnung von Kardinalitäten verwendet.

15.2.3. Einige andere abzählbare endliche Mengen

Das Prinzip ist für alle endlichen Mengen, mit denen wir in Sage Kombinatorik betreiben wollen, im wesentlichen das gleiche; wir beginnen mit der Konstruktion eines Objektes, das diese Menge modelliert und benutzen dann die passenden Methoden, die einem immer gleichen Interface folgen[2]. Es folgen noch ein paar weitere typische Beispiele.

Intervalle natürlicher Zahlen:

```
sage: C = IntegerRange(3, 21, 2); C
{3, 5, ..., 19}
```

[1] (

[2] Oder zumindest sollte das so sein. Es sind aber immer noch etliche Ecken zu fegen.

```
sage: C.cardinality()
9
sage: C.list()
[3, 5, 7, 9, 11, 13, 15, 17, 19]
```

Permutationen:

```
sage: C = Permutations(4); C
Standard permutations of 4
sage: C.cardinality()
24
sage: C.list()
[[1, 2, 3, 4], [1, 2, 4, 3], [1, 3, 2, 4], [1, 3, 4, 2],
 [1, 4, 2, 3], [1, 4, 3, 2], [2, 1, 3, 4], [2, 1, 4, 3],
 [2, 3, 1, 4], [2, 3, 4, 1], [2, 4, 1, 3], [2, 4, 3, 1],
 [3, 1, 2, 4], [3, 1, 4, 2], [3, 2, 1, 4], [3, 2, 4, 1],
 [3, 4, 1, 2], [3, 4, 2, 1], [4, 1, 2, 3], [4, 1, 3, 2],
 [4, 2, 1, 3], [4, 2, 3, 1], [4, 3, 1, 2], [4, 3, 2, 1]]
```

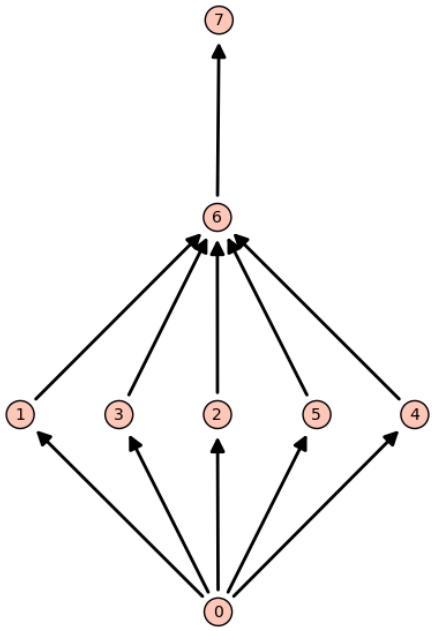


Abb. 15.2 - Ein Poset mit 8 Knoten

Mengenpartitionen:

```
sage: C = SetPartitions([1,2,3]); C
Set partitions of {1, 2, 3}
sage: C.cardinality()
5
sage: C.list()
[{{1, 2, 3}}, {{1}, {2, 3}}, {{1, 3}, {2}}, {{1, 2}, {3}}, {{1}, {2}, {3}}]
```

Teilweise Ordnungen (posets) auf Mengen mit 8 Elementen bis auf Isomorphie:

```
sage: C = Posets(8); C
Posets containing 8 elements
sage: C.cardinality()
16999
```

Wir wollen eins dieser Posets zeichnen (siehe Abb. 15.2):

```
sage: show(C.unrank(20))
```

Wir können auch über alle diese Graphen bis auf Isomorphie iterieren. Es existieren beispielsweise 34 einfache Graphen mit 5 Knoten:

```
sage: len(list(graphs(5)))
34
```

Hier nun, wie wir diejenigen bekommen, die höchstens 4 Kanten haben (siehe Abb. 15.3):

```
sage: for g in graphs(5, lambda G: G.size() <= 4): show(g)
```

Allerdings steht die *Menge C* dieser Graphen in Sage *noch nicht* zur Verfügung. Die fogenden Befehle sind daher noch nicht implementiert:

```
sage: C = Graphs(5); C.cardinality()
34
sage: Graphs(5); C.cardinality()
24637809253125004524383007491432768
sage: Graphs(19).random_element()
Graph on 19 vertices
```

Was wir bis jetzt gesehen haben, lässt sich im Prinzip auf endliche algebraische Strukturen wie Diedergruppen anwenden:

```
sage: G = DihedralGroup(4); G
Dihedral group of order 8 as a permutation group
sage: G.cardinality()
8
sage: G.list()
[(), (1,4)(2,3), (1,2,3,4), (1,3)(2,4), (1,3), (2,4), (1,4,3,2), (1,2)(3,4)]
```

oder die Algebra der 2×2-Matrizen auf dem endlichen Körper $\mathbb{Z}/2\mathbb{Z}$:

```
sage: C = MatrixSpace(GF(2), 2); C.list()
[
[0 0]  [1 0]  [0 1]  [0 0]  [0 0]  [1 1]  [1 0]  [1 0]  [0 1]  [0 1]
[0 0], [0 0], [0 0], [1 0], [0 1], [0 0], [1 0], [0 1], [1 0], [0 1],

[0 0]  [1 1]  [1 1]  [1 0]  [0 1]  [1 1]
[1 1], [1 0], [0 1], [1 1], [1 1], [1 1]
]

sage: C.cardinality()
16
```

Übung 53. Listen Sie alle Monome 5. Grades in drei Variablen auf (siehe `IntegerVectors`). Bearbeiten Sie die geordneten Mengenpartitionen (`OrderedSetPartitions`) und die Standardtabellen (`StandardTableaux`).

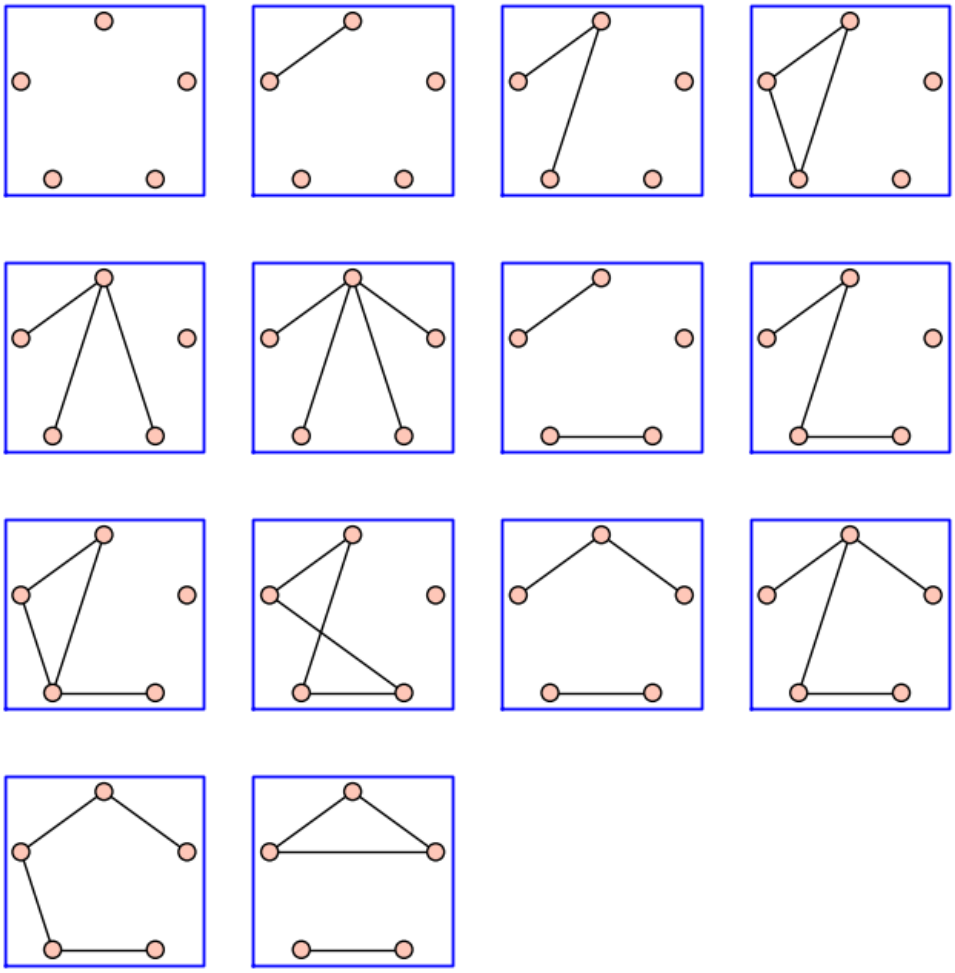


Abb. 15.3 - Die einfachen Graphen mit 5 Knoten und höchstens 4 Kanten

Übung 54. Listen Sie die Matrizen mit alternierenden Vorzeichen der Größe 3, 4 und 5 auf und versuchen Sie, die Definition zu erraten (siehe `AlternatingSignMatrices`). Entdeckung und Beweis dieser Formel für die Auflistung dieser Matrizen (siehe die Methode `cardinality`), die durch Berechnungen von Determinanten in der Physik motiviert worden waren, sind eine eigene Geschichte. Insbesondere ist der erste Beweis von Zeilberger 1992 durch ein Rechnerprogramm automatisch geführt worden. Er war 84 Seiten lang und es brauchte nahezu 100 Leute, um ihn zu verfizieren [Zei96].

Übung 55. Berechnen Sie die Anzahl der Vektoren in $(\mathbb{Z}/2\mathbb{Z})^5$ von Hand und dann die Anzahl der Matrizen in $\mathrm{GL}_3(\mathbb{Z}/2\mathbb{Z})$ (d.h. die Anzahl der 3×3-Matrizen mit Koeffizienten in $\mathbb{Z}/2\mathbb{Z}$ und Inversen). Verifizieren Sie Ihre Antworten mit Sage. Verallgemeinern Sie das zu $\mathrm{GL}_n(\mathbb{Z}/q\mathbb{Z})$.

Mengenraffungen und Iteratoren

Wir zeigen nun einige der von Python angebotenen Möglichkeiten der Bildung von (und des Durchlaufs durch) Mengen in einer Schreibweise, die flexibel ist und nahe am üblichen mathematischen Gebrauch sowie insbesondere, welche Vorteile das für die Kombinatorik hat.

Wir beginnen mit der Bildung der endlichen Menge $\{i^2 \mid i \in \{1,3,7\}\}$:

```
sage: [ i^2 for i in [1, 3, 7] ]
[1, 9, 49]
```

und dann der gleichen Menge, nur dass i nun von 0 bis 9 läuft:

```
sage: [ i^2 for i in range(1,10) ]
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

In Python heißt diese Art der Bildung einer Menge *Mengenraffung*. Eine Bedingung kann hinzugefügt werden, um nur die primen Elemente zu behalten:

```
sage: [ i^2 for i in range(1,10) if is_prime(i) ]
[4, 9, 25, 49]
```

Durch Kombination zweier Mengenraffungen kann die Menge $\{(i,j) \mid 1 \leq k < i < 6\}$ erzeugt werden:

```
sage: [ (i,j) for i in range(1,6) for j in range(1,i) ]
[(2, 1), (3, 1), (3, 2), (4, 1), (4, 2), (4, 3),
 (5, 1), (5, 2), (5, 3), (5, 4)]
```

oder das pascalsche Dreieck:

```
sage: [[binomial(n, i) for i in range(n+1)] for n in range(10)]
[[1],
 [1, 1],
 [1, 2, 1],
 [1, 3, 3, 1],
 [1, 4, 6, 4, 1],
 [1, 5, 10, 10, 5, 1],
 [1, 6, 15, 20, 15, 6, 1],
 [1, 7, 21, 35, 35, 21, 7, 1],
 [1, 8, 28, 56, 70, 56, 28, 8, 1],
 [1, 9, 36, 84, 126, 126, 84, 36, 9, 1]]
```

Die Ausführung einer Raffung erfolgt in zwei Schritten; zuerst wird ein *Iterator* erzeugt, dann wird eine Liste nach und nach mit den vom Iterator zurückgegebenen Werten gefüllt. Technisch gesehen ist ein *Iterator* ein Objekt mit einer Methode `next`, die solange es geht bei jedem Aufruf einen neuen Wert zurückgibt.

```
sage: it = (binomial(3, i) for i in range(4))
```

gibt nacheinander die Binomialkoeffizienten $\binom{3}{i}$ mit $i = 0, 1, 2, 3$ zurück:

```
sage: it.next()
1
```

```
sage: it.next()
3
sage: it.next()
3
sage: it.next()
1
```

Wenn der Iterator schließlich erschöpft ist, erscheint eine Exception:

```
sage: it.next()
Traceback (most recent call last):
  ...
StopIteration
```

Allgemein, eine *Iterable* ist ein Python-Objekt `L` (eine Liste, eine Menge,...), über deren Elemente iteriert werden kann. Technisch wird der Iterator durch `iter(L)` erzeugt. In der Praxis werden die Befehle `iter` und `next` selten gebraucht, weil `for`-Schleifen und Raffungen eine viel elegantere Syntax bieten:

```
sage: for s in Subsets(3): s
{}
{1}
{2}
{3}
{1, 2}
{1, 3}
{2, 3}
{1, 2, 3}
```

```
sage: [ s.cardinality() for s in Subsets(3) ]
0, 1, 1, 1, 2, 2, 2, 3]
```

Welches Interesse besteht an Iteratoren? Sehen wir uns das folgende Beispiel an:

```
sage: sum( [ binomial(8, i) for i in range(9) ] )
256
```

Bei der Ausführung wird eine Liste von 9 Elementen gebildet, dann wird sie als Argument an `sum` übergeben, um die Elemente hinzuzufügen. Übergibt man den Iterator jedoch direkt an `sum` (zu beachten ist das Fehlen der eckigen Klammern),

```
sage: sum( binomial(8, i) for i in xrange(9) )
256
```

übernimmt die Funktion `sum` den Iterator direkt und kann die Konstruktion der Zwischenliste kurzschließen. Bei einer großen Anzahl von Elementen vermeidet das die Allokation eines großen Speicherbereichs, der mit der Liste gefüllt wird, die sofort wieder gelöscht wird[3].

Die meisten Funktionen die eine Liste von Elementen als Argument erhalten, akzeptieren stattdessen auch einen Iterator (oder eine Iterable). Um damit zu beginnen, können wir die Liste (oder das Tupel) von Elementen wie folgt erhalten:

[3] Technisches Detail: `xrange` gibt einen Iterator über $\{1, \ldots, 8\}$ zurück, während `range` die entsprechende Liste zurückgibt. Ab Python 3.0 verhält sich `range` genauso wie `xrange` und `xrange` wird nicht länger gebraucht.

```
sage: list(binomial(8, i) for i in xrange(9))
[1, 8, 28, 56, 70, 56, 28, 8, 1]
sage: tuple(binomial(8, i) for i in xrange(9))
(1, 8, 28, 56, 70, 56, 28, 8, 1)
```

Wir betrachten nun die Funktionen `all` und `any`, die wie `and` bzw. `or`, aber für mehrere Argumente wirken.

```
sage: all([True, True, True, True])
True
sage: all([True, False, True, True])
False
sage: any([False, False, False, False])
False
sage: any([False, False, True, False])
True
```

Das folgende Beispiel bestätigt, dass alle Primzahlen von 3 bis 99 ungerade sind:

```
sage: all( is_odd(p) for p in xrange(3,100) if is_prime(p) )
True
```

Eine Mersenne-Zahl ist eine Primzahl der Form $2^p - 1$. Wir verifizieren für $p < 1000$, dass p prim ist, wenn $2^p - 1$ prim ist:

```
sage: def mersenne(p): return 2^p - 1
sage: [ is_prime(p) for p in range(1000) if is_prime(mersenne(p)) ]
[True, True, True, True, True, True, True, True, True, True,
 True, True, True, True]
```

Ist auch die Umkehrung wahr?

Übung 56. Probieren Sie die folgenden Befehle aus und erklären Sie die beträchtlichen Unterschiede in den Ausführungszeiten.

```
sage: all( [ is_prime(mersenne(p)) for p in range(1000) if is_prime(p)] )
False
sage: all( is_prime(mersenne(p)) for p in range(1000) if is_prime(p) )
False
```

Wir versuchen jetzt, das kleinste Gegenbeispiel zu finden. Dazu verwenden wir die Sage-Funktion `exists`:

```
sage: exists( (p for p in range(1000) if is_prime(p)),
...:      lambda p: not is_prime(mersenne(p)) )
(True, 11)
```

Alternativ können wir einen Iterator über alle Gegenbeispiele konstruieren:

```
sage: contre_exemples = \
....:     (p for p in range(1000)
....:        if is_prime(p) and not is_prime(mersenne(p)))
sage: contre_exemples.next()
11 sage: contre_exemples.next()
23
```

Übung 57. Was bewirken die folgenden Befehle?

```
sage: cubes = [t**3 for t in range(-999,1000)]
sage: exists([(x,y) for x in cubes for y in cubes], lambda (x,y): x+y == 218)
sage: exists(((x,y) for x in cubes for y in cubes), lambda (x,y): x+y == 218)
```

Welche der letzten beiden ist sparsamer an Zeit? An Speicherplatz?

Übung 58. Probieren Sie alle folgenden Befehle und erläutern Sie die Ergebnisse. Warnung: Bei einigen davon muss die Ausführung abgebrochen werden.

```
sage: x = var('x'); sum( x^len(s) for s in Subsets(8) )

sage: sum( x^p.length() for p in Permutations(3) )

sage: factor(sum( x^p.length() for p in Permutations(3) ))

sage: P = Permutations(5)
sage: all( p in P for p in P )

sage: for p in GL(2, 2): print p; print

sage: for p in Partitions(3): print p

sage: for p in Partitions(): print p

sage: for p in Primes(): print p

sage: exists( Primes(), lambda p: not is_prime(mersenne(p)) )

sage: contre_exemples = (p for p in Primes()
....:                    if not is_prime(mersenne(p)))
sage: for p in contre_exemples: print p
```

Operationen auf Iteratoren. Zur Manipulation von Iteratoren bietet Python zahlreiche Möglichkeiten; die meisten davon sind Teil der Bibliothek `itertools`, die importiert werden kann mit

```
sage: import itertools
```

Wir zeigen einige Anwendungen und nehmen als Ausgangspunkt die Permutationen von 3:

```
sage: list(Permutations(3))
[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
```

Wir können die Elemente einer Menge auflisten, indem wir sie numerieren:

```
sage: list(enumerate(Permutations(3)))
[(0, [1, 2, 3]), (1, [1, 3, 2]), (2, [2, 1, 3]),
 (3, [2, 3, 1]), (4, [3, 1, 2]), (5, [3, 2, 1])]
```

oder nur die Elemente an den Positionen 2, 33 und 4 (analog zu $l[1:4]$):

```
sage: list(itertools.islice(Permutations(3), 1, 4))
[[1, 3, 2], [2, 1, 3], [2, 3, 1]]
```

oder eine Funktion auf alle Elemente anwenden:

```
sage: list(itertools.imap(lambda z: z.cycle_type(), Permutations(3)))
[[1, 1, 1], [2, 1], [2, 1], [3], [3], [2, 1]]
```

oder die Elemente auswählen, die einer bestimmten Bedingung genügen:

```
sage: list(itertools.ifilter(lambda z: z.has_pattern([1,2]),
sage: ....:                  Permutations(3)))
[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2]]
```

In allen diesen Situationen kann `attrcall` bei der Erstellung einer anonymen Funktion eine vorteilhafte Alternative sein:

```
sage: list(itertools.imap(attrcall("cycle_type"), Permutations(3)))
[[1, 1, 1], [2, 1], [2, 1], [3], [3], [2, 1]]
```

Implementierung neuer Iteratoren. Neue Iteratoren zu konstruieren ist einfach, wenn man in einer Funktion das Schlüsselwort `yield` anstelle von `return` verwendet:

```
sage: def f(n):
....:     for i in range(n):
....:         yield i
```

Nach `yield` wird die Ausführung nicht beendet, sondern nur angehalten und bereit vom selben Punkt an fortzufahren. Das Ergebnis der Funktion ist daher ein Iterator über die aufeinander folgenden Werte, die von `yield` zurückgegeben werden:

```
sage: g = f(4)
sage: g.next()
0
sage: g.next()
1
sage: g.next()
2
sage: g.next()
3

sage: g.next()
Traceback (most recent call last):
  ...
StopIteration
```

Die Funktion kann folgendermaßen angewendet werden:

```
sage: [ x for x in f(5) ]
[0, 1, 2, 3, 4]
```

Dieses Berechnungsschema namens *Kontinuation* erweist sich für die Kombinatorik als sehr nützlich, besonders wenn es mit Rekursion gekoppelt wird. (Siehe auch Unterabschnitt 2.2.2 wegen weiterer Anwendungen.) Hier folgt, wie alle Wörter einer gegebenen Länge über einem gegebenen Alphabet erzeugt werden:

```
sage: def words(alphabet,l):
....:     if l == 0: yield []
```

```
....:     else:
....:         for word in words(alphabet, l-1):
....:             for l in alphabet: yield word + [l]
sage: [ w for w in words(['a','b'], 3) ]
[['a', 'a', 'a'], ['a', 'a', 'b'], ['a', 'b', 'a'], ['a', 'b', 'b'],
 ['b', 'a', 'a'],['b', 'a', 'b'],['b', 'b', 'a'], ['b', 'b', 'b']]
```

Diese Wörter können gezählt werden mit

```
sage: sum(1 for w in words(['a','b','c','d'], 10))
1048576
```

Das Abzählen der Wörter eines nach dem anderen ist hier natürlich kein effizientes Verfahren, da man die Formel n^l benutzen kann; beachten Sie aber, dass dies nicht der stupideste Ansatz ist - er vermeidet immerhin die Erzeugung der kompletten Liste und deren Speicherung.

Wir betrachten nun die Wörter der Dyck-Sprache. Das sind wohlgeformte Ausdrücke mit den Buchstaben „(“ und „)“. Die untenstehende Funktion erzeugt nun alle Dyck-Wörter gegebener Länge (wobei unter Länge die Anzahl der Klammerpaare zu verstehen ist), indem sie die rekursive Definition anwendet, die besagt, dass ein Dyck-Wort entewder leer ist oder von der Form $(w_1)w_2$, wobei w_1 und w_2 Dyck-Wörter sind.

```
sage: def dyck_words(l):
....:       if l == 0: yield ''
....:       else:
....:           for k in range(l):
....:               for w1 in dyck_words(k):
....:                   for w2 in dyck_words(l-k-1):
....:                       yield '(' + w1 + ')' + w2
```

Hier sind alle Dyck-Wörter der Länge 4:

```
sage: list(dyck_words(4))
['()()()()', '()()(())', '()(())()', '()((()))', '(())()()',
 '(())(())', '(()())()', '((()))()', '(()()())',
 '(()(()))', '((())())', '((()()))', '(((())))']
```

Beim Abzählen entdecken wir eine wohlbekannte Folge:

```
sage: [ sum(1 for w in dyck_words(l)) for l in range(10) ]
[1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862]
```

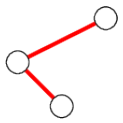
Übung 59. Erzeugen Sie einen Iterator auf der Menge C_n der vollständigen Binärbäume mit n Blättern (siehe Unterabschnitt 15.1.2).

Hinweis: Verwenden Sie `BinaryTree`; im untenstehenden Beispiel erzeugen wir ein Blatt und den zweiten Baum von Abbildung 15.1.

```
sage: BT = BinaryTree
sage: BT()
.
sage: t = BT([BT([BT(), BT([BT(),BT()])]), BT()]); t
[[., [., .]], .]
```

Vorsicht! Sage benutzt beim Zeichnen eines vollständigen Binärbaums die klassische Konvention, nur sein Skelett darzustellen.

```
sage: view(t)
```



15.3. Konstruktionen

Wir wollen nun sehen, wie ausgehend von diesen Bausteinen neue Mengen gebildet werden. In Wirklichkeit haben wir im vorigen Abschnitt mit der Konstruktion von $\mathcal{P}(\mathcal{P}(\mathcal{P}(\{1,2,3\})))$ und mit der Konstruktion von Kartenmengen in Abschnitt 15.1 schon damit begonnen.

Betrachten wir ein großes cartesisches Produkt:

```
sage: C = cartesian_product([Compositions(8), Permutations(20)]); C
The Cartesian product of (Compositions of 8, Standard permutations of 20)
sage: C.cardinality()
311411457046609920000
```

Klar, es ist unpraktisch, die Liste aller dieser Elemente des cartesischen Produktes zu bilden. Trotzdem kann man mit ihr arbeiten und beispielsweise ein zufälliges Element generieren:

```
sage: C.random_element()
([2, 3, 2, 1], [10, 6, 11, 13, 14, 3, 4, 19, 5, 12, 7, 18, 15, 8, 20, 1,
     17, 2, 9, 16])
```

Die Konstruktion `cartesian_product` kennt die algebraischen Eigenschaften ihrer Elemente. Deshalb ist H im folgenden Beispiel genauso wie seine Produktgruppe mit den normalen Operationen der Kombinatorik ausgestattet.

```
sage: G = DihedralGroup(4)
sage: H = cartesian_product([G,G])
sage: H.cardinality()
64
sage: H in Sets().Enumerated().Finite()
True
sage: H in Groups
True
```

Wir bilden jetzt die disjunkte Vereinigung der beiden vorliegenden Mengen:

```
sage: C = DisjointUnionEnumeratedSets([Compositions(4),Permutations(3)])
sage: C
Disjoint union of Family (Compositions of 4, Standard permutations of 3)
sage: C.cardinality()
14
sage: C.list()
```

```
[[1, 1, 1, 1], [1, 1, 2], [1, 2, 1], [1, 3], [2, 1, 1], [2, 2], [3, 1],
 [4], [1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
```

Es ist auch möglich, die Vereinigung von mehr als zwei disjunkten Mengen zu nehmen, ja sogar von einer unbegrenzten Anzahl von Mengen. Wir bilden jetzt die Menge aller Permutationen, aufgefasst als Vereinigung der Mengen P_n der Permutationen der Größe n. Wir beginnen mit der unendlichen Familie $F = (P_n)_{n\in\mathbb{N}}$:

```
sage: F = Family(NonNegativeIntegers(), Permutations); F
Lazy family (<class 'sage.combinat.permutation.Permutations'>(i))_{i in
    Non negative integers}
sage: F.keys()
Non negative integers
sage: F[1000]
Standard permutations of 1000
```

Jetzt können wir die disjunkte Vereinigung $\bigcup_{n\in\mathbb{N}} P_n$ bilden:

```
sage: U = DisjointUnionEnumeratedSets(F); U
Disjoint union of
Lazy family (<class 'sage.combinat.permutation.Permutations'>(i))_{i in
    Non negative integers}
```

Das ist eine unendliche Menge:

```
sage: U.cardinality()
+Infinity
```

die aber die Iteration über ihre Elemente nicht verhindert, so dass sie irgendwann abgebrochen werden muss.

```
sage: for p in U: p
[]
[1]
[1, 2]
[2, 1]
[1, 2, 3]
[1, 3, 2]
[2, 1, 3]
[2, 3, 1]
[3, 1, 2]
...
```

Beachten Sie, dass obige Menge auch direkt erzeugt werden kann mit

```
sage: U = Permutations(); U
Standard permutations
```

Zusammenfassung. Um es kurz zu sagen, Sage bietet eine Biblithek von abzählbaren Mengen, die durch Standardkonstruktionen kombiniert werden können, was eine Toolbox ergibt, die flexibel ist (aber noch erweitert werden kann). Es ist auch möglich, Sage mit wenigen Zeilen neue Bausteine hinzuzufügen (siehe den Code in `Sets().Enumerated().Finite()`). Das wird möglich durch die Übereinstimmung der Schnittstellen und die Tatsache, dass Sage auf einer objektorientierten Sprache basiert. Es können dank der faulen Auswertungsstrategie (Iteratoren usw.) auch sehr große und sogar unendliche Mengen bearbeitet werden.

Da ist keine Zauberei dabei: unter der Haube wendet Sage die normalen Regeln an (beispielsweise, dass die Kardinalität von $E \times E$ gleich $|E|^2$ ist); der Mehrwert stammt aus der Fähigkeit, komplizierte Strukturen zu verarbeiten. Die Situation ist vergleichbar mit Sages Implementierung der Differentialrechnung: Sage verwendet die bekannten Regeln für die Ableitung von Funktionen und ihre Kompositionen, wobei hier der Wertzuwachs aus der Möglichkeit hervorgeht, komplizierte Gleichungen zu verarbeiten. In diesem Sinne implementiert Sage eine *Analysis* endlicher abzählbarer Mengen.

15.4. Generische Algorithmen

15.4.1. Lexikographische Erzeugung ganzzahliger Listen

Unter den klassischen abzählbaren Mengen ist besonders in der algebraischen Kombinatorik eine bestimmte Anzahl aus ganzzahligen Listen mit konstanter Summe aufgebaut, wie Partitionen, Kompositionen oder ganzzahligen Vektoren. Diese Beispiele können auch ergänzende Nebenbedingungen aufweisen, die ihnen hinzugefügt worden sind. Hier sind einige Beispiele. Wir beginnen mit dem ganzzahligen Vektor mit der Summe 10 und der Länge 3 mit Teilen, die unten durch 2, 4 und 2 begrenzt sind

```
sage: IntegerVectors(10, 3, min_part = 2, max_part = 5,
                     inner = [2, 4, 2]).list()
[[4, 4, 2], [3, 5, 2], [3, 4, 3], [2, 5, 3], [2, 4, 4]]
```

Die Komposition von 5 mit jedem Teil von höchstens 3 und Länge 2 oder 3:

```
sage: Compositions(5, max_part = 3,
                   min_length = 2, max_length = 3).list()
[[3, 2], [3, 1, 1], [2, 3], [2, 2, 1], [2, 1, 2], [1, 3, 1],
 [1, 2, 2], [1, 1, 3]]
```

Die streng fallenden Partitionen von 5:

```
sage: Partitions(5, max_slope = -1).list()
[[5], [4, 1], [3, 2]]
```

Diesen Mengen liegt die gleiche algorithmische Struktur zugrunde, die in der allgemeineren - und etwas umständlicher zu benutzenden - Klasse `IntegerListsLex` implementiert ist. Diese Klasse modelliert Mengen von Vektoren $(l_0, \ldots, l_l)$ nicht-negativer ganzer Zahlen mit Nebenbedingungen für Summe und Länge, sowie Grenzen für Teile aufeinander folgende Differenzen zwischen den Teilen. Hier sind noch einige Beispiele:

```
sage: IntegerListsLex(10, length=3, min_part = 2, max_part = 5,
                      floor = [2, 4, 2]).list()
[[4, 4, 2], [3, 5, 2], [3, 4, 3], [2, 5, 3], [2, 4, 4]]

sage: IntegerListsLex(5, min_part = 1, max_part = 3,
                      min_length = 2, max_length = 3).list()
[[3, 2], [3, 1, 1], [2, 3], [2, 2, 1], [2, 1, 2], [1, 3, 1],
 [1, 2, 2], [1, 1, 3]]

sage: IntegerListsLex(5, min_part = 1, max_slope = -1).list()
[[5], [4, 1], [3, 2]]

sage: list(Compositions(5, max_length=2))
[[5], [4, 1], [3, 2], [2, 3], [1, 4]]

sage: list(IntegerListsLex(5, max_length=2, min_part=1))
[[5], [4, 1], [3, 2], [2, 3], [1, 4]]
```

Für das Modell von `IntegerListsLex` spricht der gute Kompromiss zwischen Allgemeingültigkeit und Effizienz der Iteration. Der Hauptalgorithmus gestattet die Iteration über eine solche Menge S in inverser lexikographischer Ordnung und konstanter mittlerer Zeitkomplexität (Constant Amortized Time - CAT), außer in pathologischen Fällen; grob gesagt ist die Zeit, um über alle Elemente zu iterieren proportional zur Anzahl der Elemente, und das ist optimal. Außerdem ist der Speicherbedarf proportional zum größten gefundenen Element, in der Praxis also vernachlässigbar,

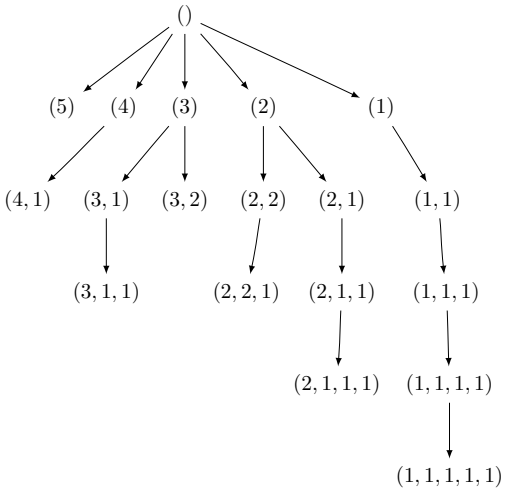


Abb. 15.4 - Der Präfix-Baum der Partitionen von 5.

Dieser Algorithmus basiert auf einem sehr allgemeinen Prinzip der Traversierung eines Entscheidungsbaumes, nämlich Branch-and-Bound: auf dem obersten Niveau durchlaufen wir alle möglichen Alternativen für l_0; zu jeder dieser Alternativen durchlaufen wir alle möglichen Alternativen für l_1 und so fort. Mathematisch gesprochen haben wir auf die Elemente von S die Struktur eines Präfix-Baumes gelegt. Ein Knoten des Baumes in der Tiefe k entspricht einem Präfix $l_0, \ldots, l_k$ von einem oder mehreren Elementen von S (siehe Abbildung 15.4).

Das übliche Problem bei diesem Anatz ist die Vermeidung schlechter Entscheidungen, die zum Verlassen des Präfix-Baumes führen und zur Erkundung toter Äste. Das ist deshalb besonders

problematisch, weil die Anzahl der Elemente mit der Tiefe exponentiell anwächst. Es stellt sich heraus, dass die oben aufgelisteten Bedingungen einfach genug sind, um die folgende Eigenschaft zu garantieren: bei gegebenem Präfix $l_0, \ldots, l_k$ von S ist die Menge der l_{k+1} so, dass $l_0, \ldots, l_{k+1}$ als Präfix von S entweder leer ist oder ein Intervall der Form $[a, b]$, und die Grenzen a und b in linearer Zeit, also proportional zur Länge des längsten Elementes von S berechnet werden können, welches das Präfix $l_0, \ldots, l_l$ besitzt.

15.4.2. Ganzzahlige Punkte in Polytopen

Auch wenn der Algorithmus in `IntegerListsLex` effizient ist, ist sein Zählalgorithmus doch naiv: er iteriert schlicht über alle Elemente.

Es gibt für dieses Problem einen alternativen Ansatz: die Modellierung der gewünschten ganzzahligen Liste als Menge der ganzzahligen Punkte eines Polytops, will sagen, die Menge der Lösungen mit ganzzahligen Koordinaten eines Systems von linearen Ungleichungen. In diesem sehr allgemeinen Kontext gibt es hoch entwickelte Zählalgorithmen (z.B. Barvinok), die in Bibliotheken wie LattE implementiert sind. Iteration stellt im Prinzip kein großes Problem dar, jedoch gibt es zwei Einschränkungen, die die Existenz von `IntegerListsLex` rechtfertigen. Die erste ist theoretidscher Natur: die Gitterpunkte in einem Polytop erlauben nur die Modellierung von Problemen fester Dimension (Länge). Die zweite ist praktischer Art: zur Zeit hat nur die Bibliothek PALP eine Schnittstelle zu Sage, und obwohl es mehrere Möglichkeiten für die Untersuchung von Polytopen bietet, produziert es in der gegenwärtigen Fassung lediglich eine Liste von Gitterpunkten, ohne für einen Iterator oder nicht-naives Zählen zu sorgen:

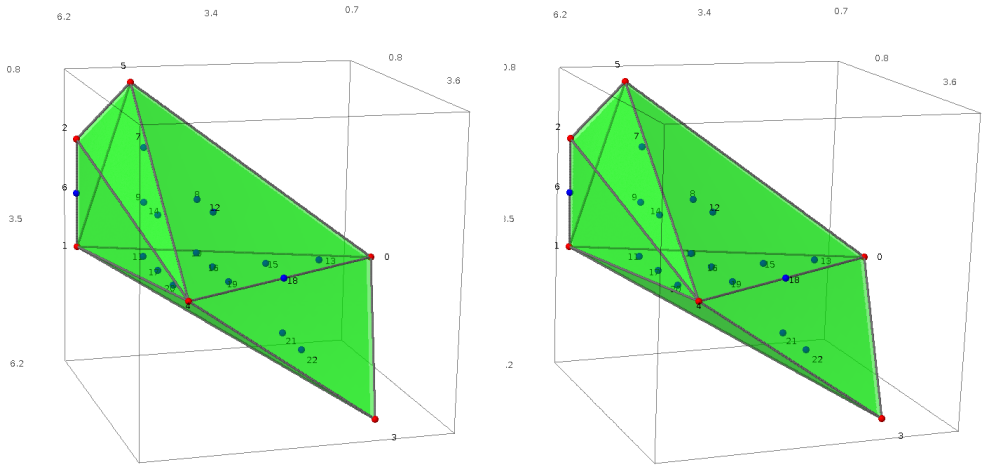


Abb. 15.5 - Das Polytop L und seine ganzzahligen Punkte in stereographischer Ansicht (Kreuzblick)

Hier folgt, wie dieses Polytop in 3D gezeichnet wird (siehe Abb. 15.5):

```
sage: A = random_matrix(ZZ,6,3,x=7)
sage: L = LatticePolytope(A)
sage: L.points()
M(1, 4, 3),
M(6, 4, 1),
...
M(3, 5, 5),
```

```
in 3-d lattice M
sage: L.points().cardinality()
23

sage: L.plot3d()
```

15.4.3. Arten, zerlegbare kombinatorischer Klassen

In Unterabschnitt 15.1.2 hatten wir gezeigt, wie die rekursive Definition binärer Bäume dazu dient, sie mit generierten Funktionen effizient abzuzählen. Die verwendeten Techniken sind sehr allgemein und lassen sich immer dann anwenden, wenn die beteiligten Mengen rekursiv definiert werden können (je nachdem, wen sie fragen, heißt eine solche Menge eine zerlegbare kombinatorische Klasse oder, grob gesagt, eine kombinatorische Art). Das umfasst alle Typen von Bäumen und auch Permutationen, Kompositionen, Funktionsgraphen usw.

Hier erläutern wir einige nur einige wenige Beispiele, die die Sage-Bibliothek für kombinatorische Arten verwenden.

```
sage: from sage.combinat.species.library import *
sage: o = var('o')
```

Wir beginnen damit, dass wir vollständige Binärbäume neu definieren; dazu wenden wir die Rekursionsbeziehung direkt auf die Mengen an:

```
sage: BT = CombinatorialSpecies()
sage: Leaf = SingletonSpecies()
sage: BT.define( Leaf + (BT*BT) )
```

Nun können wir die Menge der Bäume mit fünf Knoten bilden, sie auflisten, sie zählen. . . :

```
sage: BT5 = BT.isotypes([o]*5); BT5.cardinality()
14
sage: BT5.list()
[o*(o*(o*(o*o))), o*(o*((o*o)*o)), o*((o*o)*(o*o)), o*((o*(o*o))*o),
 o*(((o*o)*o)*o), (o*o)*(o*(o*o)), (o*o)*((o*o)*o), (o*(o*o))*(o*o),
 ((o*o)*o)*(o*o), (o*(o*(o*o)))*o, (o*((o*o)*o))*o, ((o*o)*(o*o))*o,
 ((o*(o*o))*o)*o, (((o*o)*o)*o)*o]
```

Die Bäume sind mittels einer generischen rekursiven Struktur erzeugt worden; die Ausgabe ist deshalb nicht berauschend. Um sie zu verbessern, müsste Sage mit spezielleren Datenstrukturen ausgestattet sein, die über die gewünschten Darstellungsfähigkeiten verfügen. Wir teffen wieder auf die erzeugende Funktion der Catalan-Zahlen:

```
sage: g = BT.isotype_generating_series(); g
x + x^2 + 2*x^3 + 5*x^4 + 14*x^5 + O(x^6)
```

die als faule Potenzreihe zurückgegen wird:

```
sage: g[100]
227508830794229349661819540395688853956041682601541047340
```

Wir schließen mit den Fibonacci-Wörtern, binären Wörtern ohne zwei aufeinander folgende „1“. Sie ermöglichen eine natürliche rekursive Definition:

```
sage: Eps = EmptySetSpecies(); Z0 = SingletonSpecies()
sage: Z1 = Eps*SingletonSpecies()
sage: FW = CombinatorialSpecies()
sage: FW.define(Eps + Z0*FW + Z1*Eps + Z1*Z0*FW)
```

Die Fibonacci-Folge kann hier leicht wiedererkannt werden, daher der Name:

```
sage: L = FW.isotype_generating_series().coefficients(15); L
[1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987]
```

```
sage: oeis(L)
0: A000045: Fibonacci numbers: F(n) = F(n-1) + F(n-2) with F(0) = 0 and
     F(1) = 1.
1: A212804: Expansion of (1-x)/(1-x-x^2).
2: A132636: a(n) = Fibonacci(n) mod n^3
```

Das ist eine unmittelbare Konsequenz der Rekursionsrelation. Mit denselben Einschränkungen, die von der generischen Anzeige herrühren, können wir auch gleich alle Fibonacci-Wörter gegebener Länge generieren.

```
sage: FW3 = FW.isotypes([o]*3)
sage: FW3.list()
[o*(o*(o*{})), o*(o*(({}*o)*{})), o*((({}*o)*o)*{}),
 (({}*o)*o)*(o*{}), (({}*o)*o)*(({}*o)*{})]
```

Nach Ersetzen von ∘ durch 0, von `{}*o` durch 1 und Weglassen der runden Klammern und auch des letzten geschweiften Klammerpaars `{}` lesen wir 000, 001, 100 bzw. 101.

15.4.4. Graphen bis auf Isomorphie

In Unterabschnitt 15.2.3 haben wir gesehen, dass Sage Graphen und partielle Ordnungen bis auf Isomorphie erzeugt. Jetzt wollen wir den zugrunde liegenden Algorithmus beschreiben, der in beiden Fällen der gleiche ist und eine wesentlich breitere Klasse von Problemen abdeckt.

Wir rufen zunächst einige Begriffe in Erinnerung. Ein Graph $G = (V, E)$ ist eine Menge V von Knoten und eine Menge E von Kanten, die diese Knoten verbinden; eine Kante wird durch ein Paar $\{u, v\}$ verschiedener Knoten aus V beschrieben. Ein solcher Graph heißt benannt; seine Knoten sind typischerweise numeriert, zum Beispiel ist $V = \{1, 2, 3, 4, 5\}$.

Bei vielen Problemen spielen die Namen der Knoten keine Rolle. Ein Chemiker möchte typischerweise alle möglichen Moleküle mit gegebener Zusammensetzung untersuchen, zum Beispiel die Alkane mit $n = 8$ Kohlenstoffatomen und $2n + 2 = 18$ Wasserstoffatomen. Daher möchte er alle Graphen finden, die aus 8 Knoten mit 4 Nachbarn bestehen und 18 Knoten mit nur einem Nachbarn. Die verschiedenen Kohlenstoffatome werden jedoch alle als identisch angesehen, desgleichen die Wasserstoffatome. Unser Chemieproblem ist nicht eingebildet; dieser Anwendungstyp steht aktuell am Anfang eines wichtigen Forschungszweiges zu isomorphen Problemen in der Graphentheorie.

Arbeitet man von Hand mit einem kleinen Graphen, ist es wie im Beispiel von Unterabschnitt 15.2.3 möglich, eine Zeichnung zu erstellen, die Bezeichner zu entfernen und die geometrische Information zur Lage der Knoten in der Ebene zu „vergessen“. Um jedoch einen Graphen in einem Rechnerprogramm darzustellen, müssen an den Knoten Bezeichner eingeführt werden,

um beschreiben zu können, wie die Kanten miteinander verbunden sind. Um dann die dadurch eingeführte Extrainformation zu kompensieren, sagen wir, dass zwei Graphen g_1 und g_2 *isomorph* sind, wenn von den Knoten von g_1 zu den Knoten von g_2 eine Bijektion existiert, welche den Kanten von g_1 die Kanten von g_2 bijektiv zuordnet. Ein *nicht benannter Graph* ist dann eine Äquivalenzklasse der benannten Graphen.

Generell ist die Prüfung zweier Graphen auf Isomorphie sehr aufwendig. Jedoch wächst die Anzahl der Graphen, auch der unbenannten, sehr schnell an. Dennoch können unbenannte Graphen anhand ihrer Nummern sehr effizient aufgelistet werden. Beispielsweise kann das Programm Nauty die 12005168 einfachen Graphen mit 10 Knoten in 20 Sekunden auflisten.

Wie in Unterabschnitt 15.4.1 ist das allgemeine Prinzip des Algorithmus die Organisation der Objekte, die numeriert werden sollen, in einem Baum, der traversiert wird. Dazu wird in jeder Äquivalenzklasse benannter Graphen (das heißt für jeden unbenannten Graphen) ein passender kanonischer Repräsentant festgelegt. Die folgenden sind die grundlegenden Operationen:

1. Testen, ob ein benannter Graph kanonisch ist;
2. Berechnen des kanonischen Repräsentanten eines benannten Graphen.

Diese unvermeidlichen Operationen bleiben aufwendig; deshalb versucht man, die Anzahl ihrer Aufrufe zu minimieren.

Die kanonischen Repräsentanten werden in der Weise bestimmt, dass es zu jedem kanonischen benannten Graphen G eine kanonische Festlegung einer Kante gibt, deren Entfernung wieder einen kanonischen unbenannten Graphen produziert, der dann der Vater von G genannt wird. Diese Eigenschaft hat zur Folge, dass die kanonischen unbenannten Graphen auf einer Menge V von Knoten als Knoten eines Baumes organisiert werden können: an der Wurzel steht der Graph ohne Kanten; darunter sein einziges Kind, der Graph mit einer Kante; dann die Graphen mit zwei Kanten usw. Die Menge der Kinder eines Graphen G kann durch *Augmentation* konstruiert werden, durch Hinzunahme jeweils einer weiteren Kante zu G, die auf alle möglichen Weisen erfolgt und nachfolgende Auswahl der Graphen, die noch kanonisch[4] sind. Rekursiv erhält man alle kanonischen Graphen.

In welchem Sinne ist dieser Algorithmus generisch? Betrachten wir z.B. planare Graphen (Graphen, die in der Ebene gezeichnet werden können, ohne dass sich Kanten kreuzen): durch Entfernung einer Kante aus einem planaren Graphen erhalten wir wieder einen planaren Graphen; somit bilden planare Graphen einen Teilbaum des vorhergehenden Graphen. Um sie zu erzeugen kann exakt derselbe Algorithmus verwendet werden, der nur die Kinder auswählt, die planar sind:

```
sage: [len(list(graphs(n, property = lambda G: G.is_planar())))
....:  for n in range(7)]
[1, 1, 2, 4, 11, 33, 142]
```

Auf ähnliche Weise kann man jede abgeschlossene Famile von Graphen durch Entfernung von Kanten erzeugen und insbesondere jede Famile, die durch verbotene Teilgraphen charakterisiert ist. Das schließt beispielsweise Wälder ein (Graphen ohne Schleifen), bipartite Graphen (Graphen ohne ungerade Schleifen) usw. Mit diesem Ansatz können auch erzeugt werden

[4] In der Praxis würde eine effiziente Implementierung die Symmetrien von G ausnutzen, d.h. seine automorphe Gruppe, um die Anzahl der zu untersuchenden Kinder zu reduzieren und den Aufwand für jeden Test, ob es kanonisch ist.

- partielle Ordnungen über die Bijektion mit Hasse-Diagrammen, gerichtete Graphen ohne Schleifen und ohne Kanten, die durch die Transitivität der Ordnungsrelation impliziert werden;
- Gitter (in Sage nicht implementiert), über die Bijektion mit passenden Halbgittern, die durch Entfernen des größten Knoten erhalten werden; in diesem Fall wird eine Augmentation der Knoten, nicht der Kanten vorgenommen.

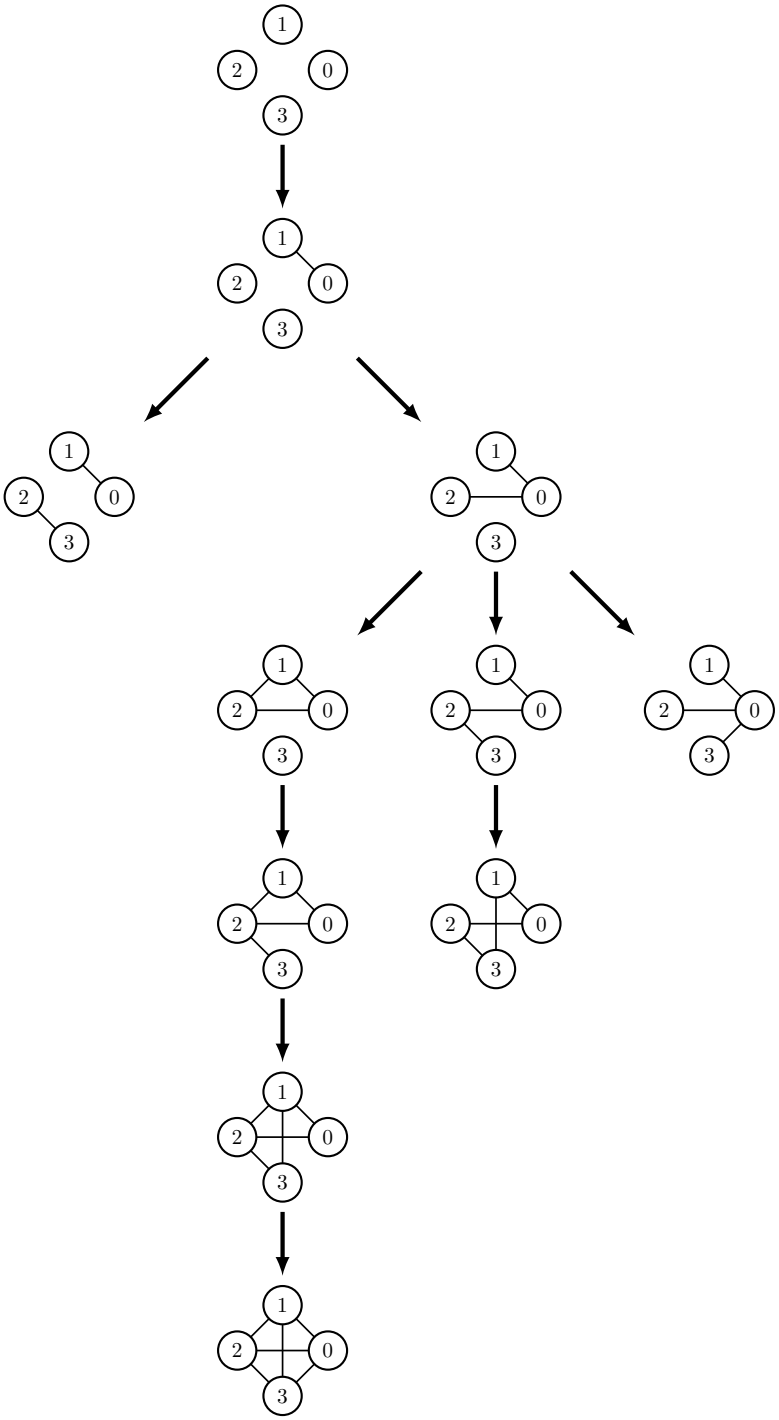


Abb. 15.6 - Der Baum der einfachen Graphen mit 4 Knoten.

16. Graphentheorie

Dieses Kapitel bietet das Studium der Graphentheorie mit Sage. Zunächst wird die Klasse Graph beschrieben (Abschnitt 16.1) wie auch ihre Methoden (Abschnitt 16.2). Dann werden sie zur Lösung praktischer Probleme angewendet (Abschnitt 16.4) bzw. zur experimentellen Verifizierung theoretischer Resultate (Abschnitt 16.3)

16.1. Erzeugen eines Graphen

16.1.1. Beginn bei Null

Hier definieren wir Graphen als Paare (V, E), wobei V eine Menge von Knoten darstellt (*vertices* im Englischen) und E eine Menge von Kanten (*edges* im Englischen). Der Graph in Abb. 16.1 ist auf der Menge der Knoten $\{0, 1, 2, 5, 9, \text{'Madrid'}, \text{'Edinburgh'}\}$ definiert und hat $(1, 2), (1, 5), (1, 9), (2, 5), (2, 9)$ sowie ('Madrid', Edinburgh) als Kanten.

In Sage werden Graphen - kaum überraschend - durch die Klasse `Graph` dargestellt:

```
sage: g = Graph()
```

Voreingestellt ist, dass der Graph g zunächst leer ist. Das folgende Beispiel illustriert, wie ihm Knoten und Kanten hinzugefügt werden: Sobald eine Kante erzeugt worden ist, werden die zugehörigen Knoten - wenn sie nicht schon im Graphen enthalten sind - stillschweigend eingefügt. Wir kontrollieren den Ablauf der Prozedur mit Methoden, deren Rollen leicht zu erraten sind:

```
sage: g.order(), g.size()
(0,0)
sage: g.add_vertex(0)
 sage: g.oder(), g.size()
(1,0)
sage: g.add_vertices([1,2,5,9])
 sage: g.order(), g.size()
(5,0)
sage: g.add_edges([(1,5),(9,2),(2,5),(1,9)])
 sage: g.order(), g.size()
(5,4)
sage: g.add_edge('Madrid','Edinburgh')
 sage: g.order(), g.size()
(7,5)
```

Das Hinzufügen der Kante $(1, 2)$ bedeutet dasselbe wie das Hinzufügen der Kante $(2, 1)$. Man sieht außerdem, dass die Methoden `add_vertex` und `add_edge` beide einen „Plural“ (`add_vertices` und `add_edges`) haben, dem jeweils eine Liste als Argument übergeben wird, was eine kompaktere Schreibweise erlaubt (siehe z.B. Unterabschnitt 16.4.1, wo wir einen Graph erzeugen, nachdem wir seine Kanten generiert haben).

Generell achtet Sage nicht auf den Typ der Objekte, die als Knoten eines Graphen verwendet werden können. Sage akzeptiert in der aktuellen Version 7 jedes nicht mutable Python-Objekt (und daher auch keine Objekte der Klasse `Graph`), d.h. alles was ein Diktionär als Schlüssel akzeptieren würde (siehe Unterabschnitt 3.3.9). Es ist dann selbstredend auch möglich, hinzugefügte Elemente mit Hilfe der Methoden `delete_*` wieder zu entfernen und die Knoten oder Kanten aufzuzählen, über die wir häufig iterieren.

```
sage: g.delete_vertex(0)
sage: g.delete_edges([(1,5), (2,5)])
sage: g.order(), g.size()
(6,3)
sage: g.vertices()
[1,2,5,9,'Edinburgh', 'Madrid']
sage: g.edges()
[(1, 9, None), (2, 9, None), ('Edinburgh', 'Madrid', None)]
```

Für Sage sind Kanten in Wahrheit Tripel, deren letzter Eintrag ein Bezeichner (*label*) ist. In ihm werden meistens numerische Werte gespeichert - die zum Beispiel von Algorithmen für den Fluss oder den Zusammenhang als Kapazitäten oder als Gewichtung für das Problem der Kopplung (*matching*) interpretiert werden - und kann auch ein nicht mutables Objekt enthalten. Voreingestellt ist der Wert `None`.

Sind die Knoten und Kanten eines Graphen im voraus bekannt, kann man ihn auf kompakte Weise mit einem Diktionär erzeugen, das jedem Knoten die Liste seiner Nachbarn zuordnet:

```
sage: g = Graph({
....:     0 : []
....:     1 : [5,9]
....:     2 : [1,5,9],
....:     'Edinburgh' : ['Madrid']
```

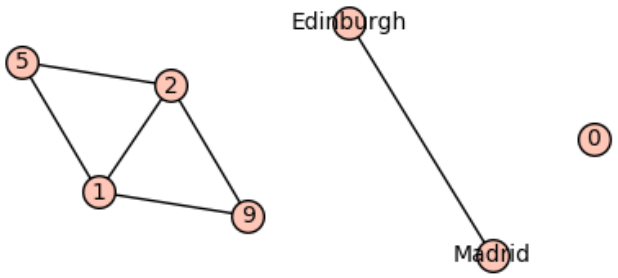


Abb. 16.1 - Ein Graph, dessen Knoten Zahlen sind oder Zeichenketten

Auch dort könnte man sich erlauben, die Zeilen zu vergessen, die den Knoten 5, 9 oder `'Madrid'` entsprechen, die ja schon als Nachbarn anderer Knoten aufgeführt worden sind. Ebenso könnte man sagen, dass 1 ein Nachbar von 2 ist, obwohl 2 nicht in der Liste der Nachbarn von 1 erscheint: die Kante (1,2) wird gleichwohl erzeugt.

Übung 60. *(Zirkulierende Graphen).* Der zirkulierende Graph mit den Parametern n, d ist ein Graph mit n Knoten, die von 0 bis $n-1$ numeriert sind (die wir uns auf einem Kreis angeordnet denken können), sodass zwei Knoten u und v dann durch eine Kante verbunden sind, wenn $u \equiv v + c \mod n$ ist und $-d \leq c \leq d$. Schreiben Sie eine Methode, der die Parameter n und d übergeben werden und die den zugehörigen Graphen zurückgibt.

16.1.2. Verfügbare Konstruktoren

Den vorstehehenden Illustrationen zum Trotz wird in Sage selten auf eine Adjazenztabelle zugegriffen oder gar manuell auf Kante nach der anderen, um einen Graphen zu erstellen. Meistens ist es effizienter, sie mit Hilfe bereits vordefinierter Elemente zu erstellen: die Methoden `graphs.*` erlauben die Erstellung von über 70 Graphen oder Graphenfamilien, die wir jetzt vorstellen. Die Graphen von Chvatal und Petersen beispielsweise erhält man in Sage mit folgenden Zeilen:

```
sage: P = graphs.PetersenGraph()
sage: C = graphs.CvatalGraph()
```

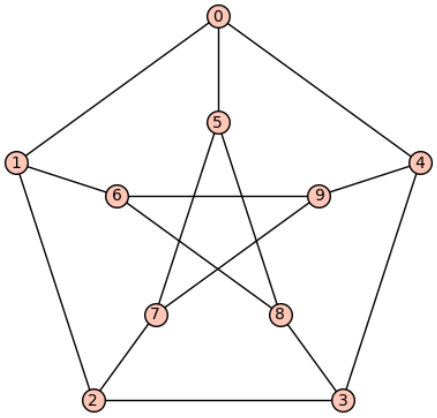
Beginnen wir die Beschreibung mit kleinen Graphen - im Gegensatz zu Graphenfamilien, denen wir später begegnen werden.

Kleine Graphen. Diese Graphen tragen zumeist den Namen ihres Entdeckers oder eines Objejektes, dem sie ähneln (ein Haus, ein Schnuller, ein Stier).

Sie erschienen oft als Gegenbeispiele zu bestimmten Vermutungen oder als kleinere Graphen, die diese oder jene Eigenschaft besaßen. Der Graph von Petersen ist zum Beispiel nicht planar: er besitzt - gleichzeitig - die beiden kleineren, vom Satz von Kuratowski verbotenen (K_5 und $K_{3,3}$). Es ist außerdem ein Graph ohne Dreieck (seine Masche - *Umfang (girth)* ist gleich 5), 3-regulär und von der chromatischen Zahl 3. Es ist auch ein knotentransitiver Graph. Jeder seiner Parameter kann von Sage mittels der entsprechenden Methoden aufgefunden werden:

	Kleine Graphen	
BullGraph	ChvatalGraph	ClawGraph
DesarguesGraph	DiamondGraph	DodecahedralGraph
FlowerSnark	FruchtGraph	HeawoodGeaph
HexahedralGraph	HigmanSimsGraph	HoffmanSingletonGraph
HouseGraph	HouseXGraph	IcosahedralGraph
KrackhardtKiteGraph	LollipopGraph	MoebiusKantorGraph
OctahedralGraph	PappusGraph	PetersenGraph
TetrahedralGraph	ThomsenGraph	

```
sage: P = graphs.PetersenGraph()
sage: P.is_planar()
False
sage: P.minor(graphs.CompleteBipartiteGraph(3,3))
{0: [0, 1, 1: [2, 7], 2: [8}, 3: [6, 9], 4: [5], 5: [3, 4]}
sage: P.minor(graphs.CompleteGraph(5))
\{0: [0, 5], 1: [4, 9], 2: [2, 7], 3: [3, 8], 4: [1, 6]}
sage: P.girth()
5
sage: P.is_regular()
True
sage: P.chromatic_number()
3
sage: P.is_vertex_transitive()
True
sage: P.show()
```



Familien von Graphen. Die hier vorgestellten Konstruktoren beschreiben Familien von Graphen, sie nehmen deshalb einen oder mehrere Parameter als Argument auf (mit einer Ausnahme, `nauty_geng`, das keine besondere Familie von Graphen beschreibt, sondern die Menge fast *aller* Graphen mit Isomorphismus - siehe Unterabschnitt 15.4.4).

Familien von Graphen	
BarbellGraph	BubbleSortGraph
CircularLadderGraph	DegreeSequence
DegreeSequenceBipartite	DegreeSequenceConfigurationModel
DegreeSequenceTree	DorogovtsevGoltsevMendesGraph
FibonacciTree	FuzzyBallGraph
GeneralizedPetersenGraph	Grid2dGraph
GridGraph	HanoiTowerGraph
HyperStarGraph	KneserGraph
LCFGraph	LadderGraph
NKStarGraph	NStarGraph
OddGraph	ToroidalGrid2dGraph
nauty_geng	

In dieser Liste findet sich eine Verallgemeinerung (eigentlich sogar zwei) des Petersengraphen: der Graph von Kneser. Dieser Graph wird ausgehend von zwei Parametern n und k erstellt, und seine Knoten sind die $\binom{n}{k}$ Teilmengen der Größe k von $\{1,\ldots,n\}$. Zwei dieser Mengen sind genau dann adjazent, wenn sie disjunkt sind. Die Kanten des Petersengraphen entsprechen den Teilmengen der Größe $k=2$ einer Menge der Größe 5:

```
sage: K = graphs.KneserGraph(5, 2); P = graphs.PetersenGraph()
sage: K.is_isomorphic(P)
True
```

Nach Konstruktion sind die Knesergraphen auch knotentransitiv. Ihre chromatische Zahl ist genau $n-2k+2$, ein überraschendes Ergebnis, das von Lovász vermittels des Satzes von Borsuk-Ulam bewiesen worden ist und somit durch topologische Betrachtungen [Mat03]. Verifizieren wir das gleich durch ein paar Beispiele:

```
sage: all(graphs.KneserGraph(n,k).chromatic_number() == n - 2*k + 2
....:     for n in range(5,9) for k in range(2,floor(n/2)))
True
```

Übung 61. *(Graphen von Kneser).* Schreiben Sie eine Funktion mit den beiden Parametern n,k, die den zugehörigen Knesergraphen ausgibt, und dies, wenn möglich, ohne „if“ zu verwenden.

Elementare Graphen. Folgende Graphen sind der „Grundbestand“, nämlich die gängigsten in der Graphentheorie (vollständige Graphen, bipartit vollständige, zirkulierende, Pfade, Kreise, Sterne usw.). Auch hier wieder eine Ausnahme: `trees`; diese Methode erlaubt die Iteration über die Menge der Bäume mit n Knoten.

Zufallsgraphen. Diese letzte Klasse von Grapohen ist sehr reich an Eigenschaften. Wir finden da unter anderem die $G_{n,p}$ und $G_{n,m}$, die beiden einfachsten Modelle von zufälligen Graphen.

	Elementare Graphen	
BalancedTree	CirculantGraph	CompleteBipartiteGraph
CompleteGraph	CubeGraph	CycleGraph
EmptyGraph	PathGraph	StarGraph
WheelGraph		trees

Die Graphen $G_{n,p}$ werden durch eine ganze Zahl n und eine reelle Zahl $0 \leq p \leq 1$ festgelegt. Wir erhalten einen Zufallsgraphen $G_{n,p}$ mit n Knoten $\{0, \ldots, n-1\}$, indem für jedes der $\binom{n}{2}$ Knotenpaare i, j eine Münze geworfen wird - wobei ihre Wahrscheinlichkeit, auf „Zahl“ zu fallen, gleich p ist - und dem Graphen die entsprechende Kante hinzugefügt wird, wenn „Zahl“ kommt.

Wir beobachten, dass bei jedem festgelegten Graphen H die Wahrscheinlichkeit, dass $G_{n,p}$ H als induzierten Teilgraphen enthält, gegen 1 tendiert, wenn $0 < p < 1$ fest ist und n gegen unendlich geht (siehe 16.3.4). H ist ein von G induzierter Teilgraph, wenn es eine Knotenmenge $S \subseteq V(G)$ gibt, sodass die Beschränkung auf S von G (d.h. der Graph, dessen Knoten S sind und dessen Kanten diejenigen Kanten von G sind, die nur die Knoten von S berühren) ist isomorph zu H. Wir stellen auch fest, dass dieser Untergraph $G[S]$ induziert.

```
sage: H = graphs.ClawGraph()
sage: def test():
....:     g = graphs.RandomGNP(20,2/5)
....:     return not g.subgraph_search(H, induced=True) is None
sage: sum(test() for i in range(100)) >= 80
True
```

16.1.3. Disjunkte Vereinigungen

Zusätzlich zu diesen Grundbausteinen erlaubt Sage mit Hilfe von zwei einfachen, aber effizienten Operationen die Vereinigung von disjunkten Graphen. Die *Addition von zwei Graphen* entspricht ihrer disjunkten Vereinigung.

```
sage: P = graphs.PetersenGraph()
sage: H = graphs.HoffmanSingletonGraph()
sage: U = P + H; U2 = P.disjoint_union(H)
sage: U.is_isomorphic(U2)
True
```

Das *Produkt* eines Graphen G mit einer ganzen Zahl k gibt die disjunkte Vereinigung von k Kopien von G zurück:

```
sage: C = graphs.ChvatalGraph()
sage: U = 3*C; U2 = C.disjoint_union(C.disjoint_union(C))
sage: U2.is_isomorphic(U)
True
```

Die folgende Zeile erzeugt die disjunkte Vereinigung von drei Kopien des Petersengraphen und zwei Kopien des Graphen von Chvatal:

```
sage: U = 3*P + 2*C
```

	Zufallsgraphen	
DegreeSequenceExpected	RandomBarabasiAlbert	RandomBipartite
RandomGNM	RandomGNP	RandomHolmeKim
RandomInterval	RandomLobster	RandomNewmanWattsStrogatz
RandomRegular	RandomShell	RandomTreePowerlaw

Es gibt viele Möglichkeiten, dieses Ergebnis zu erzielen, die alle Ausflüchte dafür sind, ein paar Zeilen Code zu schreiben. Beispielsweise dadurch, dass sichergestellt wird, dass jede Verbindung mit einem der beiden Graphen isomorph ist:

```
sage: all((CC.is_isomorphic(P) or CC.is_isomorphic(C))
....:       for CC in U.connected_components_subgraphs())
True
```

oder durch Berechnung der genauen Anzahl von Untergraphen:

```
sage: sum(CC.is_isomorphic(P)
....:     for CC in U.connected_components_subgraphs())
3 sage: sum(CC.is_isomorphic(C)
....:     for CC in U.connected_components_subgraphs())
2
```

Technische Details. Es muss gesagt werden, dass die Operationen Addition und Multiplikation hinsichtlich des Aufwands an Speicherplatz und Zeit *Kopien* sind. Das kann zuweilen Quelle von Verlangsamung werden. Andererseits wird die Modifikation von P oder C nach vorstehendem Beispiel U nicht verändern. Mehr noch: bei diesen Operationen gehen zwei Informationen verloren:

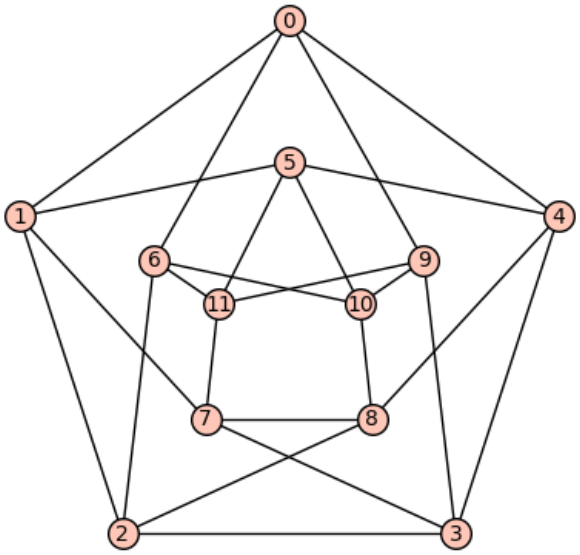
- die Knoten des Endgraphen werden mit ganzen Zahlen $\{0, 1, 1, \ldots\}$ umnumeriert,
- die Positionen der Knoten (im Plan) bleiben in U nicht erhalten.

Die Methode `disjoint_union` zeigt ein anderes Verhalten: wenn der Graph g einen Knoten a enthält und der Graph h einen Knoten b, enthält der von `g.disjoint_union(h)` zurückgegebene Graph die Knoten $(0, a)$ und $(1, b)$. Für den Fall, dass a oder b nicht ganzzahlig sind sondern andere Objekte (Zeichenkette, Tupel), dann vereinfacht die Anwendung dieser Methode die Arbeit mit dem resultierenden Graphen erheblich.

16.1.4. Ausgabe von Graphen

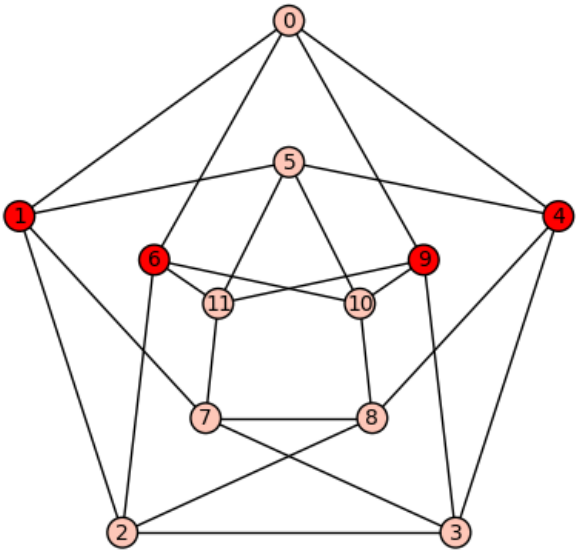
Ein sehr willkommener Aspekt des Studiums von Graphen in Sage ist die Möglichkeit, sie zu visualisieren. Ohne Varianten oder Schnörkel genügt ein einziger Befehl:

```
sage: C = graphs.ChvatalGraph(); C.show()
```



Dabei handelt es sich um ein präzises Werkzeug zur Visualisierung bestimmter Funktionen. Hier lassen wir eine unabhängige Menge zeichnen:

```
sage: C.show(partition = [C.independent_set()])
```

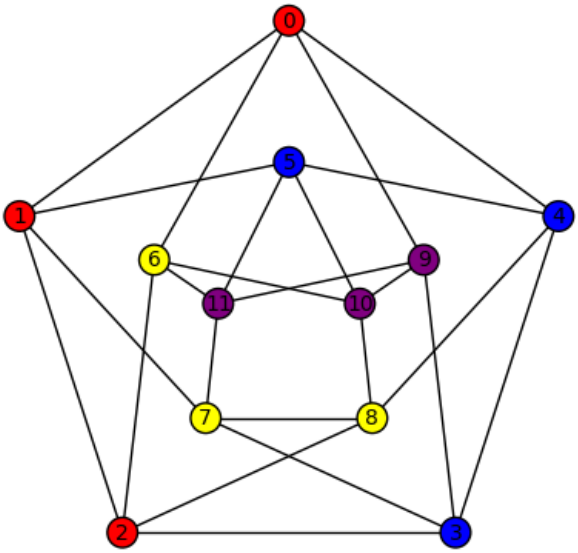


Dem Argument `partition` der Methode `show` wird, wie der Name sagt, eine Partition der Knotenmenge, eine Zerlegung also, übergeben. Eine Farbe wird dann jeder Menge (hier nur eine: rot) der Partition zugewiesen, um sie sichtbar zu unterscheiden. Eine letze Farbe (hier:

rosa) wird dann denjenigen Knoten zugewiesen, die nicht in der Partition erscheinen. In unserem Beispiel haben wir somit insgesamt zwei Farben.

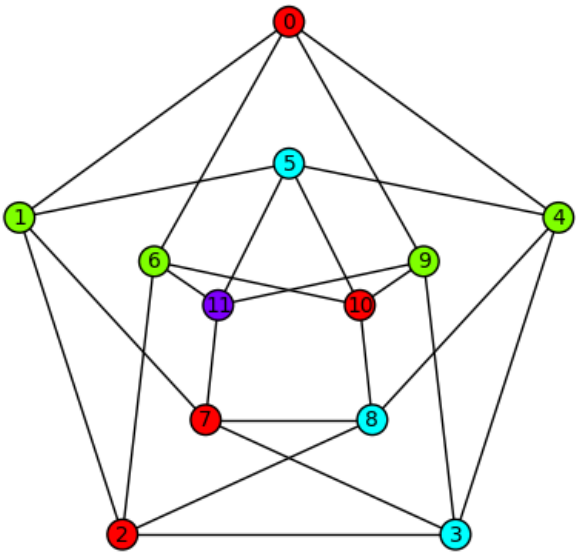
Es ist selbstverständlich möglich, die Farben selbst festzulegen, die man den Knoten geben möchte, nämlich mittels eines Diktionärs und einer einigermaßen natürlichen Syntax:

```
sage: C.show(vertex_colors = {
....:   "red" : [0,1,2],     "blue" : [3,4,5],
....:   "yellow" : [6,7,8], "purple" : [9,10,11]})
```



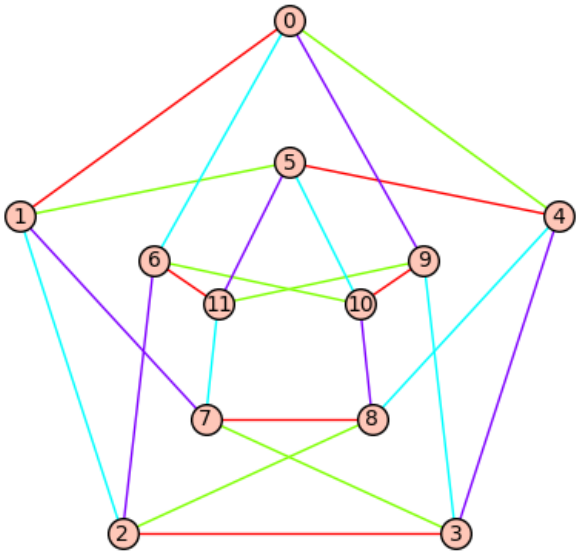
Zuweilen wünscht man sich keine Grundfarben oder Mischfarben 1. Grades. Man kann die Farben dann mit ihrem Hexadezimalcode festlegen, wie das auch bei HTML der Fall ist. Methoden wie `coloring` sind genau dafür gemacht:

```
sage: C.coloring(hex_colors = True)
{'#00ffff': [3,8,5],  '#7f00ff': [11],
 '#ff0000': [0,2,7,10], '#7fff0000': [1,4,6,9]}
```



Genauso verhält sich das Argument `edge_colors`:

```
sage: from sage.graphs.graph_coloring import edge_coloring
sage: edge_coloring(C, hex_coloring = True)
{#00ffff}: [(0,6),(1,2),(3,9),(4,8),(5,10),(7,11)]
 #7f00ff}: [(0,9),(1,7),(2,6),(3,4),(5,11),(8,10)]
 #ff0000}: [(0,1),(2,3),(4,5),(6,11),(7,8),(9,10)]
 #7fff00}: [(0,4),(1,5),(2,8),(3,7),(6,10),(9,11)]
sage: C.show(edge_colors = edge_coloring(C, hex_colors = True))}
```



Bilder exportieren. Es ist auch möglich, die von Sage erzeugten Graphen einzeln zu exportieren. Das folgende Beispiel erzeugt vollständige Graphen mit $3, 4, \ldots, 12$ Knoten und speichert sie in den Dateien `graph0.png, ..., graph9.png`.

```
sage: L = [graphs.CompleteGraph(i) for in range(3,3+10)]
sage: for number, G in enumerate(L):
....:     G.plot().save('/tmp'/ + 'graph' + str(number) + '.png')
```

Für die Befehle `show` und `plot` gibt es keine ausführliche Dokumentation. Wir erwähnen immerhin die Option `figsize = 15`, mit der die Auflösung des Bildes bestimmt wird. Sie erweist sich bei großformatigen Graphen als nützlich.

16.2. Methoden der Klasse Graph

Die Klasse `Graph` verfügt über mehr als 250 Methoden, wenn man diejenigen beiseite lässt, die exclusiv in der Klasse `DiGraph` definiert sind, wo sie nur in angehängten Modulen erscheinen. Das macht Sage zu einer ausdrucksstarken und vollständigen Bibliothek der Graphentheorie, die es erlaubt, sich auf das Wesentliche zu konzentrieren, d.h. es wird vermieden, vor den Funktionen, die uns interessieren erst noch andere Funktionen programmieren zu müssen.

Wie beim Erlernen jeder Programmiersprache (oder Bibliothek) ist es hilfreich, wenigstens einmal die Liste ihrer Funktionen durchzugehen, um zu erkunden, was sie alles kann. Dieser

(nicht erschöpfende) Abschnitt versucht, sie ..... vorzustellen. Dem Leser sei geraten, die paar Minuten zu opfern, die für die Lektüre des Inhalts jeder Kategorie nötig sind - *und auch die vollständige Liste der verfügbaren Methoden:* die dafür benötigte Zeit wird sich angesichts eines Graphenproblem schließlich als gute Investition erweisen! Es ist ebenfalls ratsam, eine offene Sagesitzung vor sich zu haben, um die Dokumentation der dargestellen Funktionen einsehen zu können (z.B. `g.degree_constrained_subgraph?`), von denen manche mehrere Optionen haben oder wenn ihr Name nicht hinreichend aussagekräftig ist.

16.2.1. Modifikation der Struktur von Graphen

Natürlich enthält ein Großteil der Klasse `Graph` Methoden, die mit der Definition und der Modifikation von Graphen zu tun haben, die erforderlich sind, obwohl sie keine merkliche Funktionalität hinzufügen.

Methoden für den Zugriff und die Modifikation der Klasse Graph		
`add_cycle`	`add_edge`	`add_edges`
`add_path`	`add_vertex`	`add_vertices`
`adjacency_matrix`	`allow_loops`	`allow_multiple_edges`
`allows_loops`	`allows_multiple_edges`	`clear`
`delete_edge`	`delete_edges`	`delete_multiedge`
`delete_vertex`	`delete_vertices`	`edge_iterator`
`edge_label`	`edge_labels`	`edges`
`edges_incident`	`get_vertex`	`get_vertices`
`has_edge`	`has_loops`	`has_multiple_edges`
`has_vertex`	`incidence_matrix`	`latex_options`
`loop_edges`	`loop_vertices`	`loops`
`merge_vertices`	`multiple_edges`	`name`
`neighbor_iterator`	`neighbors`	`networkx_graph`
`num_edges`	`num_verts`	`number_of_loops`
`orde`	`relabel`	`remove_loops`
`remove_multiple_edges`	`rename`	`reset_name`
`save`	`set_edge_label`	`set_latex_options`
`set_vertex`	`set_vertices`	`size`
`subdivide_edge`	`subdivide_edges`	`vertex_iterator`
`vertices`	`weighted`	`weighted_adjacency_matrix`

16.2.2. Operatoren

Ebenfalls zu den Methoden zählen die *Operatoren*, die ein Objekt des Typs `Graph` (oder `DiGraph`) zurückgeben. Wird beispielsweise die Methode `complement` auf G angewendet, gibt sie einen Graphen mit denselben Knoten zurück, dessen Kante uv nur existiert, wenn $uv \notin G$ ist. Die Methode `subgraph` erlaubt ihrerseits, zu einem Graphen G den von einer gegebenen Knotenmenge *induzierten* Untergraphen zu erhalten, eine Operation, die $G[\{v_1, \ldots, v_k\}]$ geschrieben wird.

Verifizieren wir einige elementare Relationen. Das Komplement von P_5 (geschrieben $\bar{P}_5$) ist ein Haus, und die Graphen P_4 und C_5 sind autocomplementär:

```
sage: P5 = graphs.PathGraph(5); Haus = graphs.HouseGraph()
sage: P5.complement().is_isomorphic(Haus)
True
sage: P4 = graphs.PathGraph(4); P4.complement().is_isomorphic(P4)
True
sage: C5 = graphs.CycleGraph(5); C5.complement().is_isomorphic(C5)
True
```

Sage definiert auch (mit der gleichnamigen Methode) den *Kantengraphen* (line graph) von G - oft mit $L(G)$ bezeichnet - dessen Knoten den Kanten von G entsprechen und bei dem zwei Knoten adjazent sind, wenn die entsprechenden Kanten in G inzident sind. Wir finden auch die Definition verschiedener Produkte von Graphen. In jedem der folgenden Beispiele nehmen wir an, dass G das Produkt von G_1 und G_2 ist, das auf der Knotenmenge $V(G_1) \times V(G_2)$ definiert ist. Zwei Knoten $(u, v), (u', v')$ sind genau dann adjazent, wenn:

Kartesisches Produkt
`cartesian_product`
oder $\begin{cases} u = u' \text{ und } vv' \in E(G_2) \\ uu' \in E(G_1) \text{ und } v = v' \end{cases}$

Lexikographisches Produkt
`lexicographic_product`
oder $\begin{cases} uu' \in E(G_1) \\ u = u' \text{ und } vv' \in E(G_2) \end{cases}$

Disjunktes Produkt
`disjunctive_product`
oder $\begin{cases} uu' \in E(G_1) \\ vv' \in E(G_2) \end{cases}$

Tensorprodukt
`tensor_product`
und $\begin{cases} uu' \in E(G_1) \\ vv' \in E(G_2) \end{cases}$

Starkes Produkt
`strong_produkt`
oder $\begin{cases} u = u' \text{ und } vv' \in E(G_2) \\ uu' \in E(G_1) \text{ und } v = v' \\ uu' \in E(G_1) \text{ und } vv' \in E(G_2) \end{cases}$

Wir können das quadratische Gitter `GridGraph` als cartesisches Produkt von zwei Pfaden bilden:

```
sage: n = 5; Path = graphs.PathGraph(n)
sage: Grid = Path.cartesian_product(Path)
sage: Grid.is_isomorphic(graphs.GridGraph([n,n]))
True
```

Produkte, Operatoren		
`cartesian_produkt`	`categforial_product`	`complement`
`copy`	`disjoint_union`	`disjunctive_product`
`kirchhoff_matrix`	`laplacian_matrix`	`line_graph`
`lexicographic_product`	`strong_product`	`subgraph`
`to_directed`	`to_simple`	`to_undirected`
`tensor_product`	`transitive_closure`	`transitive_reduction`
`union`		

16.2.3. Traversierung von Graphen und Abstände

Sage bietet die üblichen Methoden zur Traversierung von Graphen wie die Tiefensuche (`depth_first_search`) und die Breitensuche (`breadth_first_search`). Das sind grundlegende Routinen für die Abstandsberechnung, den Fluss oder die Konnektivität. Sage enthält auch eine Implementierung des weniger klassischen `lex_BFS` (*lexicographic breadt_first search*), die zum Beispiel zum Erkennen triangulierter (cordaler) Graphen (vgl. `is_chordal`). Diese Methoden geben Listen von Knoten zurück, deren Ordnung der Reihenfolge ihres Auffindens entspricht[1]:

```
sage: g = graphs.RandomGNP(10, .6)
sage: list(g.depth_first_search(0))
[0, 6, 9, 8, 4, 7, 3, 2, 1, 5]
sage: list(g.breadth_first_search(0))
[0, 1, 6, 2, 3, 5, 8, 9, 4, 7]
sage: g.lex_BFS(0)
[0, 1, 6, 3, 8, 9, 5, 2, 4, 7]
```

Mit Hilfe dieser Traversierungen definieren wir die Methode `shortest_path`, die wahrscheinlich die meistbenutzte von allen ist[2]. Sage erlaubt gleichfalls die Berechnung verschiedener Invarianten, die mit Abständen zu tun haben:

- `eccentricity`: ordnet einem Knoten v den maximalen Abstand zu allen anderen Knoten des Graphen zu;
- `center`: gibt einen *zentralen* Knoten des Graphen zurück - sozusagen die kleinste Exzentrizität;
- `radius`: liefert die Exzentrizizät eines zentralen Knotens;
- `diameter`: gibt den größten Abstand zwischen zwei Knoten zurück;
- `periphery`: gibt die Liste der Knoten zurück, deren Exzentrizität gleich dem Durchmesser ist.

Abstände, Traversierung		
`average_dstance`	`breadth_first_search`	`center`
`depth_first_search`	`diameter`	`distance`
`distance_all_pairs`	`distance_graph`	`eccentricity`
`lex_BFS`	`periphery`	`radius`
`shortest_path`	`shortest_path_all_pairs`	`shortst_path_length`
`shortest_path_length`	`shortest_paths`	

[1] Wenn `lex_BFS` eine Knotenliste zurückgibt, sind die Methoden `depth_first_search` und `breadth_first_search` die Iteratoren über die Knoten, daher die Verwendung der Methode `list`

[2] Es ist anzumerken, dass `shortest_path` nicht unbedingt `breadth_first_search` aufruft: sind die Kanten mit Abständen gewichtet, wird es von Implementierungen des Algorithmus von Dijkstra (Standard oder bidirektional) abgelöst.

16.2.4. Fluss, Konnektivität, Paarung (Matching)

Mit Hilfe der Methode `flow` (vgl. Unterabschnitt 17.4.3) kann Sage Probleme des größten Flusses lösen[3]. Dank der vorstehend erwähnten Traversierung enthält Sage auch mehrere Methoden, die mit dem Zusammenhang zu tun haben (`is_connected, edge_connectivity, vertex_connectivity, connected_components,...`) wie auch mit Konsequenzen aus dem Satz von Menger:

Seien G ein Graph und u,v zwei seiner Knoten. Die folgenden Aussagen sind äquivalent:

- *der Wert des maximalen Flusses zwischen u und v ist k (siehe* `flow`);
- *es existieren k (aber nicht k+1) kantendisjunkte Pfade zwischen u und v (siehe* `edge_disjoint_paths`);
- *es existiert eine Menge von k Kanten in G, durch deren Entfernung aus dem Graphen entfernt u und v getrennt werden (siehe* `edge_cut`).

Gegenstücke dieser Methoden hinsichtlich des Zusammenhangs der Knoten sind `flow` (dank seiner Option `vertex_bound=True`), `vertex_cut` und `vertex_disjoint_paths`.

Wir verifizieren, dass der Zusammenhang eines Zufallsgraphen $G_{n,p}$ mit (sehr) großer Wahrscheinlichkeit seinem minimalen Grad gleicht:

```
sage: n = 30; p = 0.3; trials = 50
sage: def equality(G):
....:     return G.edge_connectivity() == min(G.degree())
sage: sum(equality(graphs.RandomGNP(n,p)) for i in range(trials))/trials
1
```

Wir können auch die Zerlegung eines Graphen in zweifache Knoten-Zusammenhangskomponenten erhalten oder auch seinen Baum von Gomory-Hu durch `blocks_and_cut_vertices` bzw. `gomory_hu_tree`.

Da es sich um eine grundlegende Funktion der Graphentheorie handelt, erwähnen wir hier auch die Methode `matching`, die mit dem Algorithmus von Edmonds eine maximale Paarbildung bewirkt. Die Paarbildung wird auch in den Unterabschnitten 16.4.2 und 17.4.2 angesprochen.

Flüsse, Zusammenhang,...	
`blocks_and_cut_vertices`	`connected_component_containing_vertex`
`connected_components`	`connected_components_number`
`connected_components_subgraph`	`degree_constraint_subgraph`
`edge_boundary`	`edge_connectivity`
`edge_cut`	`edhe_disjoint_paths`
`edge_disjoint_spanning_trees`	`flow`
`gomory_hu_tree`	`is_connected`
`matching`	`multicommodity_flow`
`vertex_boundary`	`vertex_connectivity`
`vertex_cut`	`vertex_disjoint_paths`

[3] Es stehen noch zwei andere Methoden zur Verfügung: die erste folgt dem Algorithmus von Ford-Fulkerson, die andere ist ein lineares Programm.

16.2.5. NP-vollständige Probleme

Für bestimmte NP-vollständige Aufgaben enthält Sage Algorithmen für eine exakte Lösung. Wohlgemerkt können dieses Aufgaben je nach Fall lange Rechenzeiten benötigen, doch haben reale Beispiele oft die angenehme Eigenschaft einfacher zu sein als die theoretischen Gegenbeispiele. Beispielsweise ist es möglich, folgende Optimierungsaufgaben zu lösen.

Cliquen und maximale unabhängige Mengen. Eine maximale Clique eines Graphen ist eine Menge von paarweise adjazenten Knoten maximaler Kardinalität (nicht adjazent im Falle einer unabhängigen Menge). Eine Anwendung dieses Aufgabentyps wird in Unterabschnitt 16.4.1 vorgestellt. Der verwendete Algorithmus ist der des Programms Cliquer [NO].
Methoden: `clique_maximum, independent_set`

Färbung von Knoten und Kanten. Eine echte Färbung der Knoten eines Graphen besteht darin, je zwei adjazenten Knoten verschiedene Farben zu geben. Sage verfügt über mehrere Funktionen zur exakten Berechnung bestimmter Farbgebungen, wobei meistens lineare Programmierung oder der Algorithmus *Dancing Links* verwendet wird. Die Erklärung eines einfachen, aber nicht optimalen Färbealgorithmus findet der Leser im Unterabschnitt 16.3.1.
Methoden: `chromatic_number, coloring, edge_coloring, grundy_coloring`[4].

Führende Menge. Eine Knotenmenge S eines Graphen G heißt *dominierend*, wenn jeder Knoten v von G Nachbar eines Elementes von S ist - wir sagen dann, dass sie *führend* ist - oder wenn er selbst Element von S ist. Da die Menge aller Knoten trivialerweise dominierend ist, besteht die Aufgabe darin, die Größe der Menge S zu minimieren. Dieses Problem wird in Sage mit Hilfe der linearen Programmierung gelöst.
Methode: `dominating_set`

Hamiltonkreis, Problem des Handlungsreisenden. Ein Graph heißt *hamiltonisch*, wenn er einen Kreis besitzt, der jeden Knoten genau einmal passiert. Im Gegensatz zum Problem des *Euler-Kreises* - hier muss der Kreis genau einmal durch jede Kante von G gehen - ist dieses Problem NP-vollständig und wird in Sage mit linearer Programmierung wie ein Spezialfall des *Problems des Handlungsreisenden* gelöst.

Wir illustrieren die Funktion `hamiltonian_cycle`, die einen Hamilton-Kreis zurückgibt, falls ein solcher Kreis existiert (Abb. 16.2):

```
sage: g = graphs.ChvatalGraph(); cycle = g.hamiltonian_cycle()
sage: g.show(vertex_labels = False); cycle.show(vertex_labels = False)
```

Methoden: `is_hamiltonian, hamiltonian_cycle, traveling_salesman_problem`

[4] Diese Methoden sind über die Klasse `Graph` nicht unmittelbar zugänglich. Um auf sie wie auch auf andere Färbefunktionen zuzugreifen, konsultieren Sie das Modul `graph_coloring`

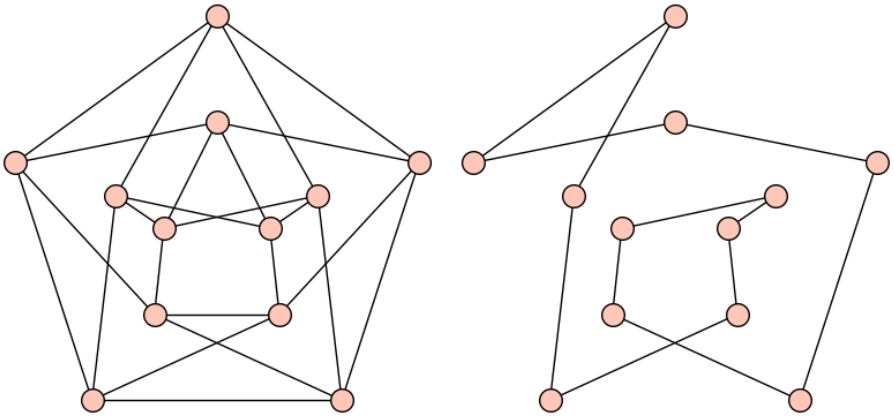


Abb. 16.2 - Der Graph von Chvatal und einer seiner Hamilton-Kreise

NP-vollständige Probleme (oder angeglichene)	
`automorphism_group`	`characteristicc_polynomial`
`chromatic_number`	`chromatic_polynomial`
`coloring`	`disjoint_routed_paths`
`edge_coloring`	`dominating_set`
`genus`	`hamiltonian_cycle`
`independent_set_of_representatives`	`is_hamiltonian`
`is_isomorphic`	`max_cut`
`minor`	`multicommodity_flow`
`multiway_cut`	`subgraph_search`
`traveling_salesman_problem`	`steiner_tree`
`vertex_cover`	

Verschiedene Probleme. Sage kann auch das *Geschlecht* (`genus`) eines Graphen berechnen. die *maximalen Schnitte* (`max_cut`), *Steiner-Bäume* (`steiner_tree`) usw. Es erlaubt auch die Lösung von wichtigen Problemen wie *Mehrfachflüsse* (`multicommodity_flow`), der Test auf Existenz von *Minoren* (`minor` - Suche eines isomorphen Minoren zu einem gegebenen Graphen) oder die Suche nach Subgraphen (`subgraph_search`).

Obwohl die theoretische Komplexität dieser Probleme noch nicht bekannt ist, sind doch Algorithmen für das Problem des Isomorphismus von Graphen `is_isomorphic` sowie auch für die Berechnung von Gruppen von Automorphismen von Graphen (`automorphism_group`) verfügbar.

16.2.6. Erkennen und Prüfen von Eigenschaften

Viele NP-vollständige Probleme besitzen einen effizienten Lösungsalgorithmus (linear, quadratisch...), wenn die Graphen zu einer bestimmten Klasse gehören. Es ist beispielsweise trivial, ein Problem der maximalen Clique auf einem chordalen (triangulierten) Graphen zu lösen, und es ist polynomial (wenn auch kompliziert), eine optimale Färbung der Knoten eines vollkommenen Graphen zu berechnen. Sage besitzt für einige elementare Klassen von Graphen Algorithmen zur Erkennung: Wälder (`is_forest`), Bäume (`is_tree`), bipartite Graphen (`is_bipartite`), eulersche Graphen (`is_eulerian`), reguläre Graphen (`is_regular`) usw. Es ist ebenfall möglich, die folgenden Klassen zu identifizieren.

Chordale Graphen. Ein Graph heißt *chordal*, wenn er keinen Kreis als induzierten Subgraph zulässt, der größer ist als 4. Auf dieselbe Weise kann jeder chordale Graph durch sequentielle Wegnahme von Knoten zerlegt werden, deren Nachbarschaft ein vollständiger Graph ist (diese Reihenfolge der Zerlegung wird *Ordnung der vollständigen Elimination* genannt). Erkannt werden diese Graphen mittels einer lexikographischen Breitensuche (`lex_BFS`).
Methode: `is_chordal`

Intervallgraphen. Sei $\mathcal{I} = \{I_1, \ldots, I_n\}$ eine endliche Menge von Intervallen von Mengen reeller Zahlen. Wir definieren für $\mathcal{I}$ einen Graphen G auf n Knoten, dessen Knoten i und j genau dann adjazent sind, wenn die Intervalle I_i und I_j eine nichtleere Schnittmenge haben. Die Graphen der Intervalle sind diejenigen, die wir auf diese Weise erhalten können. Sie bilden eine Unterklasse der chordalen Graphen, welche in linearer Zeit dank der Struktur von PQ-Bäumen zu erkennen ist.
Methode: `is_interval`

Vollkommene Graphen. Ein Graph G heißt *vollkommen*, wenn für jeden induzierten Subgraph $G' \subseteq G$ die chromatische Zahl von G gleich der maximalen Größe einer Clique ist (d.h. die Gleichheit von $\chi(G') = \omega(G)$ ist gegeben). Auch wenn das Erkennen solcher Graphen ein polynomiales Problem ist, sind die bekannten Algorithmen komplex und die Implementierung von Sage verwendet einen exponentiellen Algorithmus.
Methode: `is_perfect`

Knotentransitive Graphen. Ein Graph G heißt *knotentransitiv*, wenn für jedes Knotenpaar u und v ein Isomorphismus $h : V(G) \mapsto V(G)$ so existiert, dass $h(u) = v$ ist. Obwohl die theoretische Komplexität dieses Problems noch nicht ermittelt ist, ist die in Sage verfügbare Implementation sehr effizient.
Methode: `is_vertex_transitive`

Kartesisches Produkt von Graphen. Nur bestimmte Graphen lassen sich kartesisches Produkt einer Folge von Graphen $G_1, \ldots, G_k$ ausdrücken. Es ist außerdem möglich, zu einem gegebenen zusammenhängenden Graphen die einzige Möglichkeit zu finden, ihn so mit Hilfe eines eleganten Ergebnisses der Charakterisierung zu finden, das leicht in einen polynomialen Algorithmus übersetzt werden kann.
Methode: `is_cartesian_transitive`

Zu zahlreichen Klassen von Graphen findet man außer ihrer Charakterisierung durch eine Konstruktion (die chordalen Graphen) oder eine spezielle Eigenschaft (die vollkommenen Graphen) eine äquivalente Formulierung mittels ausgeschlossener Subgraphen. Das läuft darauf hinaus zu sagen, dass ein Graph G genau dann zu einer Klasse $\mathcal{C}$ gehört, wenn in ihm Graphen aus $\{G_1, \ldots, G_k\}$ vorkommt. In diesem Fall kann man einen Erkennungsalgorithmus beschreiben, der nur auf die Existenz jedes dieses Subgraphen testet, was der Befehl `subgraph_search` *zu tun erlaubt.*

Erkennen und Prüfen von Eigenschaften		
`is_bipartite`	`is_cordal`	`is_directed`
`is_equitable`	`is_eulerian`	`is_even_hole_free`
`is_forest`	`is_interval`	`is_odd_hole_free`
`is_overfull`	`is_regular`	`is_split`
`is_subgraph`	`is_transitively_reduced`	`is_tree`
`is_triangle_free`	`is_vertex_transitive`	

16.3. Graphen in Aktion

Es ist nun an der Zeit, aus den Funktionen, die wir gefunden haben, Nutzen zu ziehen. Die folgenden Beispiele haben zur Motivation einen praktischen oder theoretischen Vorwand und sind nicht generell die beste Art und Weise, Sage zur Lösung der Probleme zu verwenden, die sie beschreiben. Oft sind sie roh und aufzählend, aber das verleiht ihnen auch ihren Charme: ihr Ziel ist natürlich das Aufstellen einer Liste von leicht und bequem zugänglichen Anwendungsbeispielen für die Verwendung der Bibliothek der Graphen von Sage - alles ist hier in der Form, nicht in der Substanz.

16.3.1. Gierige Färbung der Knoten eines Graphen

Das Färben der Knoten eines Graphen besteht darin, jedem von ihnen eine Farbe zuzuweisen (wir nehmen hier an, dass eine ganze Zahl eine Farbe bedeutet), und dies so, dass jeder Knoten eine Farbe hat, die von den Farben seiner Nachbarn verschieden ist. Das ist selbstverständlich möglich: es genügt, soviele Farben zu wählen wie der Graph Knoten hat; aus diesem Grund ist das Färbungsproblem ein Minimierungsproblem: zu einem gegebenen Graphen G die kleinste Anzahl von Farben $chi(G)$ zu finden, die die Färbung der Knoten unter der genannten Bedingung erlaubt.

Auch wenn die Berechnung der Zahl $chi(G)$ ein schwieriges Problem ist, das sich an eine beeindruckende Literatur anlehnt, wird ein Leser mit eher praktischem Blick entzückt sein zu erfahren, dass es recht schnell gelingen kann, näher am Optimum zu sein als durch Verwendung von $|V|$ Farben. Diesen schlagen wir folgenden Algorithmus vor: „Gierige Färbung der Knoten eines Graphen“.

Wir wählen einen Knoten zufällig und geben ihm die Farbe 0. Der Reihe nach wählen wir einen noch nicht gefärbten Knoten und geben ihm als Farbe die kleinste ganze Zahl, die von seinen Nachbarn nicht verwendet wird.

Dieser Algorithmus erfordert zur Erklärung nur einige Zeilen und genauso läuft es in Sage ab. Wir wenden ihn hier auf einen Zufallsgraphen an:

```
sage: n = 100; p = 5/n; g = graphs.RandomGNP(n, p)

sage: # Menge der verfügbaren Farben.
sage: # In der Mehrzahl der Fälle genügen n Farben
sage: verfuegbare_farben = Set(range(n))

sage: # Dieses Diktionär enthält die jedem
sage: # Knoten des Graphen zugeordnete Farbe
sage: farbe = {}
sage: for u in g:
....:     verbotene = Set([farbe[v] for v in g.neighbors(u)
....:                                 if v in farbe])
....:     farbe[u] = min(verfuegbare_farben - verbotene)
sage: # Anzahl der verwendeten Farben sage: max(farbe.values()) + 1
6
```

Das ist deutlich effizienter als mit 100 Farben zu arbeiten. Es ist indessen einfach, diesen Algorithmus zu verbessern, da man bemerkt, dass die Färbung eines Graphen von einer Unbekannten abhängt: der Reihenfolge, in welcher die Knoten ausgewählt werden. Es stimmt, dass wir mit diesem Algorithmus über die Reihenfolge keinerlei Kontrolle haben, weil wir uns damit begnügen, die Knoten mittels „`for u in g`“ aufzuzählen, was vor allem der Kürze des Programms zugute kommt. Das lehrreiche Kapitel 15 hat die Aufzählung von Mengen in Sage behandelt, wozu auch Permutationen gehören. Besser noch, diese Klasse beitzt eine Methode `random_element`, derer wir uns nun bedienen können.

```
sage: P = Permutations([0,1,2,3]); P.random_element()
[2, 0, 1, 3]
```

Daher werden wir versuchen, bessere Resultate zu erzielen, in dem wir die Knoten unseres Graphen 30 mal kolorieren, jeweils in der Reihenfolge wie sie die zufällige Permutation ergibt. Es ergibt sich der folgende Code, den wir auf den eben definierten Graphen anwenden:

```
sage: verfuegbare_farben = Set(range(n))

sage: anzahl_tests = 30
sage: knoten = g.vertices()
sage: P = Permutations(range(n))
sage: bessere_faerbung = {}
sage: bessere_chromatische_zahl = +oo

sage: for t in range(anzahl_tests):
....:     # Zufällige Ordnung auf den Knoten
....:     p = P.random_element()
....:     farbe = {}
....:     for i in range(g.order()):
....:         u = knoten[p[i]]
....:         verbotene = Set([farbe[v] for v in g.neighbors(u)
....:                          if v in farbe])
....:         farbe[u] = min(verfuegbare_farben - verbotene)
....:     # Aktualisierung der besseren Färbung
....:     if max(farbe.values()) + 1 < bessere_chromatische_zahl:
....:         bessere_faerbung = farbe
....:         bessere_chromatische_zahl = 1 + max(farbe.values())
```

```
sage: # Anzahl der verwendeten Farben
sage: bessere_chromatische_zahl
4
```

Ein Gewinn, immerhin! Doch war diese ganze Artillerie zur Aktualisierung des Mininimums gar nicht erforderlich. Das Umprogrammieren, wie hier ausgeführt, erübrigt sich. In Python ist die große Mehrheit der Objekte - und im vorliegenden Fall die Paare „Ganzzahl, Diktionär“ - vergleichbar, und dies in lexikographischer Ordnung (zuerst vergleichen wir die ersten Terme - eine ganze Zahl - dann den zweiten - ein Diktionär - was hier keine große Bedeutung hat). Wir schreiben unsere ersten Codezeilen um und machen eine Funktion daraus. Man kann das folgende Ergebnis erhalten:

```
sage: def gierige_faerbung(G, permutation):
....:     n = g.order()
....:     verfuegbare_farben = Set(xrange(n))
....:     knoten = g.vertices()
....:     farbe = {}
....:     for i in range(g.order()):
....:         u = knoten[permutation[i]]
....:         verbotene = Set([farbe[v] for v in g.neighbors(u)
....:                          if v in couleur])
....:         farbe[u] = min(verfuegbare_farben - verbotene)
....:     return max(farbe.values()) + 1, farbe
```

Ist diese Funktion definiert, machen wir 50 Versuche und das Auffinden des Minimums ist eine Trivialität:

```
sage: P = Permutations(range(g.order()))
sage: anzahl_farben, coloration = min(
....:     gierige_faerbung(g, P.random_element()) for i in range(50))
sage: nombre_couleurs
4
```

Um einen Graphen mit der Mindestzahl an Farben zu färben, ist die Verwendund der Methode `coloring` vorzuziehen. Da diese Aufgabe NP-vollständig ist, muss man sich mit einer längeren Rechenzeit abfinden als bei einer gierigen Färbung.

Übung 62. *(Optimale Reihenfolge bei gieriger Färbung)* Der Algorithmus der gierigen Färbung kann einen Graphen mit einem Minimum an möglichen Farben färben. Mit der Methode `coloring`, die eine optimale Färbung berechnet, ist eine Funktion zu schreiben, die eine Reihenfolge der Knoten zurückgibt, bei welcher der Algorithmus der gierigen Färbung eine optimale Färbung ausbildet.

16.3.2. Erzeugung von Graphen unter Bedingungen

Zufallsgraphen $G_{n,p}$ haben sehr interessante Zusammenhangseigenschaften. Insbesondere sind ihre minimalen Paarungen fast immer die Nachbarschaft eines Knotens: wir finden daher dort Paarungen, von denen eine der beiden verbundenen Komponenten ein eindeutiger Knoten ist. Das kann auch ungünstig aussehen: die ganze beachtliche Menge der Knoten definiert eine Paarung, deren Kardinalität angesichts der minimalen Paarung sehr groß ist. Es ist indessen

mit viel Geduld (um große Graphen zu bekommen) und wenigen Zeilen Sage möglich, etwas andere Zufallsgraphen zu erzeugen. Hier das Verfahren, das wir implementieren werden.

Seien n und k zwei Zahlen; die erste bezeichne eine Knotenzahl und die zweite einen Zusammenhang. Der Algorithmus beginnt mit einem Baum mit n Knoten, berechnet eine minimale Paarung und ihre beiden korrespondierenden Mengen. Solange diese minimale Paarung von kleinerer Kardinalität ist als $k' < k$, werden $k - k'$ aus den beiden Mengen zufällig ausgewählte Kanten hinzugefügt.

Wie vorher auch werden wir zu zwei gegebenen Mengen S und T ein Paar von Elementen $(s, \bar{s}) \in S \times \bar{S}$ erzeugen müssen. Dafür nehmen wir die Klasse `CartesianProduct` und ihre Methode `random_element`.

```
sage: n = 20; g = graphs.RandomGNP(n, 0.5)
....:    g = g.subgraph(edges = g.min_spanning_tree())

sage: while True:
....:     -, edges, [S,Sb] = g.edge_connectivity(vertices = True)
....:     cardinality = len(edges)
....:     if cardinality < k:
....:         CP = CartesianProduct(S, Sb)
....:         g.add_edges([CP.random_element()
....:                      for i in range(k - len(edges))])
....:     else: break
```

Damit ist alles gesagt.

16.3.3. Anwendung eines probabilistischen Algorithmus bei der Suche nach einer großen unabhängigen Menge

Auch wenn Sage eine Methode `Graph.independent_set` besitzt, die in einem Graphen die Suche nach einer unabhängigen Menge maximaler Größe ermöglicht (einer Menge von paarweise nicht adjazenten Knoten), hindert uns nichts, mittels amüsanter Ergebnisse der Graphentheorie unsererseits eine unabhängige Menge aufzupüren. Wir lesen beispielsweise in dem Buch *„The Probalistic Method“* [AS00], dass in jedem Graphen G eine unabhängige Menge S existiert, sodass

$$|S| \geq \sum_{v \in G} \frac{1}{d(v) + 1},$$

wobei $d(v)$ der Grad von v ist. Der Beweis dieses Ergebnisses erfolgt in folgendem Algorithmus.

Sei $n : V \to \{1, \ldots, |V|\}$ eine zufällige Bijektion, die jedem Knoten in G eine ganze Zahl eindeutig zuordnet. Ordnen wir nun dieser Funktion eine unabhängige Menge S_n zu, die als die Menge von Knoten aus G definiert ist, welche ein kleineres Bild haben als alle ihre Nachbarn (minimale Knoten). Als Formel schreiben wir dafür:

$$S_n = \{v \in G \mid \forall u\colon uv \in E(G), n(v) < n(u)\}.$$

Diese Menge ist unabhängig kraft Definition, doch wie vergewissern wir uns ihrer Größe? Tatsächlich reicht es hin, sich bei jeden Knoten zu fragen, mit welcher Häufigkeit er in der

Menge S_n erscheint. Wenn wir die Menge P der Bijektionen von V auf $\{1, \ldots, |V|\}$ betrachten, stellen wir fest

$$\begin{aligned}\sum_{n\in P} |S_n| &= \sum_{n\in P}\sum_{v\in G} \text{„1 wenn } v \text{ minimal ist für } n, 0 \text{ sonst“} \\ &= \sum_{v\in G}\left(\sum_{n\in P} \text{„1 wenn } v \text{ minimal ist für } n, 0 \text{ sonst“}\right) \\ &= \sum_{v\in G} \frac{|P|}{d(v)+1} = |P| \sum_{v\in G} \frac{1}{d(v)+1}.\end{aligned}$$

Folglich entspricht eine solche Funktion im Mittel einer unabhängigen Menge der Größe $\sum_{v\in G} \frac{1}{d(v)+1}$. Um eine Menge dieser Größe in Sage zu bekommen, verwenden wir deshalb zufällige Bijektionen aus der Klasse `Permutations`, bis wir die durch den Satz versprochene Menge erhalten:

```
sage: g = graphs.RandomGNP(40, 0.4)
sage: P = Permutations(range(g.order()))
sage: mittel = sum(1/(g.degree(v)+1) for v in g)

sage: while True:
sage:     n = P.random_element()
sage:     S = [v for v in g if all(n[v] < n[u] for u in g.neighbors(v))]
sage:     if len(S) >= mittel: break
```

16.3.4. Suchen eines induzierten Subgraphen in einem Zufallsgraphen

Wir werden uns jetzt mit Zufallsgraphen $G_{n,p}$ beschäftigen, die in Unterabschnitt 16.1.2 bei den Konstruktoren für Graphen kurz beschrieben worden sind. Wie bereits erwähnt, besitzen diese Graphen die folgende Eigenschaft:

> *Sei H ein Graph und $0 < p < 1$. Dann gilt:*
>
> $$\lim_{n\to+\infty} P\left[H \text{ ist ein von } G_{n,p} \text{ induzierter Subgraph}\right] = 1$$
>
> *was bedeutet, dass für gegebene H und p ein großer Zufallsgraph immer einen induzierten Subgraphen H enthalten wird (siehe Definition in 16.1.2).*

Sagen wir es deutlicher: sind ein Graph H und ein ein großer Zufallsgraph gegeben, ist es möglich, eine Kopie von H in $G_{n,p}$ zu finden, und zwar dadurch, dass jedem Knoten v_i von $V(H) = \{v_1, \ldots, v_k\}$ iterativ ein Repräsentant h_i zugeordnet wird, wobei jeder v_i eine „korrekte Erweiterung“ der bereits ausgewählten Knoten ist. Wir werden daher diesem Algorithmus folgen:

- ordne v_1 einen Zufallsknoten $h(v_1) \in G_{n,p}$ zu;
- ordne v_2 einen Zufallsknoten $h(v_2) \in G_{n,p}$ zu, sodass $h(v_1)$ und $h(v_2)$ in G genau dann adjazent sind, wenn v_1 und v_2 in H adjazent sind;
- ...

- nach $j < k$ Schritten haben wir jedem der v_i $(i \leq j)$ einen Repräsentanten $h(v_i) \in G_{n,p}$ so zugeordnet, dass für alle $i, i' \leq j, h(v_i)h(v_{i'}) \in E(H)$ dann und nur dann gilt, wenn $v_i v_{i'} \in E(H)$ ist. Nun ordnen wir v_{j+1} einen Zufallsknoten $h(v_{j+1})$ so zu, dass für alle $i \leq j$ dann $h(v_i)h(v_{j+1})$ bestätigt, dass $h(i)h_{j+1} \in E(G)$ genau dann gilt, wenn $v_i v_{j+1} \in E(H)$ ist.

- ...

- nach k Schritten ist der von den Repräsentanten der Knoten $v_1, \ldots, v_k$ induzierte Subgraph von $G_{n,p}$ eine Kopie von H.

PROPOSITION. Wenn n groß wird, funktioniert diese Strategie mit großer Wahrscheinlichkeit.

Beweis. Wir schreiben $H_j = H[\{v_1, \ldots, v_j\}]$ und bezeichnen mit $P[H \mapsto_{\text{ind}} G_{n,p}]$ die Wahrscheinlichkeit, dass H ein induzierter Subgraph von Gn, p ist. Die Wahrscheinlichkeit, dass H_j, doch nicht H_{j+1} ein induzierter Subgraph von $G_{n,p}$ ist, können auf folgende Weise wir grob eingrenzen:

- Ist eine Kopie von H_j in einem Graphen $G_{n,p}$ gegeben, berechnen wir die Wahrscheinlichkeit dafür, dass kein anderer Knoten die aktuelle Kopie zu einer Kopie von H_{j+1} vervollständigen kann. Ist die Wahrscheinlichkeit, dass ein Knoten akzeptiert wird

$$p^{d_{H_{j+1}}(v_{j+1})}(1-p)^{j-d_{H_{j+1}}(v_{j+1})} \geq \min(p, 1-p)^j,$$

  dann ist die Wahrscheinlichkeit, dass keiner der verbliebenen $n - j$ Knoten akzeptabel ist, höchstens

$$(1 - \min(p, 1-p)^j)^{n-j}.$$

- In unserem Graphen gibt es maximal $j!\binom{n}{j}$ verschiedene Kopien von H_j (tatsächlich $\binom{n}{j}$ Möglichkeiten, eine Menge von j Knoten auszuwählen und $j!$ Bijektionen zwischen diesen Knoten und denen in H_j).

Da $0 < p < 1$ ist, schreiben wir $0 < \epsilon = \min(p, 1-p)$; infolgedessen ist die Wahrscheinlichkeit, dass H_j ein induzierter Subgraph von $G_{n,p}$ ist, nicht aber H_{j+1}, für jedes bestimmte $j \leq k$ höchstens

$$j!\binom{n}{j}(1-\epsilon^j)^{n-j} \leq j! n^j (1-\epsilon^j)^{n-j} = o(1/n),$$

was mit wachsendem n asymptotisch gegen null geht. Wir haben schließlich

$$\begin{aligned} P[H \mapsto_{\text{ind}} G_{n,p}] &\geq 1 - P[H_2 \mapsto_{\text{ind}} G_{n,p}, H_3 \not\mapsto_{\text{ind}} G_{n,p}] \\ &\quad - P[H_3 \mapsto_{\text{ind}} G_{n,p}, H_4 \not\mapsto_{\text{ind}} G_{n,p}] \\ &\quad \ldots \\ &\quad - P[H_{k-1} \mapsto_{\text{ind}} G_{n,p}, H_k \not\mapsto_{\text{ind}} G_{n,p}] \\ P[H \mapsto_{\text{ind}} G_{n,p}] &\geq 1 - \sum_{j \leq k} j! n^j (1-\epsilon^j)^{n-j} \\ &\geq 1 - k\, o\left(\frac{1}{n}\right). \end{aligned}$$

Außerdem liefert uns dieser Beweis einen Zufallsalgorithmus, der uns die Suche nach einer Kopie des Graphen H in einem Zufallsgraphen $G_{n,p}$ ermöglicht. Auch wenn dieser Algorithmus

eine existierende Kopie von H nicht immer findet, geht die Wahrscheinlichkeit dafür doch gegen 1, wenn n gegen unendlich geht.

```
sage: def suche_induziert(H, G):
sage:     # die Abbildung von V(H) auf V(G), die wir zu definieren suchen
sage:     f = {}
sage:     # Menge der von f noch nicht verarbeiteten Knoten von G
sage:     G_verbleibend = G.vertices()
sage:     # Menge der Knoten, für die wir noch keinen Repräsentanten gefunden haben
sage:     H_verbleibend = H.vertices()
sage:     # Solange die Funktion noch nicht vollständig ist
sage:     while H_verbleibend:
sage:         v = H_verbleibend.pop(0)
sage:         # wir suchen den nächsten Knoten aus H und seine möglichen Bilder
sage:         kandidaten = [u for u in G_verbleibend if
sage:                all([.has_edge(h,v) == G.has_edge(f_h,u)
sage:                       for h, f_h in f.iteritems()])]
sage:         # wenn wir keinen finden, brechen wir ab
sage:         if not kandidaten:
sage:             raise ValueError('No copy of H has been found in G')
sage:         # wenn doch, nehmen wir den ersten davon
sage:         f[v] = kandidaten[0]
sage:         G_verbleibend.remove(f[v])
sage:     return f
```

Als Einzeiler. Um einen Graphen H in einem Graphen G zu suchen, ist es effizienter, von der Methode `Graph.subgraph_search` Gebrauch zu machen.

16.4. Einige mit Graphen modellierte Probleme

16.4.1. Ein Rätsel aus der Zeitschrift „Le Monde 2“

In der Nr. 609 von „Le monde 2“ konnte man das folgende Rätsel finden:

> *Wie groß ist die größte Menge $S \subseteq \{0, \ldots, 100\}$, die keine Paare ganzer Zahlen $i, j \in S$ enthält, für die gilt, dass $|i - j|$ ein Quadrat ist?*

Dieses Problem ist mit dem Formalismus der Graphentheorie einfach zu modellieren. Die Relation „$|i-j|$ *ist ein Quadrat*“ ist binär und symmetrisch, daher können wir damit beginnen, den Graphen auf der Menge der Knoten $\{0, \ldots, 100\}$ zu erzeugen, worin jeweils zwei Knoten adjazent (inkompatibel) sind, wenn ihre Differenz ein Quadrat ist. Dafür verwenden wir die Klasse `Subsets`, die uns im vorliegenden Fall ermöglicht, über die Teilmengen der Größe 2 zu iterieren.

```
sage: n = 100; V = range(n+1)
sage: G = Graph()
sage: G.add_edges([(i,j) for i,j in Subsets(V,2) if is_square(abs(i-j))])
```

Nachdem wir auf der Suche nach einer Mengen mit einem Maximum an „kompatiblen“ Elementen sind, können wir die Methode `independent_set` aufrufen, die eine größte Teilmenge von Elementen zurückgibt, die paarweise nicht adjazent sind.

```
sage: G.independent_set()
[4, 6, 9, 11, 16, 21, 23, 26, 28, 33, 38, 43, 50,
56, 61, 71, 76, 78, 83, 88, 93, 95, 98, 100]
```

Die Antwort ist also 24 und nicht 42. Folglich war das Rätsel aus “Le Monde 2„ nicht die große Frage „Was ist das Leben, die Welt und überhaupt„. Die Antwort darauf müssen wir noch weiter suchen.

16.4.2. Die Zuordnung von Aufgaben

Wir befinden uns nun in folgender Situation: auf einer wichtigen Baustelle sollen zehn Personen insgesamt zehn Aufgaben ausführen. Wir können jede Person mit einer Teilliste von Aufgaben verbinden, die in ihren Kompetenzbereich fallen. Wie sind die Aufgaben zu verteilen, damit alles nach den Regeln der Kunst abläuft?

Auch hier beginnen wir wieder mit der Modellierung des Problems durch einen Graphen: er sei zweiteilig, definiert auf der Vereinigungsmenge $\{w_0, \ldots, w_9\} \cup \{t_0, \ldots, t_9\}$ von Personen und Aufgaben, und wir definieren t_i als adjazent an w_j, falls w_j in der Lage ist, t_i auszuführen.

```
sage: tasks = {0: [2, 5, 3, 7], 1: [0, 1, 4],
....:   2: [5, 0, 4],          3: [0, 1],
....:   4: [8],                5: [2], ....:   6: [8, 9, 7]           7: [5,8,7],
....:   8: [2, 5,  3, 6, 4]    9: [2, 5, 8, 6, 1]} sage: G = Gtraph()
sage: for i in tasks:
....:   G.add_edges(("W" + str(i), "t" + str(j)) for j in tasks[i])
```

Es bleibt uns jetzt nur noch, die Methode `matching` anzuwenden, die uns eine größte Menge von Aufgaben zurückgibt, die gleichzeitig von den Personen ausgeführt werden, die das können:

```
sage: for task, worker,_ in sorted(G.matching()):
....:   print task, "kann von", worker, "ausgeführt werden"
t0 kann von w2 ausgeführt werden
t1 kann von w3 ausgeführt werden
t2 kann von w5 ausgeführt werden
t3 kann von w8 ausgeführt werden
t4 kann von w1 ausgeführt werden
t5 kann von w7 ausgeführt werden
t6 kann von w9 ausgeführt werden
t7 kann von w0 ausgeführt werden
t8 kann von w4 ausgeführt werden
t9 kann von w6 ausgeführt werden
```

16.4.3. Planung eines Turniers

Gegeben ist eine Anzahl von Mannschaften, die an einem Turnier teilnehmen, bei dem jede Mannschaft einmal gegen jede andere Mannschaft antritt. Wie kann man den Ablauf auf die schnellstmögliche Art organisieren, wenn mehrere Begegnungen gleichzeitig stattfinden können?

Dieses Aufgabe ist ein typischer Fall für die Anwendung der *Kantenfärbung* aus der Graphentheorie. Die Aufgabe besteht darin, für einen gegebenen Graphen G jeder Kante eine (evtl. abstrakte) Farbe so zuzuweisen, dass kein Knoten zwei Kanten gleicher Farbe berührt. Auf gleiche Art läuft dieses Problem darauf hinaus, eine Partition von gekoppelten Kanten (Vereinigungsmenge disjunkter Kanten) mit minimaler Kardinalität zu finden. Im vorliegenden Fall versuchen wir, die Kanten eines vollständigen Graphen zu färben, von denen jede ein Spiel repräsentiert, das von zwei Mannschaften ausgetragen wird, für welche die beiden Knoten stehen.

```
sage: n = 10
sage: G = graphs.CompleteGraph(n)
sage: from sage.graphs.graph_coloring import edge_coloring
sage: for tag, spiele in enumerate(edge_coloring(G)):
....:     print "Spiele Tag", tag, ":", spiele
Spiele Tag 0 : [(0, 9), (1, 8), (2, 7), (3, 6), (4, 5)]
Spiele Tag 1 : [(0, 2), (1, 9), (3, 8), (4, 7), (5, 6)]
Spiele Tag 2 : [(0, 4), (1, 3), (2, 9), (5, 8), (6, 7)]
Spiele Tag 3 : [(0, 6), (1, 5), (2, 4), (3, 9), (7, 8)]
Spiele Tag 4 : [(0, 8), (1, 7), (2, 6), (3, 5), (4, 9)]
Spiele Tag 5 : [(0, 1), (2, 8), (3, 7), (4, 6), (5, 9)]
Spiele Tag 6 : [(0, 3), (1, 2), (4, 8), (5, 7), (6, 9)]
Spiele Tag 7 : [(0, 5), (1, 4), (2, 3), (6, 8), (7, 9)]
Spiele Tag 8 : [(0, 7), (1, 6), (2, 5), (3, 4), (8, 9)]
```

Diese Lösung kann leicht an den Fall angepasst werden, wo nicht jeder gegen jeden spielen muss.

17. Lineare Programmierung

Dieses Kapitel geht auf die lineare Programmierung (LP) ein und die lineare Programmierung mit ganzen Zahlen (MILP) mit vielen Illustrationen von Problemen, die damit gelöst werden können. Die angeführten Beispiele stammen großenteils aus der Graphentheorie, die einfachsten können ohne besondere Vorkenntnisse verstanden werden. Und was die Kombinatorik betrifft, reduziert sich der Gebrauch der linearer Programmierung auf das Verständnis, wie ein Existenz- oder Optimierungsproblem in die Form linearer Bedingungen umgeschrieben werden kann.

17.1. Definition

Ein *lineares Programm* ist ein System von linearen Ungleichungen, für das wir eine optimale Lösung suchen. Formell definieren wir es durch eine Matrix $A : \mathbb{R}^m \to \mathbb{R}^n$ und zwei Vektoren $b \in \mathbb{R}^n$ und $c \in \mathbb{R}$. Die Lösung eines linearen Programms läuft nun darauf hinaus, einen Vektor x zu finden, der eine *Zielfunktion* maximiert und dabei einem System von Bedingungen genügt, d.h.

$$c^t tx = \max_{y, sodass AY \leq b}$$

wobei die Relation $u \leq u'$ zwischen zwei Vektoren bedeutet, dass alle Einträge von u kleiner oder gleich den entsprechenden Einträgen von u' sind. Dafür schreiben dafür

$$\begin{aligned} \text{maximiere: } & c^t x \\ \text{sodass: } & Ax \leq b. \end{aligned}$$

Das folgende lineare Programm bekommt die Lösung $x = 4$, $y = 0$ und $z = 1.6$:

$$\begin{aligned} \text{max: } & x + y + 3 \\ \text{sodass: } & x + 2y \leq 4 \\ & 5z - y \leq 8 \end{aligned}$$

Genauso können wir sagen, dass die Lösung eines linearen Programms darauf hinausläuft, einen Punkt zu finden, der eine lineare Funktion maximiert, die auf einem Polytop definiert ist (Urbild $A^{-1}(\leq b)$). Diese Definitionen erklären uns bislang noch nicht den Zusammenhang der linearen Programmierung mit der Kombinatorik, dem Hauptziel dieses Kapitels. Wir werden daher sehen, wie mit diesem Formalismus andere Probleme zu lösen sind, darunter das Rucksackproblem (Unterabschnitt 17.4.1), das Problem der Kopplung (Unterabschnitt 17.4.2) oder des Flusses (Unterabschnitt 17.4.3). In Abschnitt 17.5 benutzen wir die Erzeuger der Bedingungen zur Lösung des Existenzproblems Hamiltonzyklus.

17.2. Ganzzahlige Programmierung

Die momentan gegebene Definition wird von einer schlechten Nachricht begleitet: die Lösung eines linearen Programm kann möglicherweise in polynomialer Zeit berechnet werden. Diese Nachricht ist schlecht, weil es bedeutet, dass es bei Definition eines linearen Programms exponentieller Größe nicht möglich sein wird, dieses Verfahren zur Lösung von NP-vollständigen Problemen einzusetzen. Glücklicherweise kompliziert sich dies alles, sobald wir als Bedingung hinzufügen, dass alle oder einige Komponenten ganzzahlig sein müssen und nicht reell. In diesen Fällen erhalten wir ein lineares Programm in ganzen Zahlen (ILP), oder wenn nur einige Komponenten ganzzahlig sein müssen, ein *gemischtes* lineares Programm (auf englisch: Mixed Integer Linear Program oder MILP).

Die Lösung eines ILP oder eines MILP ist ein NP-vollständiges Problem. Deshalb können wir lange darauf warten.

17.3. In der Praxis

17.3.1. Die Klasse `MixedIntegerLinearProgram`

Die Klasse `MixedIntegerLinearProgram` stellt in Sage ein ... MILP dar. Wir verwenden sie indessen auch für kontinuierliche Probleme falls erforderlich. Sie besitzt eine sehr reduzuierte Anzahl von Methoden, die zur Definition des MILP dienen, zu seiner Lösung und dann zum Lesen der erhaltenen Lösung. Es ist auch möglich, seinen Inhalt zu visualisieren oder es in den Formaten LP oder MPS zu exportieren - beide sind Standard.

Um das zu illustrieren, lösen wir wieder das lineare Programm aus Abschnitt 17.1. Dazu müssen wir ein Objekt der Klasse `MixexIntegerLinearProgram` erzeugen,

```
sage: p = MixedIntegerLinearProgram(),
```

die drei Variablen, die wir brauchen, nachdem wir die Methode `new_variable` aufgerufen haben,

```
sage: v = p.new_variable(real=True, nonnegative=True)
sage: x, y, z = v['x'], v['y'], v['z']
```

die Zielfunktion,

```
sage: p.set_objective(x + y + 3*z)
```

und die Bedingungen

```
sage: p.add_constraint(x + 2*y <= 4)
 sage: p.add_constraint(5*7 - y <= 8)
```

Die Methode `solve` von `MixedIntegerLinearProgram` gibt den optimalen Wert der Zielfunktion zurück:

```
sage: p.solve()
8.8
```

Wir erhalten eine optimale Zuordnung zu x, y und z durch Aufruf der Methode `get_values`:

```
sage: p.get_values(x), p.get_values(y), p.get_values(z)
(4.0, 0.0, 1.6)
```

17.3.2. Variable

Die mit einer Instanz von `MixedIntegerLinearProgram` verbundenen Variablen sind Objekte des Typs `MIPVariable`, aber das berührt uns nicht wirklich. Im vorstehenden Beispiel haben wir diese Variablen durch die Kurzschreibweise `v['x']` erhalten, was ausreicht, solange die Anzahl der Variablen überschaubar ist. Die linearen Programme, die wir im Folgenden definieren werden, verlangen regelmäßig die Verbindung von Variablen mit einer Liste von Objekten, wie ganzen Zahlen, Knoten eines Graphen oder Größen anderer Art. Es ist daher nötig, von einem Vektor von Variablen sprechen zu können oder auch von einem Diktionär von Variablen.

Benötigt unser Programmm beispielsweise die Definition der Variablen $x_1, \ldots, x_{15}$, wird es wohl einfacher sein, die Methode `new_variable` mit einen Vektor aufzurufen:

```
sage: x = p.new_variable(real=True, nonnegative=True)
```

Nun ist es möglich, die Bedingungen mit Hilfe unserer 15 Variablen zu definieren:

```
sage: p.add_constraint(x[1] + x[2] - x[14] >= 8)
```

Wie man sieht, muss die Größe des Vektors `x` nicht festgelegt werden. Tatsächlich akzeptiert `x` jeden Schlüssel nicht mutablen Typs (siehe Unterabschnitt 3.3.7), genau wie ein Diktionär. Daher können wir uns erlauben zu schreiben:

```
sage: p.add_constraint(x["ich_bin_ein_gültiger_Index"] + x["a",pi] <= 3)
```

Nebenbei sei bemerkt, dass dieser Formalismus auch die Verwendung von Variablen mit mehreren Indizes erlaubt. Zur Definition der Variablen $\sum_{0 \leq i,j < 4} x_{i,j} \leq 1$ schreiben wir

```
sage: p.add_constraint(p.sum(x[i,j] for i in range(4) for j in range(4))<=1)
```

Die Schreibweise `x[i,j]` ist mit der Schreibweise `x[(i,j)]` gleichwertig.

Variablentypen. Wir haben von `new_variable` verlangt, dass die zurückgebenen Variablen positiv reell sein sollen. Das entspricht der Voreinstellung. Es ist aber auch möglich, die Variablen mit dem Argument `binary = Tru` als binär zu definieren oder mit `integer = True` als ganzzahlig. Später kann man mit `set_max` und `set_min` maximale oder minimale Schranken definieren oder umdefinieren (zum Beispiel um negative Werte zu erhalten. Auch kann der Typ einer Variablen nach ihrer Erzeugung mit den Methoden `set_binary, set_inter` oder `set_real` wieder geändert werden.

17.3.3. Nicht lösbare oder nicht endende Probleme

Manche linearen Programme haben keine Lösung. Es ist wirklich ehrgeizig, eine Funktion - selbst eine lineare - auf einer leeren Menge optimieren zu wollen oder umgekehrt auf einer Definitionsmenge, wenn die Zielfunktion nicht begrenzt ist. In beiden Fällen wirft Sage bei Aufruf der Methode `solve` eine Ausnahme aus:

```
sage: p = MixedIntegerLinearProgram
sage: v = p.new_variable(real=True, nonnegative=True)
sage: p.set_objective(v[3] + v[2])
sage: p.add_constraint(v[3] <= 5)

sage: p.solve()
Traceback (most recent call last):
...
sage.numerical.mip.MIPSolverException:
 GLPK: The LP (relaxation) problem has no dual feasible solution

sage: p.add_constraint(v[2] <= 8)
sage: p.solve
13.0

sage: p.add_constraint(v[3] >= 6); p.solve()
Traceback (click to the left of this block for traceback)
...
sage.numerical.mip.MIPSolverException: GLPK: Problem has no feasible
solution
```

Auch kann die Vorschrift der Ganzzahligkeit eine leere Menge ergeben:

```
sage: p = MixedIntegerLinearProgram()
sage: v = p.new_variable()
sage: p.set_objective(v[3])
sage: p.add_constraint(v[3] <= 4.75); p.add_constraint(v[3] >= 4.25)
sage: p.solve()
4.75

sage: p.set_integer(v[3]); p.solve()
Traceback (click to the left of this block for traceback)
...
sage.numerical.mip.MIPSolverException: GLPK: Problem has no feasible
solution
```

Welche Möglichkeit auch immer eintritt, es wäre unvernünftig, einen Code unter dem Vorwand, ein lineares Programm könne nicht gelöst werden, abrupt zu stoppen; manche linearen Programme haben übrigens den einzigen Zweck, die *Existenz* einer Lösung zu prüfen und sind daher selten lösbar. Um diese Möglichkeiten zu erzeugen, wird man den klasssischen Mechanismus des *„try-except“* von Python verwenden, um die Ausnahme abzufangen:

```
sage: try:
....:     p.solve
....:     print "Das Problem hat eine Lösung!"
....: except:
```

```
....:     print "Das Problem ist nicht lösbar!"
Das Problem ist nicht lösbar!
```

17.4. Erste Anwendungen auf die Kombinatorik

Nachdem nun die Grundlagen geklärt sind, gehen wir zu einem interessanteren Aspekt über, der Modellierung. In diesem Abschnitt finden sich mehrere Optimierungs- oder Existenzprobleme: wir beginnen mit ihrer abstrakten Definition, dann folgt eine Modellierung als MILP, was ermöglicht, mit wenigen Zeilen Code einen Algorithmus für ein NP-vollständiges Problem zu erhalten.

17.4.1. Das Rucksackproblem

Das sogenannte Rucksackproblem ist das folgende: Wir haben eine Reihe von Gegenständen mit jeweils eigenem Gewicht vor uns, sodass ein „Nutzen“ als reelle Zahl gemessen werden kann. Wir möchten jetzt einige dieser Gegenstände auswählen, und dabei sicher sein, dass dass das zulässige Gesamtgewicht C nicht überschritten wird. Die Anzahl der Gegenstände soll möglichst groß werden.

Dazu weisen wir jedem Objekt o aus einer liste L eine binäre Variable `genommen[o]` zu, die gleich 1 ist, wenn der Gegenstand in den Rucksack gepackt ist, und 0, wenn nicht. Wir versuchen nun das folgende MILP zu lösen:

$$\begin{aligned}\max: &\sum_{o\in L} \text{Nutzen}_o \times \text{genommen}_o\\ \text{sodass}: &\sum_{o\in L} \text{Gewicht}_o \times \text{genommen}_0 \leq C\end{aligned}$$

In Sage weisen wir den Objekten einen Preis und einen Nutzen zu:

```
sage: C = 1
sage: L = ["Topf", "Buch", "Messer", "Flasche", "Taschenlampe"]
sage: p = [0.57, 0.35, 0.98, 0.39, 0.08]; u = [0.57, 0.26, 0.29, 0.85, 0.23]
sage: Gewicht = {}; Nutzen = {}
sage: for o in L:
....:     Gewicht[o] = p[L.index(o)]; Nutzen[o] = u[L.index(o)]
```

Nun können wir das MILP schreiben:

```
sage: p = MixedIntegerLinearProgram()
sage: genommen = p.new:_variable(binary = True)
sage: p.add_constraint(
....:     p.sum(Gewicht[o]*genommen[o] for o in L) <= C
sage: p.set_objective(
....:     p.sum(Nutzen[o]*genommen[o] for o in L))
sage: p.solve()
1.42
sage: mitnehmen = p.get_values(mitnehmen)
```

Die gefundene Lösung bestätigt die Gewichtbeschränkung:

```
sage: sum(Gewicht[o]*mitnehmen[o] for o in L)
0.96
```

Muss man eine Flasche mitnehmen?

```
sage: prendre["Flasche"]
```

Übung 63. *(Teilmengenproblem)* Die *Teilmengenproblem* genannte Rechenaufgabe besteht darin, in einer Menge ganzer Zahlen eine nichtleere Teilmenge von Elementen zu suchen, deren Summe null ist. Lösen Sie diese Aufgabe mit einem ganzzahligen linearen Programm für die Menge `{28, 10, -89, 69, 42, -37, 76, 78, -40, 92, -93, 45}`.

17.4.2. Paarung (Matching)

In einem Graphen eine Paarung zu suchen, heißt, eine Menge von Kanten zu finden, die paarweise disjunkt sind. Da die leere Menge schon als Paarung gilt, richtet sich unser Augenmerk auf eine maximale Paarung: wir versuchen, die Anzahl der Kanten in einer Paarung zu maximieren. Die Aufgabe der maximalen Paarung ist dank eines Ergebnisses von Jack Edmonds [Edm65] ein polynomiales Problem. Sein Algorithmus basiert auf lokalen Verbesserungen, auch der Beweis, dass der Algorithmus nur bei einer maximalen Paarung stoppt. Dabei handelt es sich nicht um einen Algorithmus der Graphentheorie, der schwieriger zu implementieren wäre, sondern seine Formulierung in MILP braucht nur zwei Zeilen.

Wir müssen dafür, wie vorher auch, jedem unserer Objekte - den Kanten eines Graphen - einen passenden binären Wert zuweisen, ob diese Kante in unserem Matsching erscheint oder nicht.

Wir müssen uns dann vergewissern, dass zwei adjazente Kanten sich nicht beide gleichzeitig im Matching befinden. Das ähnelt, zumindeswt von weitem, an eine lineare Bedingung: wenn x und y zwei Kanten eines Graphen sind, und wenn m_x und m_y die beiden Variablen sind, denen die Kanten zugewiesen worden sind, dann reicht es hin zu verlangen, dass $m_x + m_y \leq 1$ ist.

Die beiden Kanten können nicht gleichzeitig in unserer Lösung zu finden sein, und wir sind daher in der Lage, ein lineares Programm zu schreiben, dass eine maximale Paarung berechnet. Es ist indessen möglich schneller zu sein, wenn man beachtet, dass wenn zwei Kanten nicht gleichzeitig in einem Matching sein können, dies der Fall ist, weil beide durch einen Knoten v des Graphen gehen. Deshalb ist es schneller zu sagen, dass jeder Knoten v von höchstens *einer* Kante eines Matching berührt wird.

Diese Bedingung ist ebenfalls linear.

$$\begin{aligned} \max : & \sum_{e \in E(G)} m_e \\ \text{sodass} : \forall v \in V(G), & \sum_{e \in E(G),\, e=uv} m_e \leq 1. \end{aligned}$$

In Sage ist ein solches MILP wieder einfach hinzuschreiben:

```
sage: g = graphs.PetersenGraph()
sage: p = MixedIntegerLinearProgram()
sage: Paarung = p.new_variable(binary = True)

sage: p.set_objective(p.sum(Paarung[e]
....:           for e in g.edges(labels = False)))

sage: for v in g:
....:     p.add_constraint(p.sum(Paarung[e]
....:         for e in g.edges_incident(v, labels = False)) <= 1)
sage: p.solve()
5.0

sage: Paarung = p.get_values(Paarung)
sage: [e for e, b in Paarung.iteritems() if b == 1]
```

Übung 64. *(Führende Menge)* Eine dominierende Menge eines Graphen ist eine Menge S von Knoten, sodass jeder Knoten, der nicht in S ist, mindestens einen Nachbarn in S besitzt. Schreiben Sie ein ganzzahliges lineares Programm, um im Petersen-Graph eine dominierende Menge minimaler Kardinalität zu suchen.

17.4.3. Fluss

Dieser Abschnitt präsentiert einen weiteren fundamentalen Algorithmus der Graphentheorie: den Fluss! Der besteht darin, zu zwei gegebenen Knoten s und t eines *gerichteten* Graphen G (d.h. seine Kanten haben eine Richtung, siehe Abb. 17.1) von s nach t durch die Kanten von G einen maximalen *Durchfluss* oder *Fluss* passieren zu lassen. Jede dieser Kanten weist eine maximale Kapazität auf, nämlich den Fluss, der maximal durch sie hindurchgehen kann.

Die Definition dieses Problems bietet uns sofort seine Formulierung als lineares Programm an: wir suchen einen reellen Wert, der jeder Kante zugeordnet wird, der die Intensität des Flusses durch diese Kante repräsentiert und der zwei Bedingungen genügt:

- die Größe des bei einem Knoten (außer s und t) ankommenden Flusses muss gleich der Größe des abgehenden Flusses sein (Knoten sind keine Speicher),
- der Fluss durch eine Kante kann ihre Kapazität nicht übersteigen.

Nachdem dies klargestellt ist, bleibt uns nur zu versuchen, den Fluss, der s verlässt, zu maximieren: der kann seinen Lauf nur am Punkt t beenden, denn die anderen Knoten können nur weiterleiten, was bei ihnen ankommt. Wir formulieren deshalb das Flussproblem durch das folgende lineare Programm (wobei wir die Kapazität der Kanten zu 1 annehmen):

$$\begin{aligned}\max: & \sum_{sv\in E(G)} f_{sv}\\ \text{sodass: } & \forall v\in V(G)\backslash\{s,t\}, \sum_{vu\in E(G)} f_{vu} = \sum_{uv\in E(G)} f_{uv}\\ & \forall uv\in E(G), f_{uv}\leq 1.\end{aligned}$$

Wir lösen hier das Flussproblem auf einer orientierung des Graphen von Chvatal (siehe Abb. 17.1), in dem alle Kapazitäten gleich 1 sind:

```
sage: g = graphs.ChvatalGraph()
sage: g = g.minimum_outdegree_orientation()


sage: p = MixedIntegerLinearProgram()
sage: f = p.new_variable()
sage: s, t = 0, 2

sage: for v in g:
....:     if v == s or v == t: continue
....:     p.add_constraint(
....:         p.sum(f[u,v] for u in g.neighbors_out(v)) ==
....:         p.sum(f[u,v] for u in g.neighbors_in(v)))

sage: for e in g.edges(labels = False): p.add_constraint(f[e] <= 1)
sage: p.set_objective(p.sum(f[s,u] for u in g.neighbors_out(s)))

sage: print p.solve()
2.0
```

17.5. Erzeugung von Bedingungen und Anwendung

Auch wenn die vorstehenden Beispiele den Eindruck großer Ausdrucksvielfalt vermitteln, ist die „Interpretation“ eines gegebenen Optimierungs- oder Existenzproblems durch seine Formulierung als lineares Programm eine willkürliche Wahl. Das gleiche Problem kann durch verschiedene Formulierungen gelöst werden, deren Leistungsfähigkeiten ebenfalls sehr unterschiedlich sind. Das bringt uns dazu, die Möglichkeiten der Löser eines MILP intelligent zu nutzen, indem wir jetzt von ihnen verlangen, solche linearen Preogramme zu lösen, deren Bedingungen nicht sämtlich bekannt sind, und indem ihm nach und nach diejenigen hinzugefügt werden, die nötig sind: diese Technik wird unverzichtbar, sobald die Anzahl der Bedingungen zu groß wird, um bei der Erstellung des linearen Programms explizit angegeben werden zu können. Wir bereiten uns nun auf die Lösung des Hamiltonkreisproblems vor (ein Spezialfall des *Problems des Handlungsreisenden*).

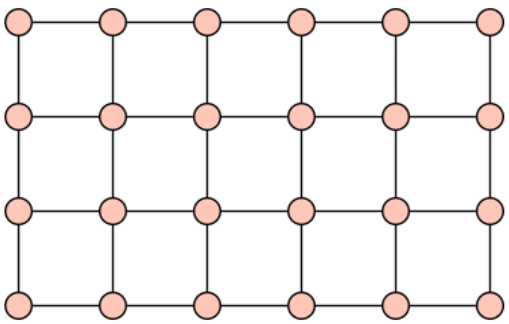


Abb. 17.2 - Das Gitter der Größe 4×6 auf dem wir unsere Formulierungen testen.

Wir sagen, dass ein in einem Graphen enthaltener Kreis $C \subseteq E(G)$ hamiltonisch ist, wenn er durch jeden Knoten von G geht. Die Prüfung auf Existenz eines Hamilton-Zyklus in einem gegebenen Graphen ist ein NP-vollständiges Problem: eine schnelle Lösung muss man also

nicht erwarten, was uns aber nicht abhalten soll zu versuchen, es als lineares Programm zu formulieren. Eine erste Formulierung könnte auf folgende Weise beginnen:

- weise jede Kante einer binären Variablen b_e zu, die anzeigt, ob die Kante im Kreis C enthalten ist,
- lege für jeden Knoten fest, dass er genau zwei Kanten besitzt, die inzident sind in C.

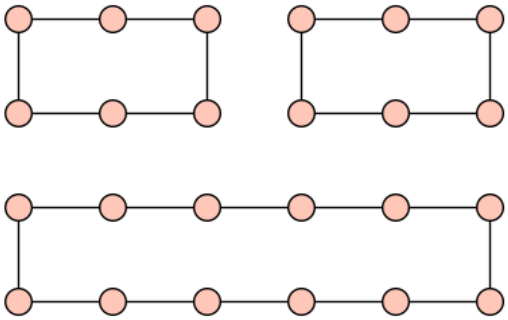


Abb. 17.3 - Eine mögliche Lösung der Gleichungen des Grades

Leider ist diese Formulierung nicht exakt. Es ist auf jeden Fall möglich, dass die mit dieser Formulierung erhaltene Lösung eine disjunkte Vereinigungsmenge mehrerer Kreise ist - jeder Knoten hätte daher wohl zwei Nachbarn in C, es wäre aber nicht unbedingt möglich, unter alleiniger Benutzung der Kanten von C von einem Knoten v zu einem anderen Knoten u zu gelangen.

Es ist indessen möglich, eine komplexere *und* exakte Formulierung (benannt nach Miller-Tucker-Zemlin) zu finden:

- man weise jedem Knoten v des Graphen eine ganze Zahl i_v zu, die den Schritt angibt, bei dem der Kreis C ihn passiert; dabei ist $i_{v0} = 0$ für einen festen Knoten v_0,
- man weise jeder Kante uv von G zwei binäre Variablen b_{uv} und b_{vu} zu, die anzeigen, ob die Kante Teil des Kreises C ist (und wir sagen auch, in welcher Richtung diese Kante durchlaufen wird),
- man lege für jeden Knoten fest, dass er in C eine eingehende und eine abgehende Kante besitzt,
- eine Kante uv kann nur in C erscheinen, wenn $i_u < i_v$ ist (die Kanten werden in wachsender Reihenfolge durchlaufen).

Diese Formulierung kann folgendermaßen in linearen Gleichungen geschrieben werden:

$$
\begin{aligned}
&\max : \text{Schritt der zu optimierenden Funktion}\\
\text{sodass: } &\forall u \in V(G), \sum_{uv \in E(G)} b_{uv} = 1\\
&\forall u \in V(G), \sum_{uv \in E(G)} b_{vu} = 1\\
&\forall uv \in E(G \backslash v_0), && i_u - i_v + |V(G)| b_{uv} \leq |V(G)| - 1\\
& && i_v - i_u + |V(G)| b_{vu} \leq |V(G)| - 1\\
&\forall v \in V(G), i_v \leq |V(G)| - 1\\
&b_{uv} \text{ ist eine binäre Variable}\\
&i_v \text{ ist eine ganzzahlige Variable}
\end{aligned}
$$

Wir bemerken in dieser Formulierung die Anwesenheit eines Koeffizienten $|V(G)|$, was oft ein Zeichen dafür ist, dass wir nicht sehr schnell zu einer Lösung des linearen Problems gelangen werden. Auch benutzen wir für das Hamiltonkreisproblem eine andere Formulierung. Sein Prinzip ist in der Tat etwas einfacher und beruht auf folgender Beobachtung: wenn in unserem Graphen ein Hamilton-Zyklus existiert, existiert er auch für jede echte Teilmenge S der Knoten mit mindestens zwei Kanten von C, die in S hineingehen oder aus S herausführen. Wenn wir uns entschließen, mit $\bar{S}$ die Menge von Kanten zu bezeichnen, die genau eine Extremität in S besitzen, erhalten wir folgende Formulierung (welche die Variablen b_{uv} und v_{vu} gleichsetzt):

$$
\begin{aligned}
&\max : \text{Schritt der zu optimierenden Funktion}\\
\text{sodass: } &\forall u \in V(G), \sum_{uv \in E(G)} b_{uv} = 2\\
&\forall S \subseteq V(G), \emptyset \neq V(G), \sum_{e \in \bar{S}} v_e \geq 2\\
&b_{uv} \text{ ist eine binäre Variable}
\end{aligned}
$$

Es wäre allerdings riskant, die vorstehende Formulierung zur Lösung eines Hamiltonkreisproblems direkt zu verwenden, selbst bei einem so kleinen Graphen wie unserem Gitter mit $6 \times 4 = 24$ Elementen: die Bedingungen bezüglich der Mengen S wären $2^{24} - 2 = 16777214$ an der Zahl. Im Gegensatz dazu bietet sich das von den Lösern linearer Ungleichungen verwendete Verfahren des *branch-and-bound* (oder *branch-and-cut*) zur Erzeugung der Bedingungen *während* der Lösung eines linearen Programms eher an. Wenn ein lineneares erst einmal gelöst ist, ist es also möglich, eine weitere Bedingung hinzuzufügen und das neue Programm mit einem Teil der während der vorigen Lösung ausgeführten Rechnungen zu lösen. Die Erzeugung der Bedingungen für das Hamiltonkreisproblem besteht also daraus, nachstehenden Etappen zu folgen:

- Erzeugen eines linearen Programms - ohne Zielfunktion - mit einer binären Variablen je Kante,
- Hinzufügen einer Bedingung zu jedem Knoten, die den Grad 2 bewirkt,
- Lösen des linearen Programms,
- falls die aktuelle Lösung kein Hamilton-Zyklus ist (das ist ein Teilgraph, der mehrere verbundene Bestandteile hat), S aufrufen, den einen der verbundenen Bestanteile, und die Bedingung hinzufügen, die bewirkt, dass wenigsten zwei Kanten aus S herausführen.

Zu unserem Glück sind die Verifikation der Ungültigkeit der aktuellen Lösung und die Erzeugung der entsprechenden Bedingung algorithmisch schnell. Hier ist die Lösung des Hamiltonkreisproblems für unser Gitter durch Erzeugung der Bedingungen durch Sage:

```
sage: g = graphs.Grid2dGraph(4, 6)
sage: p = MixedIntegerLinearProgram()
sage: b = p.new_variable(binary = True)
```

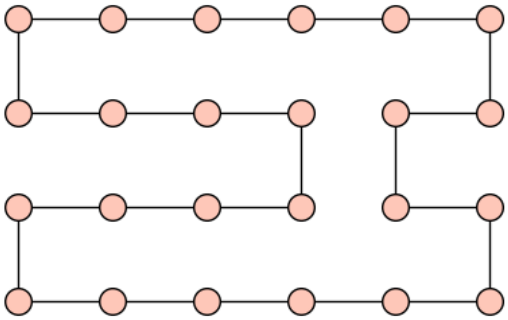


Abb. 17.4 - Ein durch Erzeugung von Bedingungen berechneter Hamilton-Zyklus.

Um keinen Unterschied zwischen den Variablen `b[u,v]` und `b[v,u]` zu machen, ist es angebracht, eine Lambda-Funktion zu erzeugen, die das geordnete Paar x, y durch die Menge $\{x, y\}$ ersetzt:

```
sage: B = lambda x, y: b[frozenset([x,y])]
```

Nun fügen wir die Bedingungen für den Grad hinzu:

```
sage: for u in g:
....:     p.add_constraint(p.sum(B(u,v) for v in g.neighbors(u)) == 2)
```

Nun ist es Zeit, die erste Lösung unseres Problems zu berechnen und den Graphen zu erzeugen, der sie abbildet.

```
sage: print p.solve()
0.0
sage: h = Graph()
sage: h.add_edges([(u,v) for u,v in g.edges(labels=False)
....:              if p.get_values(B(u,v)) == 1.0])
```

und dann mit den Iterationen zu beginnen:

```
sage: while not h.is_connected():
....:     S = h.connected_components()[0]
....:     p.add_constraint(
....:         p.sum(B(u,v) for u,v
....:               in g.edge_boundary(S, labels = False)) >= 2)
....:     zero = p.solve()
....:     h = Graph()
....:     h.add_edges([(u,v) for u,v in g.edges(labels=False)
....:                  if p.get_values(B(u,v)) == 1.0])
```

Mit mindestens zehn Iterationen (womit im Vergleich zu $2^{24} - 2$ eine Ersparnis an Rechenzeit einhergeht) erhalten wir eine zulässige Lösung für unser Gitter (siehe Abb. 17.4). Die Leistungsfähigkeit dieses Lösungsverfahrens ist mit derjenigen nach Miller-Tucker-Zemlin nicht vergleichbar. Wenn wir beide Programme in Sage implementieren, erhalten wir für einen Zufallsgraphen $\mathcal{G}_{35,0.3}$ folgende Rechenzeiten:

```
sage: g = graphs.RandomGNP(35, 0.3)
sage: %time MTZ(g)
CPU times: user 51.52 s, sys: 0.24 s, total: 51.76 s
Wall time: 52.84 s
sage: time constraint_generation(g)
sage: %time constraint_generation(g)
CPU times: user 0.23 s, sys: 0.00 s, total: 0.23 s
Wall time: 0.26 s
```

A. Lösungen der Übungen

A.1. Erste Schritte

Übung 1 Seite 15. Der Befehl `SR.var('u')` erzeugt die symbolische Variable u und weist sie der Sage-Variablen `u` zu. Die Sage-Variable `u` empfängt dann in zwei Wiederholungen ihren aktuellen Wert plus eins, also erst $u+1$, dann $u+2$ (wobei u immer die symbolische Variable ist).

A.2. Analysis und Algebra mit Sage

Übung 2 Seite 27. *(Rekursive Berechnung der Summe)*

```
sage: n = var('n'); pmax = 4; s = [n + 1]
....:     for p in [1..pmax]:
....:         s += [factor(((n+1)^(p+1) - sum(binomial(p+1, j)*s[j]
....:                 for j  in [0..p-1]))/(p+1))]
sage: s
```

Damit erhalten wir:

$$\sum_{k=0}^{n} k = \frac{1}{2}(n+1)n, \quad \sum_{k=0}^{n} k^2 = \frac{1}{6}(n+1)(2n+1)n$$
$$\sum_{k=0}^{n} k^3 = \frac{1}{4}(n+1)^2n^2, \quad \sum_{k=0}^{n} k^4 = \frac{1}{30}(n+1)(2n+1)(3n^2+3n-1)n$$

Übung 3 Seite 30. *(Eine symbolische Grenzwertberechnung)* Um die Frage zu beantworten, werden wir eine symbolische Funktion verwenden, mit der wir die Taylorentwicklung um 0 bis zum 3. Grad berechnen.

```
sage: x, h, a = var('x h a'); f = function('f')
sage: g(x) = taylor(f(x), x, a, 3)
sage: phi(h) = (g(a+3*h) - 3*g(a+2*h) + 3*g(a+h) - g(a))/h^3; phi(h)
D[0, 0, 0](f)(a)
```

Die Funktion g unterscheidet sich von f um einen Rest, der gegenüber h^3 vernachlässigbar ist, somit unterscheidet sich die Funktion `phi` vom untersuchten Quotienten um einen Rest $\mathcal{O}(1)$; also hat `phi` mit null den gesuchten Grenzwert. Abschließend gilt

$$\lim_{h\to 0} \frac{1}{h^3}(f(a+3h) - 3f(a+2h) + 3f(a+h) - f(a)) = f'''(a).$$

Diese Formel gestattet die näherungsweise Berechnung der 3. Ableitung von f, ohne auch nur eine Ableitung vorzunehmen.

Wir können davon ausgehen, dass die Formel in folgender Form generalisiert wird:

$$\lim_{h\to 0}\frac{1}{h^n}\left(\sum_{k=0}^{n}(-1)^{n-k}\binom{n}{k}f(a+kh)\right)=f^{(n)}(a).$$

Zur Überprüfung dieser Formel für größere Werte von n, können wir den obigen Code nun einfach anpassen:

```
sage: n = 7; x, h, a = var('x h a'); f = function('f')
sage: g(x) = taylor(f(x), x, a, n)
sage: sum((-1)^(n-k)*binomial(n,k)*g(a+k*h) for k in (0..n))/h^n
D[0, 0, 0, 0, 0, 0, 0](f)(a)
```

Übung 4 Seite 31. *(Eine Formel von Gauß).*

1. Wir wenden nacheinander `trig_expand` und `trig_simplify` an:

   ```
   sage: theta = 12*arctan(1/38) + 20*arctan(1/57) \
   ....:       + 7*arctan(1/239) + 24*arctan(1/268)
   sage: tan(theta).trig_expand().trig_simplify()
   1
   ```

2. Die Tangensfunktion ist auf $I=\left[0,\frac{\pi}{4}\right]$ konvex, deshalb liegt ihr Graph unterhalb ihrer Sehne; anders gesagt, $\forall x\in I$, $\tan x\le\frac{4}{\pi}x$.

   ```
   sage: 12*(1/38) + 20*(1/57) + 7*(1/239) + 24*(1/268)
   37735/48039
   ```

   Daraus leiten wir her:

   $$\begin{aligned}\theta&=12\arctan\frac{1}{38}+20\arctan\frac{1}{57}+7\arctan 1239+24\arctan 1268\\&\le 12\cdot\frac{1}{28}+20\cdot\frac{1}{57}+7\cdot\frac{1}{239}+24\cdot\frac{1}{268}\\&=\frac{37735}{48039}\le\frac{4}{\pi}.\end{aligned}$$

   Deshalb ist $\theta\in I$; nun ist (siehe Frage 1) $\tan\theta=1=\tan\pi 4$ und der Tangens ist injektiv auf I. Daraus schließen wir $\theta=\frac{\pi}{4}$.

3. Wir setzen das Taylorpolynom in die Formel von Gauß ein:

   ```
   sage: x = var('x'); f(x) = taylor(arctan(x), x, 0, 21)
   sage: approx = 4*(12*f(1/38) + 20*f(1/57) + 7*f(1/239) + 24*f(1/268))
   sage: approx.n(digits=50); pi.n(digits=50)
   3.1415926535897932384626433832795028851616168852864
   3.1415926535897932384626433832795028841971693993751
   sage: approx.n(digits=50) - pi.n(digits=50)
   9.6444748591132486785420917537404705292978817080880e-37
   ```

Übung 5 Seite 32. *(Asymptotische Entwicklung einer Folge).* Die Einschachtelung von x^n erlaubt die Behauptung $x_n \sim n\pi$, es sei also $x_n = n\pi + \wr(n)$.

Wir setzen diese Gleichung dann in folgende Gleichung ein, die wir aus $\arctan x + \arctan(1/x) = \pi/2$ erhalten haben:

$$x_n = n\pi + \frac{\pi}{2} - \arctan\left(\frac{1}{x_n}\right).$$

Wir setzen danach die asymptotischen Entwicklungen von x_n wieder ein, die wir mit dieser Gleichung bekommen haben, bis repetita placent[1] (Methode der fortschreitenden Verfeinerung).

Da wir wissen, dass bei jedem Schritt eine Entwicklung der Ordnung p eine Entwicklung der Ordnung $p+2$ ermöglicht, können wir in vier Schritten eine Entwicklung der Ordnung 6 erhalten. Wie wir in Kapitel 3 schon vorweggenommen haben, können wir diese vier Etappen in einer Schleife ausführen:

```
sage: n = var('n'); phi = lambda x: n*pi + pi/2 - arctan(1/x)
sage: x = n*pi
sage: for i in range(4):
....:     x = taylor(phi(x), n, infinity, 2*i); x
```

Wir bekommen schließlich

$$x_n = \frac{1}{2}\,\pi + \pi n - \frac{1}{\pi n} + \frac{1}{2\,\pi n^2} - \frac{3\,\pi^2+8}{12\,\pi^3 n^3} + \frac{\pi^2+8}{8\,\pi^3 n^4} - \frac{15\,\pi^4+240\pi^2+208}{240\,\pi^5 n^5} + \frac{3\,\pi^4+80\,\pi^2+208}{96\,\pi^5 n^6} + \mathcal{O}\left(\frac{1}{n^7}\right)$$

Übung 6 Seite 33. *(Ein Gegenbeispiel von Peano zum Satz von Schwarz)* Die partiellen Ableitungen $f(x,0)$ und $f(0,x)$ sind in $(0,0)$ identisch null; daraus folgern wir ohne Rechnung $\partial_1 f(0,0) = \partial_2 f(0,0) = 0$. Dann berechnen wir die Werte der zweiten partiellen Ableitungen in $(0,0)$:

```
sage: h = var('h'); f(x, y) = x * y * (x^2 - y^2) / (x^2 + y^2)
sage: D1f(x, y) = diff(f(x,y), x); limit((D1f(0,h) - 0) / h, h=0)
-1
sage: D2f(x, y) = diff(f(x,y), y); limit((D2f(h,0) - 0) / h, h=0)
1
sage: g = plot3d(f(x, y), (x, -3, 3), (y, -3, 3))
sage: g.show(viewer='tachyon')
```

[1] lat. etwa: Wiederholungen tun gut

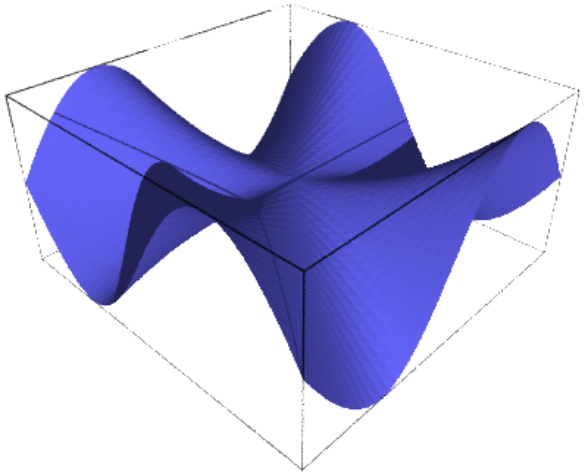


Abb. A.1 - Die Peano-Fläche

Daraus leiten wir $\partial_1\partial_2 f(0,0) = 1$ und $\partial_2\partial_1 f(0,0) = -1$ her. Somit liefert diese Funktion ein Gegenbeispiel zum Satz von Schwarz (Abb. A.1).

Übung 7 Seite 34. *(BBP-Formel)*

1. Wir beginnen mit dem Vergleich von
   $u_n = \int_0^{1/\sqrt{2}} f(t) \cdot t^{8n} \mathrm{d}t$ und $v_n = \left(\frac{4}{8n+1} - \frac{2}{8n+4} - \frac{1}{8n+5} - \frac{1}{8n+6}\right)\left(\frac{1}{16}\right)^n$.

   ```
   sage: n, t = var('n, t')
   sage: v(n) = (4/(8*n+1)-2/(8*n+4)-1/(8*n+5)-1/(8*n+6))*1/16^n
   sage: assume(8*n+1>0)
   sage: f(t) = 4*sqrt(2)-8*t^3-4*sqrt(2)*t^4-8*t^5
   sage: u(n) = integrate(f(t) * t^(8*n), t, 0, 1/sqrt(2))
   sage: (u(n)-v(n)).simplify_full()
   0
   ```
   [2]

   Wir folgern daraus $u_n = v_n$. Wegen der Linearität des Integrals erhalten wir:

   $$I_N = \int_0^{1/\sqrt{2}} f(t) \cdot \left(\sum_l imits_{n=0}^N t^{8n}\right) \mathrm{d}t = \sum_{n=0}^N u_n = \sum_{n=0}^N v_n = S_N.$$

[2] Sage vereinfacht so weit nicht, der gelieferte Ausdruck lässt sich jedoch leicht als nullwertig erkennen.

2. Die Potenzreihe $\sum_{n\geq 0} t^{8n}$ hat den Konvergenzradius, sie konvergiert daher im Intervall $[0, \frac{1}{\sqrt{2}}]$. Auf diesem Intervall können wir deshalb Integral und Grenze vertauschen:

$$\begin{aligned}\lim_{N\to\infty} S_N &= \lim_{N\to\infty} \int_0^{1/\sqrt{2}} f(t)\cdot\left(\sum_{n=0}^{N} t^{8n}\right)\mathrm{d}t \\ &= \int_0^{1/\sqrt{2}} f(t)\cdot\left(\sum_{n=0}^{\infty} t^{8n}\right)\mathrm{d}t \\ &= \int_0^{1/\sqrt{2}} f(t)\cdot\frac{1}{1-t^8}\mathrm{d}t = J.\end{aligned}$$

3. Wir fahren dann mit der Berechnung von J fort.

```
sage: t = var('t'); J = integrate(f(t)/(1-t^8), t, 0, 1/sqrt(2))
sage: J.simplify_full()
pi + 2*log(sqrt(2) - 1) + 2*log(sqrt(2) + 1)
```

Zur Vereinfachung dieses Ausdrucks müssen wir Sage anweisen, die Summe der Logarithmen zu bilden:

```
sage: J.simplify_log().simplify_full()
pi
```

Letztendlich erhalten wir die verlangte Gleichung:

$$\sum_{n=0}^{+\infty}\left(\frac{4}{8n+1}-\frac{2}{8n+4}-\frac{1}{8n+5}-\frac{1}{8n+6}\right)\left(\frac{1}{16}\right)^n = \pi.$$

Mit Hilfe dieser Gleichung geben wieder einen Näherungswert für π:

```
sage: l = sum(v(n) for n in (0..40)); l.n(digits=60)
3.14159265358979323846264338327950288419716939937510581474759
sage: pi.n(digits=60)
3.14159265358979323846264338327950288419716939937510582097494
sage: print "%e" % (l-pi).n(digits=60)
-6.227358e-54
```

Übung 8 Seite 35. *(Polynomiale Näherung für den Sinus)* Wir versehen den Vektorraum $\mathcal{C}^\infty([-\pi,\pi])$ mit dem Skalarprodukt $\langle f|G\rangle = \int_{-\pi}^{\pi} fg$. Das gesuchte Polynom ist die orthogonale Projektion der Sinusfunktion auf den untervektorraum $\mathbb{R}_5[X]$. Die Bestimmung dieses Polynoms wird auf die Lösung eines linearen Gleichungssystem zurückgeführt: tatsächlich ist P die Projektion des Sinus genau dann, wenn die Funktion $(P - \sin)$ zu jedem Vektor der kanonischen Basis von $\mathbb{R}_5[X]$ orthogonal ist. Hier folgt der Sage-Code:

```
sage: var('x'); ps = lambda f, g : integral(f * g, x, -pi, pi)
sage: n = 5; a = var('a0, a1, a2, a3, a4, a5')
sage: P = sum(a[k] * x^k for k in (0..n))
sage: equ = [ps(P - sin(x), x^k) for k in (0..n)]
sage: sol = solve(equ, a)
```

```
sage: P = sum(sol[0][k].rhs() * x^k for k in (0..n)); P
105/8*(pi^4 - 153*pi^2 + 1485)*x/pi^6 - 315/4*(pi^4 - 125*pi^2 +
            1155)*x^3/pi^8 + 693/8*(pi^4 - 105*pi^2 + 945)*x^5/pi^10
sage: g = plot(P,x,-6,6,color='red') + plot(sin(x),x,-6,6,color='blue')
sage: g.show(ymin = -1.5, ymax = 1.5)
```

Nun können wir die Sinusfunktion und ihre orthogonale Projektion zeichnen und die Qualität dieser polynomialen Approximation beurteilen.

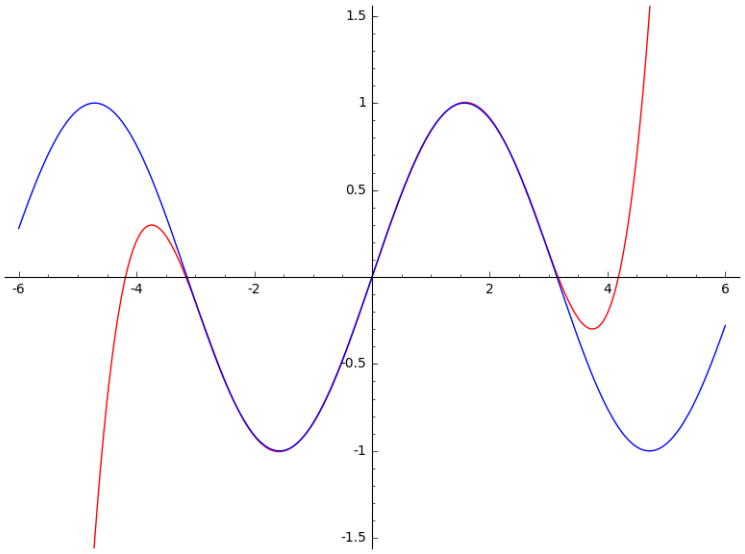


Abb. A.2 - Approximation des Sinus mit der Methode der kleinsten Quadrate

Übung 9 Seite 36. *(Das Problem von Gauß)* Zunächst beweisen wir die geforderten Beziehungen symbolisch. Dem schließt sich die numerische Ableitung an. Wir beginnen mit der Definition der Vektoren $\vec{r_i}$:

```
sage: p, e = var('p, e')
sage: theta1, theta2, theta3 = var('theta1, theta2, theta3')
sage: r(theta) = p / (1 - e * cos(theta))
sage: r1 = r(theta1); r2 = r(theta2); r3 = r(theta3)
sage: R1 = vector([r1 * cos(theta1), r1 * sin(theta1), 0])
sage: R2 = vector([r2 * cos(theta2), r2 * sin(theta2), 0])
sage: R3 = vector([r3 * cos(theta3), r3 * sin(theta3), 0])
```

- Wir verifizieren, dass $\vec{S} + e \cdot \left(\vec{\imath} \wedge \vec{D}\right)$ der Nullvektor ist.

```
sage: D = R1.cross_product(R2)+R2.cross_product(R3)+R3.cross_product(R1)
sage: S = (r1 - r3) * R2 + (r3 - r2) * R1 + (r2 - r1) * R3
sage: i = vector([1, 0, 0]); V = S + e * i.cross_product(D)
sage: V.simplify_full()
(0, 0, 0)
```

Daraus leiten wir die geforderte Beziehung her: $e = \frac{\|\vec{S}\|}{\|\vec{\imath} \wedge \vec{D}\|} = \frac{\|\vec{S}\|}{\|\vec{D}\|}$, denn $\vec{D}$ steht senkrecht auf der Bahnebene und somit auch auf $\vec{\imath}$.

• Nun verifizieren wir, dass $\vec{\imath}$ zu $\vec{S} \wedge \vec{D}$ kollinear ist:

```
sage: S.cross_product(D).simplify_full()[1:3]
(0, 0)
```

Dieses Ergebnis zeigt, dass die zweite und die dritte Komponente verschwinden.

• Ebenso verifizieren wir, dass $p \cdot \vec{S} + e \cdot \left(\vec{\imath} \wedge \vec{N}\right)$ der Nullvektor ist.

```
sage: N = r3 * R1.cross_product(R2) + r1 * R2.cross_product(R3)\
....: + r2 * R3.cross_product(R1)
sage: W = p * S + e * i.cross_product(N)
sage: W.simplify_full()
(0, 0, 0) Daraus folgern wir:
```

$$p = e\frac{\|\vec{\imath} \wedge \vec{N}\|}{\|\vec{S}\|} = e\frac{\|\vec{N}\|}{\|\vec{S}\|} = \frac{\|\vec{N}\|}{\|\vec{D}\|},$$

denn $\vec{N}$ steht auf der Bahnebene senkrecht und somit auch auf $\vec{\imath}$.

• Als Formel für Kegelschnitte haben wir $a = \frac{p}{1-e^2}$.
• Wir kommen jetzt zur verlangten numerischen Abbildung:

```
sage: R1=vector([0,1,0]); R2=vector([2,2,0]); R3=vector([3.5,0,0])
sage: r1 = R1.norm(); r2 = R2.norm(); r3 = R3.norm()
sage: D = R1.cross_product(R2) + R2.cross_product(R3) \
....:   + R3.cross_product(R1)
sage: S = (r1 - r3) * R2 + (r3 - r2) * R1 + (r2 - r1) * R3
sage: N = r3 * R1.cross_product(R2) + r1 * R2.cross_product(R3) \
....:   + r2 * R3.cross_product(R1)
sage: e = S.norm() / D.norm(); p = N.norm() / D.norm()
sage: a = p/(1-e^2); c = a * e; b = sqrt(a^2 - c^2)
sage: X = S.cross_product(D); i = X / X.norm()
sage: phi = atan2(i[1], i[0]) * 180 / pi.n()
sage: print("%.3f %.3f %.3f %.3f %.3f %.3f" % (a, b, c, e, p, phi))
2.360 1.326 1.952 0.827 0.746 17.917
```

Als Erkenntnis gewinnen wir:

$$a \approx 2.360, \quad b \approx 1.326, \quad c \approx 1.952, \quad e \approx 0.827, \quad p \approx 0.746, \quad \varphi \approx 17.917.$$

Die Neigung der großen Achse gegen die Abszisse beträgt 17.92°.

Übung 10 page 37. *(Basen von Untervektorräumen)*

1. Die zu A gehöige Lösungsmenge S eines homogenen linearen Gleichungssystems ist ein Untervektorraum des $\mathbb{R}^5$, von dem wir dank der Funktion `right_kernel` die Dimension und eine Basis erhalten:

   ```
   sage: A = matrix(QQ, [[ 2,  -3,   2, -12, 33],
   ....:                 [ 6,   1,  26, -16, 69],
   ....:                 [10, -29, -18, -53, 32],
   ....:                 [ 2,   0,   8, -18, 84]])
   sage: A.right_kernel()
   ```

```
Vector space of degree 5 and dimension 2 over Rational Field
Basis matrix:
[     1      0  -7/34   5/17   1/17]
[     0      1  -3/34 -10/17  -2/17]
```

S ist daher die von den beiden obigen Vektoren aufgespannte Ebene (die man zeilenweise lesen muss, wie weiter unten).

2. Wir isolieren aus der gegebenen Familie von Erzeugenden auf folgende Weise eine Basis des gesuchten Untervektorraums. Wir reduzieren die Matrix A (mit den Spalten u_1, u_2, u_3, mu_4, u_5) bezüglich der Zeilen auf hermitesche Form:

```
sage: H = A.echelon_form(); H
```

$$\begin{pmatrix} 1 & 0 & 4 & 0 & -3 \\ 0 & 1 & 2 & 0 & 7 \\ 0 & 0 & 0 & 1 & -5 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Sei $F = \mathrm{Vect}(u_1, u_2, u_3, u_4, u_5)$ die Familie der Spaltenvektoren von A. Das ist ein Untervektorraum von R^4. In H bemerken wir, dass sich die Pivotstellen in den Spalten 1, 2 und 4 befinden. Genauer haben wir

$$\begin{cases} (u_1, u_2, u_4) \text{ ist eine linear unabhängige Familie,} \\ u_3 = 4u_1 + 2u_2, \\ u_5 = -3u_1 + 7u_2 - 5u_4 \end{cases}$$

Daher wird $F = \mathrm{Vect}(u_1, u_2, u_3, u_4, u_5) = \mathrm{Vect}(u_1, u_2, u_4)$ von der Familie (u_1, u_2, u_4) erzeugt; nun ist diese Familie linear unabhängig; deshalb ist (u_1, u_2, u_4) eine Basis von F. Wir hätten ebenso, wenn auch direkter, die Funktion `column_space` verwenden können:

```
sage: A.column_space()
Vector space of degree 4 and dimension 3 over Rational Field
Basis matrix:
[       1        0        0 1139/350]
[       0        1        0    -9/50]
[       0        0        1   -12/35]
```

3. Wir suchen jetzt Gleichungen des erzeugten Untervektorraums. Dafür reduzieren wir die um ein zweites Glied erweiterte Matrix A, indem wir Sage anzeigen, dass wir auf einem Polynomring mit vier Unbestimmten arbeiten:

```
sage: S.<x, y, z, t> = QQ[]
sage: C = matrix(S, 4, 1, [x, y, z, t])
sage: B = block_matrix([A, C], ncols=2)
sage: C = B.echelon_form()
sage: C[3,5]*350
-1139*x + 63*y + 120*z + 350*t
```

Daraus folgern wir, dass F eine Hyperebene des $\mathbb{R}^4$ ist mit der Gleichung

$$-1139 * x + 63 * y + 120 * z + 350 * t = 0$$

Diese Gleichung hätten wir auch durch Berechnung des Kerns von A von links bekommen können, was die Koordinaten der Linearformen ergibt, die F definieren (davon gibt es hier nur eine):

```
sage: K = A.left_kernel(); K
sage: Vector space of degree 4 and dimension 1 over Rational Field
sage: Basis matrix:
[        1  -63/1139 -120/1139 -350/1139]
```

Die durch diese Linearform definierte Hyperebene hat als Basis die folgenden drei Vektoren, die wir schon mit `A.column_space()` erhalten hatten:

```
sage: matrix(K.0).right_kernel()
Vector space of degree 4 and dimension 3 over Rational Field
Basis matrix:
[       1        0        0 1139/350]
[       0        1        0    -9/50]
[       0        0        1   -12/35]
```

Übung 11 Seite 37. *(Eine Matrixgleichung)* Wir beginnen mit der Definition der Matrizen A und C.

```
sage: A = matrix(QQ, [[-2, 1, 1], [8, 1, -5], [4, 3, -3]])
sage: C = matrix(QQ, [[1, 2, -1], [2, -1, -1], [-5, 0, 3]])
```

Die Gleichung $A = BC$ ist eine lineare Gleichung, daher ist die Lösungsmenge ein affiner Untervektorraum von $\mathcal{M}_3(\mathbb{R})$. Wir suche also eine partikuläre Lösung unserer Gleichung.

```
sage: B = C.solve_left(A); B
[ 0 -1  0]
[ 2  3  0]
[ 2  1  0]
```

Nun bestimmen wir die allgemeine Form der Lösungen der homogenen Gleichung, oder anders gesagt, den Kern von links von C.

```
sage: C.left_kernel()
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[1 2 1]
```

Schließlich geben wir die allgemeine Form der Lösung unserer Gleichung an:

```
sage: x, y, z = var('x, y, z'); v = matrix([[1, 2, 1]])
sage: B = B + (x*v).stack(y*v).stack(z*v); B
```

Wir können das Ergebnis schnell verifizieren:

```
sage: sage: A == B*C
True
```

Schließlich ist die Lösungsmenge ein affiner Untervektorraum der Dimension 3:

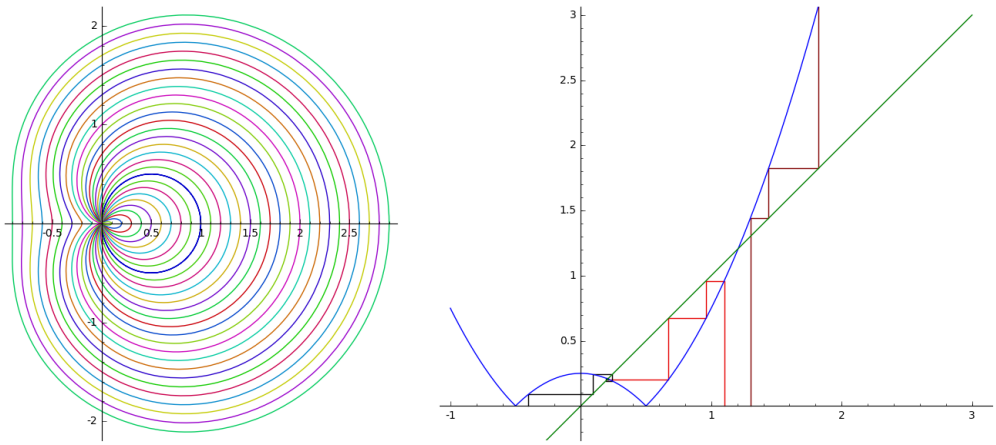
$$\left\{ \begin{pmatrix} x & 2x-1 & x \\ y+2 & 2y+3 & y \\ z+2 & 2z+1 & z \end{pmatrix} \;\middle|\; (x,y,z) \in \mathbb{R}^3 \right\}.$$

A.3. Programmierung und Datenstrukturen

A.4. Graphik

Übung 12 Seite 79. *(Pascal-Conchoïden)*

```
sage: t = var('t'); liste = [a + cos(t) for a in srange(0, 2, 0.1)]
sage: g = polar_plot(liste, (t, 0, 2 * pi)); g.show(aspect_ratio = 1)
```



(a) Pascal-Conchoïden (b) Untersuchung einer rekursiven Folge

Übung 13 page 83. *(Zeichnung der Terme einer rekursiven Folge)*

```
sage: f = lambda x: abs(x**2 - 1/4)
sage: def liste_pts(u0, n):
....:     u = u0; liste = [[u0,0]]
....:     for k in range(n):
....:         v, u = u, f(u)
....:         liste.extend([[v,u], [u,u]])
....:     return(liste)
sage: g = line(liste_pts(1.1, 8), rgbcolor = (.9,0,0))
sage: g += line(liste_pts(-.4, 8), rgbcolor = (.01,0,0))
sage: g += line(liste_pts(1.3, 3), rgbcolor = (.5,0,0))
sage: g += plot(f, -1, 3, rgbcolor = 'blue')
sage: g += plot(x, -1, 3, rgbcolor = 'green')
sage: g.show(aspect_ratio = 1, ymin = -.2, ymax = 3)
```

Übung 14 Seite 85. *(Lineare Differentialgleichung 1. Ordnung)*

```
sage: x = var('x'); y = function('y')
sage: DE = x^2 * diff(y(x), x) - y(x) == 0
sage: desolve(DE, y(x))
_C*e^(-1/x)
sage: g = plot([c*e^(-1/x) for c in srange(-8, 8, 0.4)], (x, -3, 3))
sage: y = var('y')
sage: g += plot_vector_field((x^2, y), (x,-3,3), (y,-5,5))
sage: g.show(ymin=-5, ymax=5)
```

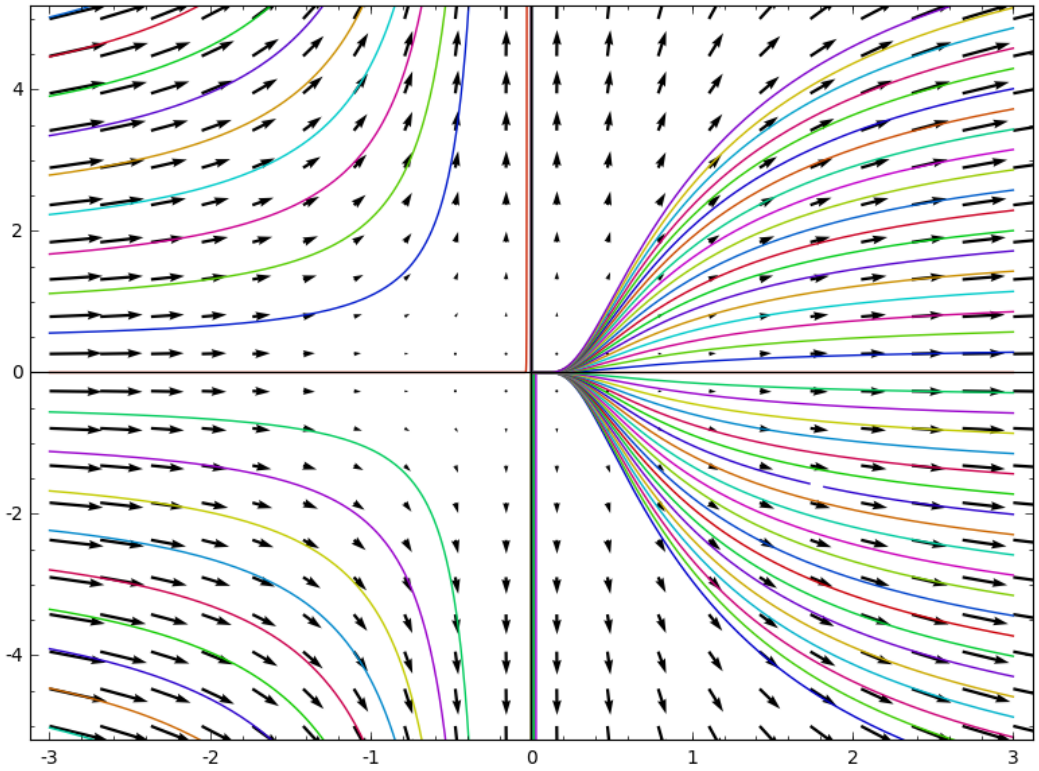


Abb. A.3 - Integralkurven von $x^2y' - y = 0$

Übung 15 Seite 88. *(Modell Beute-Räuber)*

```
sage: from sage.calculus.desolvers import desolve_system_rk4
sage: f = lambda x, y: [a*x-b*x*y,-c*y+d*b*x*y]
sage: x, y, t = var('x, y, t')
sage: a, b, c, d = 1., 0.1, 1.5, 0.75
sage: P = desolve_system_rk4(f(x,y), [x,y],
....:        ics=[0,10,5], ivar=t, end_points=15)
sage: Qh = [[i,j] for i,j,k in P]; p = line(Qh, color='red')
sage: p += text("Hasen", (12,37), fontsize=10, color='red')
sage: Qf = [[i,k] for i,j,k in P]; p += line(Qf, color='blue')
sage: p += text("Fuechse", (12,7), fontsize=10, color='blue')
sage: p.axes_labels(["Zeit", "Population"])
sage: p.show(gridlines = True)
```

Wir können auch die rechte Graphik in Abb. 4.12 reproduzieren:

```
sage: n = 10; L = srange(6, 18, 12/n); R = srange(3, 9, 6/n)
sage: def g(x,y): v = vector(f(x,y)); return v/v.norm()
sage: q = plot_vector_field(g(x, y), (x, 0, 60), (y, 0, 36))
sage: for j in range(n):
....:     P = desolve_system_rk4(f(x,y), [x,y],
....:            ics=[0,L[j],R[j]], ivar=t, end_points=15)
....:     Q = [[j,k] for i,j,k in P]
....:     q += line(Q, color=hue(.8-j/(2*n)))
sage: q.axes_labels(["Hasen", "Fuechse"]); q.show()
```

Übung 16 Seite 88. *(Ein autonomes System von Differentialgleichungen)*

```
sage: from scipy import integrate
sage: def dX_dt(X, t=0): return [X[1], 0.5*X[1] - X[0] - X[1]^3]
sage: t = srange(0, 40, 0.01); x0 = srange(-2, 2, 0.1); y0 = 2.5
sage: CI = [[i, y0] for i in x0] + [[i, -y0] for i in x0]
sage: def g(x,y): v = vector(dX_dt([x, y])); return v / v.norm()
sage: x, y = var('x, y'); n = len(CI)
sage: q = plot_vector_field(g(x, y), (x, -3, 3), (y, -y0, y0))
sage: for j in xrange(n):
....:     X = integrate.odeint(dX_dt, CI[j], t)
sage: q += line(X, color=(1.7*j/(4*n),1.5*j/(4*n),1-3*j/(8*n)))
sage: X = integrate.odeint(dX_dt, [0.01,0], t)
sage: q += line(X, color = 'red'); q.show()
```

Übung 17 Seite 88. *(Strömung um einen Zylinder mit Magnuseffekt)*

Für die Lösung dieser Aufgabe können wir beispielsweise die Funktion `odeint` von Scipy verwenden:

```
sage: from scipy import integrate
sage: t = srange(0, 40, 0.2)
sage: n = 35; CI_cart = [[4, .2 * i] for i in range(n)]
sage: CI = map(lambda x: [sqrt(x[0]^2+x[1]^2),\
....:      pi - arctan(x[1]/x[0])], CI_cart)
sage: for alpha in [0.1, 0.5, 1, 1.25]:
....:     dX_dt = lambda X, t=0: [cos(X[1])*(1-1/X[0]^2), \
....:             -sin(X[1]) * (1/X[0]+1/X[0]^3) + 2*alpha/X[0]^2]
....:     q = circle((0, 0), 1, fill=True, rgbcolor='purple')
....:     for j in range(n):
....:         X = integrate.odeint(dX_dt, CI[j], t)
....:         Y = [[u[0]*cos(u[1]), u[0]*sin(u[1])] for u in X]
....:         q += line(Y, xmin = -4, xmax = 4, color='blue')
....:     q.show(aspect_ratio = 1, axes = False)
```

Abbildung A.5 zeigt die Lösungen zu den Fällen $\alpha = 0.1, 0.5, 1$.

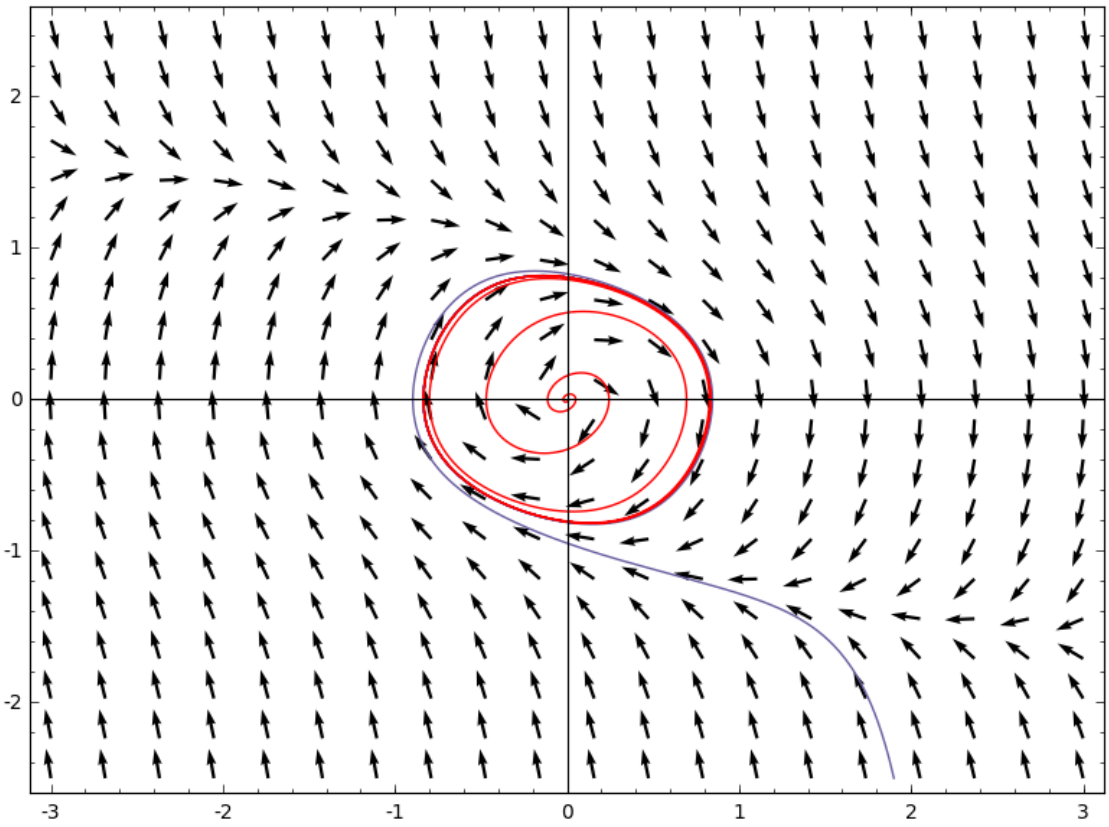
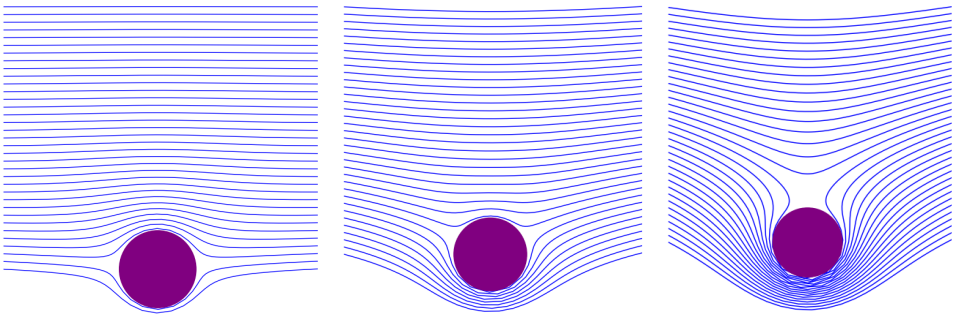


Abb. A.4 - Ein autonomes System von Differentialgleichungen



Fall $\alpha = 0.1$ Fall $\alpha = 0.5$ Fall $\alpha = 1$

Abb. A.5 - Magnuseffekt

A.5. Definitionsmengen

Übung 18 Seite 100. Die Klasse des Ringes `ZZ` heißt `IntegerRing_class`, wie wir mit den folgenden Befehlen sehen können:

```
sage: print type(ZZ)
'sage.rings.integer_ring.IntegerRing_class'>
```

```
sage: ZZ.__class__
'sage.rings.integer_ring.IntegerRing_class'>
```

Tatsächlich ist der Ring `ZZ` die einzige Instanz dieser Klasse, die wir mit der *Kategorie* von `ZZ` nicht verwechseln dürfen

```
sage: ZZ.category()
Join of Category of euclidean domains and Category of infinite
enumerated sets and Category of metric spaces
```

und auch nicht mit mit Klasse seiner Elemente

```
sage: ZZ.an_element().__class__
<type 'sage.rings.integer.Integer'>
```

A.6. Endliche Körper und elementare Zahlentheorie

Übung 19 Seite 121. Wir setzen $n = pqr$ voraus mit $p < q < r$. Dann muss $p^3 \leq n$ sein und die Funktion wird

```
sage: def enum_carmichael(N, verbose=True):
....:     p = 3; s = 0
....:     while p^3 <= N:
....:         s += enum_carmichael_p(N, p, verbose); p = next_prime(p)
....:     return s
```

worin die Funktion `enum_carmichael_p` die Anzahl der Carmichael-Vielfachen zählt, die von der Form $a + \lambda m$ sind mit $\lambda \in \mathbb{N}_0$, $a = p$ und $m = p(p-1)$, weil n ein Vielfaches von p sein muss und $n - 1$ ein Vielfaches von $p - 1$:

```
sage: def enum_carmichael_p (n, p, verbose):
....:     a = p; m = p*(p-1); q = p; s = 0
....:     while p*q^2 <= n:
....:         q = next_prime(q)
....:         s += enum_carmichael_pq(n, a, m, p, q, verbose)
....:     return s
```

Die Funktion `enum_carmichael_pq` zählt die Anzahl der Carmichael-Vielfachen von pq, die von der Form $a' + \mu m'$ sind mit $\mu \in \mathbb{N}_0$, wobei $a' \equiv a \bmod m$ ist, $a' \equiv q \bmod q(q-1)$ und m' gleichzeitig Vielfaches von $m = p(p-1)$ und von $q(q-1)$. Zur Lösung der simultanen Kogruenzen nehmen wir die Funktion `crt`. Dabei schließen wir den Fall aus, dass es keine Lösung geben kann, sonst würde Sage einen Fehler verursachen. Wir setzen auch $a' > pq^2$, um $r > q$ zu haben:

```
sage: def enum_carmichael_pq(n,a,m,p,q,verbose):
....:     if (a-q) % gcd(m,q*(q-1)) <> 0: return 0
....:     s = 0
....:     a = crt (a, q, m, q*(q-1)); m = lcm(m,q*(q-1))
....:     while a <= p*q^2: a += m
....:     for t in range(a, n+1, m):
....:         r = t // (p*q)
```

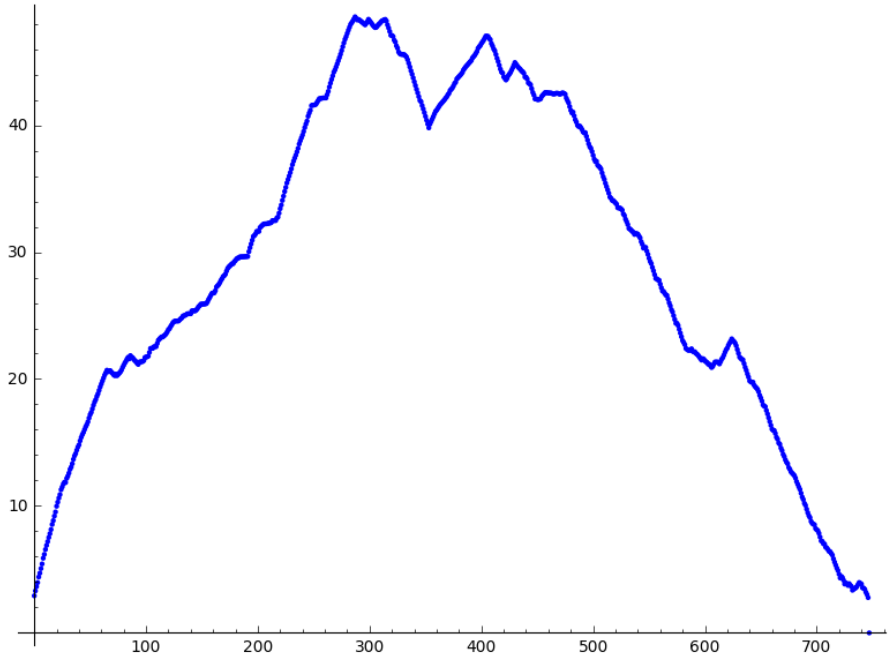
```
....:         if is_prime(r) and t % (r-1) == 1:
....:             if verbose:
....:                 print p*q*r, factor(p*q*r)
....:         s += 1
....:     return s
```

Mit diesen Funktionen erhalten wir

```
sage: enum_carmichael(10^4)
561 3 * 11 * 17
1105 5 * 13 * 17
2465 5 * 17 * 29
1729 7 * 13 * 19
2821 7 * 13 * 31
8911 7 * 19 * 67
6601 7 * 23 * 41
7 sage: enum_carmichael(10^5, False)
12 sage: enum_carmichael(10^6, False)
23 sage: enum_carmichael(10^7, False)
47
```

Übung 20 Seite 123. Zunächst schreiben wir eine Funktion `aliq`, welche für n die aliquote Folge berechnet und stoppt, sobald wir 1 erhalten oder eine Wiederholung auftritt:

```
sage: n = 840
sage: s = [n]
sage: while n > 1:
....:     n = sigma(n) - n
....:     s.append(n)
....: print s[:5], s[-5:]
(748, [840, 2040, 4440, 9240, 25320], [2714, 1606, 1058, 601, 1])
sage: points([(x,log(s[x])/log(10.0)) for x in range(len(s))])
```



Übung 21 Seite 124. *Konstante von Masser-Gramain* Bei Frage 1 sei C der Grenzkreis einer kleineren Kreisscheibe. Ohne Beschränkung der Allgemeinheit können wir annehmen, dass der Ursprung O auf dem Kreis liegt - tatsächlich gibt es mindestens einen Punkt von $\mathbb{Z}^2$ auf dem Kreis, sonst ist die Scheibe nicht optimal. Wir können auch voraussetzen, dass der Kreismittelpunkt im ersten Quadranten liegt (durch Rotation der Scheibe um ein Vielfaches von $\pi/2$) um O). Wir werden zugestehen, dass wir ebenfalls zwei Punkte A und B im ersten Quadranten auf dem Kreis haben, und der Kreis somit Umkreis des Dreiecks OAB ist. Die Schranke $r_k < \sqrt{\frac{k}{\pi}}$ ermöglicht, die Punkte A und B einzugrenzen, denn ihr Abstand zu O ist höchstens $\sqrt{\frac{2k}{\pi}}$. Wir können annehmen, dass einer der Punkte A und B, beispielsweise A, im zweiten Oktanten liegt (wenn beide wegen der Symmetrie zur Geraden $x = y$ im ersten Oktanten liegen, bringen wir sie in den zweiten Oktanten). Wir können auch annehmen, dass der Winkel OAB spitz ist (indem wir A und B notfalls vertauschen, wegen der Symmetrie zur Geraden $x = y$ liegen sie in verschiedenen Oktanten). Die Abszisse von A veriiziert deshalb $x_A < \sqrt{\frac{2k}{\pi}}$, seine Ordinate verifiziert $x_A \leq y_A < \sqrt{4k/\pi - x_A^2}$. Für den Punkt B haben wir $0 \leq x_B < 2\sqrt{\frac{k}{\pi}}$ und $0 \leq x_A y_B + y y_A x_B \leq x_A^2 + y_A^2$ (spitzer Winkel bei A). Das ergibt den folgenden Code, wobei die Routine `rk_aux` die Anzahl der Punkte in der Scheibe mit dem Zentrum $x_c/d, y_c/d$ und dem Radius $\sqrt{r_2/d}$, wobei x_c, y_c, d und r_2 sämtlich ganzzahlig sind.

```
sage: def rk_aux(xc, yc, d, r2):
....:     s = 0
....:      xmin = ceil((xc - sqrt(r2))/d)
....:      xmax = floor((xc + sqrt(r2))/d)
....:      for x in range(xmin,xmax+1):
....:          r3 = r2 - (d*x-xc)^2 # (d*y-yc)^2 <= r2 - (d*x-xc)^2
....:          ymin = ceil((yc - sqrt(r3))/d)
....:          ymax = floor((yc + sqrt(r3))/d)
....:          s += ymax + 1 - ymin
....:      return s
```

```
sage: def rk(k): # renvoie (r_k^2, xc, yc)
....:     if k == 2: return 1/4, 1/2, 0
....:     dmax = (2*sqrt(k/pi)).n(); xamax = (sqrt(2*k/pi)).n()
....:     sol = (dmax/2)^2, 0, 0, 0
....:     for xa in range(0, floor(xamax)+1):
....:         # falls xa=0, ya > 0 den A kann nicht auf 0 liegen
....:         yamin = max(xa, 1)
....:         for ya in range(yamin, floor(sqrt(dmax^2-xa^2))+1):
....:             xbmin = 0 # wir wollen xb*ya <= xa^2+ya^2
....:             if xa == 0:
....:                 xbmin = 1 # 0, A, B dürfen nicht über- oder nebeneinander liegen
....:             xbmax = min(floor(dmax), floor((xa*xa+ya*ya)/ya))
....:             for xb in range(xbmin, xbmax+1):
....:                 ybmax = floor(sqrt(dmax^2-xb^2))
....:                 if xa > 0: # wir wollen xb*ya+yb*xa <= xa^2+ya^2
....:                     tmp = floor((xa*xa+ya*ya-xb*ya)/xa)
....:                     ybmax = min(ybmax, tmp)
....:                 # si xb=0, yb > 0 denn B muss verschieden sein von 0
....:                 ybmin = 0
....:                 if xb == 0:
....:                     ybmin = 1
....:                 for yb in range(ybmin,ybmax+1):
....:                     d = 2*abs(xb*ya - xa*yb)
....:                     if d <> 0:
....:                         ra2 = xa^2+ya^2; rb2 = xb^2+yb^2
....:                         xc = abs(ra2*yb - rb2*ya)
....:                         yc = abs(rb2*xa - ra2*xb)
....:                         r2 = ra2*rb2*((xa-xb)^2+(ya-yb)^2)
....:                         m = rk_aux(xc,yc,d,r2)
....:                         if m >= k and r2/d^2 < sol[0]:
....:                             sol = r2/d^2, xc/d, yc/d
....:     return sol


sage: for k in range(2,10): print k, rk(k)
2 (1/4, 1/2, 0)
3 (1/2, 1/2, 1/2)
4 (1/2, 1/2, 1/2)
5 (1, 0, 1)
6 (5/4, 1/2, 1)
7 (25/16, 3/4, 1)
8 (2, 1, 1)
9 (2, 1, 1)
```

Die Lösung zu Frage 2 ist die folgende:

```
sage: def plotrk(k):
....:     r2, x0, y0 = rk(k); r = n(sqrt(r2))
....:     var('x, y')
....:     c = implicit_plot((x-x0)^2+(y-y0)^2-r2,
....:         (x, x0-r-1/2, x0+r+1/2),(y, y0-r-1/2, y0+r+1/2))
```

```
....:     center = points([(x0,y0)], pointsize=50, color='black')
....:     # wir wollen (i-x0)^2+(j-y0)^2 <= r2
....:     # daher |i-x0| <= r et |j-y0| <= r2 - (i-x0)^2
....:     l = [(i, j) for i in range(ceil(x0-r), floor(x0+r)+1)
....:                 for j in range(ceil(y0-sqrt(r^2-(i-x0)^2)),
....:                                floor(y0+sqrt(r2-(i-x0)^2))+1)]
....:     d = points(l, pointsize=100)
....:     return (c+center+d).show(aspect_ratio=1, axes=True)
```

Frage 3 erfordet etwas Überlegung. Wir schreiben $S_{i,j} = \sum_{k=i}^{j} 1/(\pi r_k^2)$. Ausgehend von der oberen Schranke (6.2) für r_k erhalten wir $r_k^2 < (k-1)/\pi$, somit ist $1/(\mathbf{i}r_k^2) > 1/(k-1)$ und $S_{n,N} > \sum_{k=n}^{N} 1/(k-1) > \int_n^{N+1} \mathrm{d}k/k = \log((N+1)/n)$.

Die untere Schranke von (6.2) ergibt $1/(\pi r_k^2) < 1/k + 2/k^{3/2}$ für $k \geq 407$, was für $n \geq 407$ zu $S_{n,N} < \sum_{k=n}^{N}(1/k + 2/k^{3/2}) < \int_{n-1}^{N}(1/k + 2/k^{3/2})\mathrm{d}k = \log(N/(n-1)) + 4/\sqrt{n-1} - 4/\sqrt{N}$ führt, woraus folgt

$$S_{2,n-1} + \log(1/n) \leq \delta \leq S_{2,n-1} + \log(1/(n-1)) + 4\sqrt{n-1}.$$

```
sage: def bound(n):
....:     s = sum(1/pi/rk(k)[0] for k in range(2,n+1))
....:     return float(s+log(1/n)), float(s+log(1/(n-1))+4/sqrt(n-1))
sage: bound(60)
(1.7327473659779615, 2.2703101282176377)
```

Daraus leiten wir $1.73 < \delta < 2.28$ her. Also ist der Näherungswert $\delta \approx 2.00$ mit einem absoluten Fehler unter 0.28.

Übung 22 Seite 125. Wir benutzen hier wieder dieselben Begriffe wie der Artikel von Beauzami [Bea09]. Wir setzen $s_i = 1 - x_i - \ldots - x_k$ mit $_{k+1} = 1$. Dann muss $x_1 + \ldots + x_{i-1} \leq s_i$ gelten und insbesondere ist $x_2 \leq x_1 \leq s_2$. Es sein

$$C_1 = \int_{x_1=x_2}^{s_2} x_1^{n_1} \mathrm{d}x_1 = \frac{1}{n_1+1}\left(s_2^{n_1+1} - x_2^{n_1+1}\right).$$

```
sage: x1, x2, s2 = var('x1, x2, s2')
sage: n1 = 9; C1 = integrate(x1^n1, x1, x2, s2); C1
1/10*s2^10 - 1/10*x2^10
```

Dann haben wir $x_3 \leq x_2 \leq s_3 = s_2 + x_2$, ersetzen wir deshalb s_2 durch $s_3 - s_2$ in C_1 und integrieren über x_2 von x_3 bis $s_2/2$ - denn $x_1 + x_2 \leq s_3$ und $x_2 \leq x_1$ - erhalten wir:

```
sage: x3, s3 = var('x3, s3')
sage: n2 = 7; C2 = integrate(C1.subs(s2=s3-x2)*x2^n2, x2, x3, s3/2); C2
44923/229417943040*s3^18 - 1/80*s3^10*x3^8 + 1/9*s3^9*x3^9 -
9/20*s3^8*x3^10 + 12/11*s3^7*x3^11 - 7/4*s3^6*x3^12 + 126/65*s3^5*x3^13
- 3/2*s3^4*x3^14 + 4/5*s3^3*x3^15 - 9/32*s3^2*x3^16 + 1/17*s3*x3^17
```

und so fort. Bei jeder Iteration ist C_i ein homogenes Polynom in x_{i+1} und s_{i+1} mit rationalen Koeffizienten und insgeamt vom Grad $n_1 + \ldots + n_i + i$. Für die letzte Variable integrieren wir von $x_k = 0$ bis $x_k = 1/k$.

Nehmen wir für Zähler und Nenner von I eine bekannte Schranke an, können wir I modulo p für verschiedene Primzahlen berechnen, die den Nenner von I nicht teilen und daraus dann mit den chinesischen Resten den Wert von I modulo dem Produkt dieser Primzahlen herleiten und schließlich mit rationaler Rekonstruktion den exakten Wert von I ermitteln.

A.7. Polynome

Übung 23 Seite 129.

1. Wir können zum Beispiel nehmen (es gibt viele andere Lösungen):

   ```
   sage: x = polygen(QQ, 'y'); y = polygen(QQ, 'x')
   ```

   Erinnern wir uns an den Unterschied in Sage zwischen Python-Variablen und mathematischen Variablen. Python-Variablen sind Namen, die zum Programmieren dienen und bezeichnen den Speicherplatz im Rechner. Mathematische Variablen, darunter die Unbestimmten der Polynome, sind von völlig anderer Art: es sind Sage-Objekte, die in Python-Variablen *gespeichert* werden können. Wenn wir eine `'x'` genannte Unbestimmte erzeugen, verpflichtet nichts dazu, in der Python-Variablen `x` zu speichern - und nichts hindert daran, dort `'y'` unterzubringen.

2. Wir fangen damit an, der Python-Variablen `x` die Unbestimmte `'x'` der Polynome mit rationalen Koeffizienten zuzuweisen. Der Ausdruck `x+1` wird dann im Polynom $x + 1 \in \mathbb{Q}[x]$ ausgewertet, die wir der Variablen `p` zugewiesen haben. Dann weisen wir der Variablen `x` die Zahl 2 zu. Das hat keinen Einfluss auf `p`, das immer noch $x+1$ ist: dieses x hier (die Unbestimmte) hat mit der Python-Variablen nichts zu tun, die ist jetzt 2. In diesem Stadium wird `p+x` zu $x + 3$ ausgewertet. Daher ist der Endwert von `p` gleich $x + 3$.

Übung 24 Seite 134. Eine einfache Lösung besteht darin, sukzessive euklidsche Divisionen durch Tschebyschow-Polynome fallenden Grades durchzuführen: wenn dass Polynom p, das in die Basis von Tschebyschow umgeschrieben werden soll, vom Grad n ist, setzen wir $p = c_n T_n + R_{n-1}$ mit $c_n \in \mathbb{Q}$ und $\deg R_{n-1} \leq n - 1$, dann $R_{n-1} = c_{n-1} T_{n-1} + R_{n-1}$ und so weiter.

Beim folgenden Sage-Code haben wir uns entschieden, statt die als els einfache Liste erhaltenen Koeffizienten c_n zurückzugeben, einen symbolischen Ausdruck zu bilden, wobei das Polynom T_n als eine „inerte“ Funktion `T(n,x)` dargestellt wird (d.h. als nicht ausgewertete Form betrachtet)

```
sage: T = sage.symbolic.function_factory.function('T', nargs=2)
sage: def to_chebyshev_basis(pol):
....:     (x,) = pol.variables()
....:     res = 0
....:     for n in xrange(pol.degree(), -1, -1):
....:         quo, pol = pol.quo_rem(chebyshev_T(n, x))
....:         res += quo * T(n, x)
....:     return res
```

Testen wir diese Funktion. Um die Resultate zu verifizieren, reicht es aus, in unsere inerte Funktion `T` die Funktion einzusetzen, welche die Tschebyschow-Polynome berechnet und zu entwickeln:

```
sage: p = QQ['x'].random_element(degree=6); p
2*x^6 - x^5 + x^3 + x^2 + 1/5*x + 1
sage: p_cheb = to_chebyshev_basis(p); p_cheb
1/16*T(6, x) - 1/16*T(5, x) + 3/8*T(4, x) - 1/16*T(3, x) + 23/16*T(2, x) + 13/40*T(1, x)
+ 13/40*T(1, x) + 17/8*T(0, x)
sage: p_cheb.substitute_function(T, chebyshev_T).expand()
2*x^6 - x^5 + x^3 + x^2 + 1/5*x + 1
```

Übung 25 Seite 134. Eine direkte Übertragung nach Sage ergibt so etwas:

```
sage: def mydiv(u, v, n):
....:     v0 = v.constant_coefficient()
....:     quo = 0; rem = u
....:     for k in xrange(n+1):
....:         c = rem[0]/v0
....:         rem = (rem - c*v) >> 1 # bitweise Rechtsverschiebung
 ....:         quo += c*x^k
....:     return quo, rem
```

(Wir werden die Laufzeit dieser Funktion anhand recht großer Beispiele messen können und versuchen, den Code effizienter zu machen, ohne den Algorithmus zu verändern.)

Doch der Quotient in der Division durch bis zur Ordnung n wachsende Potenzen ist einfach die Reihenentwicklung des gebrochen rationalen Ausdrucks u/v, die mit der Ordnung $n+1$ abgebrochen wird. Mit der Division der formalen Potenzreihen (siehe Abschnitt 7.5) können wir daher die Division durch steigende Potenzen wie folgt berechnen.

```
sage: def mydiv2(u, v, n):
....:     x = u.parent().gen()
....:     quo = (u / (v + O(x^(n+1)))).polynomial()
....:     rem = (u - quo*v) >> (n+1)
....:     return quo, rem
```

Die Zeile `quo = ...` konvertiert erstens ein Polynom in eine abbrechende Reihe, indem sie ein $\mathcal{O}(.)$ anhängt und zweitens erfolgt die Division durch eine Reihe in der Voreinstellung mit der Genauigkeit des Divisors.

Übung 26 Seite 135. Zunächst sind wir darauf gefasst, dass $u_{10^{10000}}$ Ziffern in der Größenordnung 10000 hat. Eine Berechnung als ganze Zahl kommt somit überhaupt nicht in Frage, Deshalb interessieren wir uns nur für die letzten fünf Ziffern, und das ist wirklich kein Problem: wir machen die ganze Rechnung modulo 10^5. Das in Unterabschnitt 3.2.4 vorgestellte Verfahren zur schnellen Potenzierung verlangt nun einige zigtausend Multiplikationen von 1000×1000-Matrizen mit Koeffizienten aus $\mathbb{Z}/10^5\mathbb{Z}$. Jedes dieser Matrizenprodukte kommt auf eine Milliarde Multiplikationen modulo 10^5 oder etwas weniger mit einem schnellen Algorithmus. Das is nicht völlig unmöglich, aber der Versuch einer einzigen Multiplikation lässt vermuten, dass die Berechnung mit Sage eine gute Stunde dauern würde:

```
sage: Mat = MatrixSpace (IntegerModRing(10^5), 1000)
sage: m1, m2 = (Mat.random_element() for i in (1,2))
sage: time m1*m2
1000 x 1000 dense matrix over Ring of integers modulo 100000 (use the
'.str()' method to see the entries)
Time: CPU 0.37 s, Wall: 0.39 s
```

Was den Algorithmus betrifft, gibt es eine bessere Möglichkeit. Mit S bezeichnen wir den Verschiebungsoperator $(a_n)_{n\in\mathbb{N}} \mapsto (a_{n+1})_{n\in\mathbb{N}}$. Die $u = (u_n)_{n\in\mathbb{N}}$ genügende Gleichung wird nun $P(S)\cdot u = 0$ geschrieben, wobei $P(x) = x^{1000} - 23x^{729} + 5x^2 - 12x - 7$ ist; und für jedes N (insbesondere $N = 10^{100}$) ist der Term u_N das erste Glied der Folge $S^n \cdot u$. Sei R der Rest der euklidschen Division von x^N durch P. Wegen $P(S)\cdot u = 0$ haben wir $S^N \cdot u = R(S)\cdot u$. Deshalb genügt die Berechnung des Bildes von x^N in $(\mathbb{Z}/10^5\mathbb{Z})[x]/\langle P(x)\rangle$. Wir gelangen zu folgendem Code, der auf derselben Maschine in weniger als einer halben Minute ausgeführt wird:

```
sage: Poly.<x> = Integers(10^5)[]
sage: P = x^1000 - 23*x^729 + 5*x^2 - 12*x - 7
sage: Quo.<s> = Poly.quo(P)
sage: op = s^(10^10000)
sage: add(op[n]*(n+7) for n in range(1000))
63477
```

Die gesuchten fünf letzten Ziffern sind also 63477. Der Unterschied zwischen beiden Versionen bei der Rechenzeit wächst mit der Rekursionstiefe rapide an.

Übung 27 Seite 146.

1. Nehmen wir an, dass $a_s u_{n+s} + a_{s-1} u_{n+s-1} + \cdots + a_0 u_n = 0$ ist für alle $n \geq 0$ und nennen wir $u(z) = \sum_{n=0}^{i} nfty$. Sei $Q(z) = a_s + a_{s-1} z + \ldots + a_0 z^8$. Dann ist

$$S(z) = Q(z)u(z) = \sum_{n=0}^{\infty} (a_s u_n + a_{s-1} u_{n-1} + \ldots + a_0 u_{n-s}) z^n$$

   mit der Konvention, dass $u_n = 0$ ist für $n < 0$. Der Koeffizient von z^n in $S(z)$ ist null für $n \geq s$. Daher ist $S(z)$ ein Polynom und $u(z) = S(z)/Q(z)$. Der Nenner $Q(z)$ ist das reziproke Polynom des charakteristischen Polynoms der Rekursion, und der Zähler kodiert die Anfangsbedingungen.

2. Die ersten Koeffizienten reichen hin, eine Rekursion 3. Ordnung zu erraten, die den gegebenen Daten offenbar genügen. Durch Aufruf von `rational_reconstruct` bekommen wir einen gebrochen rationalen Ausdruck, mit dem wir eine Reihe entwickeln können, in der alle gegebenen Koeffizienten wiederzufinden sind sowie die wahrscheinlich dann folgenden:

```
sage: p = previous_prime(2^30); ZpZx.<x> = Integers(p)[]
sage: s = ZpZx([1, 1, 2, 3, 8, 11, 34, 39, 148, 127, 662, 339])
sage: num, den = s.rational_reconstruct(x^12, 6, 6)
sage: S = ZpZx.completion(x)420
sage: map(lift_sym, S(num)/S(den))
[1, 1, 2, 3, 8, 11, 34, 39, 148, 127, 662, 339, 3056, 371,
14602, -4257, 72268, -50489, 369854, -396981]
```

   (Die Funktion `lift_sym` wurde in Kapitel 7 definiert. Die ersten 20 Koeffizienten liegen weit unter 2^{29}, sodass wir die Rekursion modulo 2^{30} entwickeln dürfen, nachdem wir das Ergebnis in $\mathbb{Z}$ erstellt haben.)

   Mit `berlekamp_massey` ist das Ergebnis das charakteristische Polynom der Rekursion direkt mit Koeffizienten in $\mathbb{Z}$:

```
sage: berlekamp_massey([1, 1, 2, 3, 8, 11, 34, 39, 148, 127])
x^3 - 5*x + 2
```

   Wir verifizieren, dass alle gegebenen Koeffizienten der Gleichung $u_{n+3} = 5u_{n+1} - 2u_n$ genügen, und davon ausgehend erraten wir die fehlenden Koeffizienten $72268 = 5 \cdot 14602 - 2 \cdot 371. - 50489 = 5 \cdot (-4257) - 2 \cdot 14602$ und so weiter.

Übung 28 Seite 146. Wir beginnen mit der Berechnung eines Polynoms 3. Grades, dass die gegebene Interpolationsbedingung erfüllt. Das ergibt eine Lösung mit $\deg p = 3$:

```
sage: R.<x> = GF(17)[]
sage: pairs = [(0,-1), (1,0), (2,7), (3,5)]
sage: s = R(QQ['x'].lagrange_polynomial(pairs)); s
6*x^3 + 2*x^2 + 10*x + 16
sage: [s(i) for i in range(4)]
[16, 0, 7, 5]
```

Wir werden damit wieder auf das Problem der rationalen Rekonstruktion verwiesen

$$p/q \equiv s \bmod x(x-1)(x-2)(x-3).$$

Da s nicht invertierbar modulo $x(x-1)(x-2)(x-3)$ ist (denn $s(1)=0$), gibt es für konstantes p keine Lösung. Mit $\deg p = 1$ finden wir:

```
sage: s.rational_reconstruct(mul(x-i for i in range(4)), 1, 2)
(15*x + 2, x^2 + 11*x + 15)
```

Übung 29 Seite 150. Der Ablauf ist derselbe wie beim Beispiel im Text: wir codieren die Gleichung $\tan x = \int_0^x (1+\tan^2 t)\mathrm{d}t$ und suchen ausgehend von der Anfangsbedingung $\tan(0)=0$ einen Fixpunkt.

```
sage: S.<x> = PowerSeriesRing(QQ)
sage: t = S(0)
sage: for i in range(7): # hier ist t korrekt bis zum Grad 2i+1
....:     # O(x^15) vermeidet das Anwachsen der Abbruchordnung
....:     t = (1+t^2).integral() + O(x^15)
sage: t
x + 1/3*x^3 + 2/15*x^5 + 17/315*x^7 + 62/2835*x^9 + 1382/155925*x^11 +
21844/6081075*x^13 + O(x^15)
```

A.8. Lineare Algebra

Übung 30 Seite 170. *(Minimalpolynom von Vektoren)*

1. φ_A ist ein Annihilator-Polynom aller Basisvektoren e_i. Es ist deshalb ein gemeinsames Vielfaches von φ_{A,e_i}. Sei ψ das kgV von φ_{A,e_i}. Dann gilt $\psi \mid \varphi_A$. Außderdem ist $\psi(A) = [\psi(A)e_1 \cdot \ldots \cdot \psi(A)e_n] = 0$ Annihilator der Matrix A. Daraus folgt $\varphi_A \mid \psi$. Da die Polynome unitär sind, sind sie gleich.

2. In diesem Fall haben alle φ_{A,e_i} die Form χ^{l_i}, wobei χ ein irreduzibles Polynom ist. Gemäß der vorigen Frage fällt φ_A mit dem der χl_i zusammen und hat den maximalen Exponenten l_i.

3. Sei φ ein Annihilator-Polynom von $e = e_i + e_j$ und $\varphi_1 = \varphi_{A,e_i}$ und $\varphi_2 = \varphi_{A,e_j}$. Wir haben $\varphi_2(A)\varphi_{A,e_i} = \varphi_2(A)\varphi(A)e - \varphi(A)\varphi_2(A)e_j = 0$. Daher ist $\varphi_2\varphi$ Annihilator von e_i und somit teilbar durch φ_1. Da nun φ_1 und φ_2 teilerfremd sind, gilt $\varphi_1 \mid \varphi$. Genauso beweist man $\varphi_2 \mid \varphi$, weshalb φ ein Vielfaches von $\varphi_1\varphi_2$ ist. Nun ist $\varphi_1\varphi_2$ Annihilator von e, deshalb ist $\varphi = \varphi_1\varphi_2$.

4. Da P_1 und P_2 teilerfremd sind, existieren zwei Polynome α und β, sodass $1 = \alpha P_1 + \beta P_2$. So haben wir $x = \alpha(A)P_1(A)x + \beta(A)P_2(A)x = x_2 + x_1$ für jedes x, wobei $x_1 = \beta(A)P_2(A)x$ ist und $x_2 = \alpha(A)P_1(A)x$. Wegen $\varphi_A = P_1P_2$ ist P_1 Annihilator von $x_1 = \beta(A)P_2(A)x$ (ebenso ist P_2 Annihilator von x_2). Ist $x_1 = 0$ für jedes x, dann ist βP_2 Annihilator von A und ist somit ein Vielfaches von P_1P_2, woraus $1 = P_1(\alpha + \gamma P_2)$ folgt, was $\deg P_1 = 0$ impliziert. Es existiert daher ein $x_1 \neq 0$, sodass P_1 ein Annihilator-Polynom von x_1 ist. Zeigen wir, dass P_1 für x_1 minimal ist: sei $\tilde{P}_1$ ein Annihilator-Polynom von x_1. Dann ist $\tilde{P}_1(A)P_2(A)x = P_2(A)\tilde{P}_1(A)x_1 + \tilde{1}(A)P_2(A)x_2 = 0$, mithin ist $\tilde{P}_1P_2$ ein Vielfaches von $\varphi_A = P_1P_2$. So gilt $P_1 \mid \tilde{P}_1$ und P_1 ist das Minimalpolynom von x_1. Für x_2 geht der Beweis genauso.

5. Zu jedem Faktor $\varphi_i^{m_i}$ existiert ein Vektor x_i. Dabei ist $\varphi_i^{m_i}$ das Minimalpolynom und der Vektor $x_1 + \ldots + x_k$ hat φ_A als Minimalpolynom.

6. Wir berechnen zunächst das Minimalpolynom der Matrix A.

```
sage: A = matrix(GF(7),[[0,0,3,0,0],[1,0,6,0,0],[0,1,5,0,0],
....:                   [0,0,0,0,5],[0,0,0,1,5]])
sage: P = A.minpoly(); P
x^5 + 4*x^4 + 3*x^2 + 3*x + 1
sage: P.factor()
(x^2 + 2*x + 2) * (x^3 + 2*x^2 + x + 4)
```

   Es ist vom höchsten Grad.

```
sage: e1 = identity_matrix(GF(7),5)[0]
sage: e4 = identity_matrix(GF(7),5)[3]
sage: A.transpose().maxspin(e1)
[(1, 0, 0, 0, 0), (0, 1, 0, 0, 0), (0, 0, 1, 0, 0)]
sage: A.transpose().maxspin(e4)
[(0, 0, 0, 1, 0), (0, 0, 0, 0, 1)]
sage: A.transpose().maxspin(e1 + e4)
[(1, 0, 0, 1, 0), (0, 1, 0, 0, 1), (0, 0, 1, 5, 5),
(3, 6, 5, 4, 2), (1, 5, 3, 3, 0)]
```

   Die Funktion `maxspin` iteriert einen Vektor von links. Wir wenden sie deshalb auf die Transponierte von A an, um ausgehend von den Vektoren e_1 und e_4 die Liste der linear unabhängigen Krylow-Iterierten zu bekommen. Das Minimalpolynom von e_1 hat daher den Grad 3, das von e_4 den Grad 2 und das von $e_1 + e_4$ den Grad 5.

   Wir sehen, dass die spezielle Gestalt der Matrix die Vektoren e_1 und e_4 wie Iterierte anderer Vektoren der kanonischen Basis wirken lässt, Diese Form heißt Frobenius-Normalform (siehe Unterabschnitt 8.2.3). Sie beschreibt, wie die Matrix den Vektorraum in zyklische invariante unterräume zerlegt, die von den Vektoren der kanonischen Basis erzeugt werden.

Übung 31 Seite 177.

```
sage: def sind_aehnlich(A, B):
....:     F1, U1 = A.frobenius(2)
....:     F2, U2 = B.frobenius(2)
....:     if F1 == F2:
....:         return True, ~U2*U1
....:     else:
....:         return False, F1 - F2
sage: B = matrix(ZZ, [[0,1,4,0,4],[4,-2,0,-4,-2],[0,0,0,2,1],
....:                 [-4,2,2,0,-1],[-4,-2,1,2,0]])
sage: U = matrix(ZZ, [[3,3,-9,-14,40],[-1,-2,4,2,1],[2,4,-7,-1,-13],
....:                 [-1,0,1,4,-15],[-4,-13,26,8,30]])
sage: A = (U^-1 * B * U).change_ring(ZZ)
sage: ok, V = sind_aehnlich(A, B); ok
True
sage: V
[                1    2824643/1601680   -6818729/1601680
```

```
-43439399/11211760  73108601/11211760]
[                 0      342591/320336     -695773/320336
-2360063/11211760  -10291875/2242352]
[                 0     -367393/640672      673091/640672
-888723/4484704   15889341/4484704]
[                 0     661457/3203360    -565971/3203360
13485411/22423520 -69159661/22423520]
[                 0   -4846439/3203360    7915157/3203360
-32420037/22423520 285914347/22423520]
sage: ok, V = sind_aehnlich(2*A, B); ok
False
```

A.9. Polynomiale Systeme

Übung 32 Seite 180. Zu einem gegebenen Polynomring `ring` gibt die Funktion `test_poly` die Summe aller Monome des gesamten Grades zurück, der durch den Wert des Parameters `deg` begrenzt ist. Der Code ist relativ kompakt, arbeitetet jedoch mit einigen Verrenkungen.

Die erste Anweisung erzeugt eine Menge (dargestellt durch ein spezifisches Objekt `SubMultiset`, siehe Abschnitt 15.2) von Listen zu jedem `deg` und weist sie der lokalen Variablen `monomials` zu, mit Elementen, deren Produkt einen Term des Polynoms bildet:

```
sage: ring = QQ['x,y,z']; deg = 2
sage: tmp1 = [(x,)*deg for x in (1,) + ring.gens()]; tmp1
[(1, 1), (x, x), (y, y), (z, z)]
sage: tmp2 = flatten(tmp1); tmp2
[1, 1, x, x, y, y, z, z]
sage: monomials = Subsets(tmp2, deg, submultiset=True); monomials
SubMultiset of [y, y, 1, 1, z, z, x, x] of size 2
sage: monomials.list()
```

$$[[y,y],[y,1],[y,z],[y,x],[1,1],[1,z],[1,x],[z,z],[z,x],[x,x]]$$

Dafür beginnen damit, dem Tupel der Unbestimmten 1 hinzuzufügen und jedes Element des Ergebnisses durch ein Tupel von `deg` Kopien von sich selbst zu ersetzen und dan diese Tupel in einer Liste zu gruppieren. Wir bemerken die Verwendung von `(1,)`. das ein Tupel mit nur einem Element bezeichnet sowie Operatoren für die Verkettung und für die Wiederholung *. Die erhalten Liste der Tupel wird durch `flatten` zu einer Liste gemacht, die jede Unbestimmte und die Konstante 1 genau `deg` mal enthält. Die Funktion `Subsets` mit der Option `submultiset=True` berechnet sodann die Menge der Teilmengen der Größe `deg` der Multimenge (eine Kollektion ohne Ordnung aber mit Wiederholung) der Elemente dieser Liste. Das Objekt `monomials` ist iterierbar: so ist `(mul(m) for m in monomials)` ein Python-Generator, der die Monome durchläuft, die beim Durchgehen der Listen mit `mul` entstanden sind und die Teilmengen darstellt. Dieser Generator wird schließlich `add` übergeben.

Die letzte Zeile könnte durch `add(map(mul, monomials))` ersetzt werden. Wir könnten auch `((1,) + ring.gens())*deg` schreiben, um den Ausdruck `[(x,)*deg for x in (1,) + ring.gens()]` zu vereinfachen.

Übung 33 Seite 181. Ein Beispiel von der Hilfeseite `PolynomialRing?` schlägt eine Lösung vor: um eine Familie von komplizierten Unbestimmten zu erhalten - hier durch Primzahlen

indiziert - übergibt man `PolynomialRing` eine mit einer Raffung erzeugte Liste (siehe Unterabschnitt 3.3.2):

```
sage: ['x%d' % n for n in [2,3,5,7]]
['x2', 'x3', 'x5', 'x7']
sage: R = PolynomialRing(QQ, ['x%d' % n for n in primes(40)])
sage: R.inject_variables()
Defining x2, x3, x5, x7, x11, x13, x17, x19, x23, x29, x31, x37
```

Die Methode `inject_variables` initialisiert die Python-Variablen `x2,x3,...`, die jeweils den entsprechenden Generator von `R` enthalten.

Übung 34 Seite 186. Wir verifizieren, dass $(3, 2, 1)$ die einzige Lösung ist, beispielsweise mit

```
sage: R.<x,y,z> = QQ[]
sage: J = R.ideal(x^2*y*z-18, x*y^3*z-24, x*y*z^4-6)
sage: J.variety(AA)
[{x: 3, z: 1, y: 2}]
```

oder auch

```
sage: V = J.variety(QQbar)
sage: [u for u in V if all(a in AA for a in u.values())]
[{z: 1, y: 2, x: 3}]
```

Eine Substitution der Forn $(x, y, z) \mapsto (\omega^a x, \omega^b y, \omega^c z)$ mit $\omega^k = 1$ lässt das System genau dann unverändert, wenn (a, b, c) Lösung modulo k des linearen homogenen Systems dieser Matrix ist:

```
sage: M = matrix([ [p.degree(v) for v in (x,y,z)]
....:              for p in J.gens()]); M
[2 1 1]
[1 3 1]
[1 1 4]
```

Nach Berechnung ihrer Determinante

```
sage: M.det()
17
```

sieht man, dass $K = 17$ passt. Wir müssen nur noch ein von null verschiedenes Element des Kerns finden:

```
sage: M.change_ring(GF(17)).right_kernel()
Vector space of degree 3 and dimension 1 over Finite Field of size 17
Basis matrix:
[1 9 6]
```

Übung 35 Seite 199. Das ist beinahe sofort:

```
sage: L.<a> = QQ[sqrt(2-sqrt(3))]; L
Number Field in a with defining polynomial x^4 - 4*x^2 + 1
sage: R.<x,y> = QQ[]
sage: J1 = (x^2 + y^2 - 1, 16*x^2*y^2 - 1)*R
```

```
sage: J1.variety(L)
[{y: 1/2*a, x: 1/2*a^3 - 2*a},
 {y: 1/2*a, x: -1/2*a^3 + 2*a},
 {y: -1/2*a, x: 1/2*a^3 - 2*a},
 {y: -1/2*a, x: -1/2*a^3 + 2*a},
 {y: 1/2*a^3 - 2*a, x: 1/2*a},
 {y: 1/2*a^3 - 2*a, x: -1/2*a},
 {y: -1/2*a^3 + 2*a, x: 1/2*a},
 {y: -1/2*a^3 + 2*a, x: -1/2*a}]
```

Somit haben wir beispielsweise für die erste Lösung oben

$$x = \frac{1}{2}(2-\sqrt{3})^{3/2} - 2\sqrt{2-\sqrt{3}}, \quad y = \frac{1}{2}\sqrt{2-\sqrt{3}}.$$

Übung 36 Seite 202. Wir haben gesehen, wie man eine Basis des $\mathbb{Q}$-Vektorraums $\mathbb{Q}[x,y]/J_2$ erhält:

```
sage: R.<x,y> = QQ[]; J2 = (x^2+y^2-1, 4*x^2*y^2-1)*R
sage: basis = J2.normal_basis(); basis
x[x*y^3, y^3, x*y^2, y^2, x*y, y, x, 1]
```

Wir berechnen dann das Bild von B unter m_x und leiten daraus die Matrix von m_x in der Basis B her:

```
sage: xbasis = [(x*p).reduce(J2) for p in basis]; xbasis
[1/4*y, x*y^3, 1/4, x*y^2, -y^3 + y, x*y, -y^2 + 1, x]
sage: mat = matrix([ [xp[q] for q in basis] for xp in xbasis])
sage: mat
[  0   0   0   0   0 1/4   0   0]
[  1   0   0   0   0   0   0   0]
[  0   0   0   0   0   0   0 1/4]
[  0   0   1   0   0   0   0   0]
[  0  -1   0   0   0   1   0   0]
[  0   0   0   0   1   0   0   0]
[  0   0   0  -1   0   0   0   1]
[  0   0   0   0   0   0   1   0]
```

Das Polynom χ_x und seine Wurzeln sind dann dann gegeben (siehe Kapitel 2 und 8) durch

```
sage: charpoly = mat.characteristic_polynomial(); charpoly
x^8 - 2*x^6 + 3/2*x^4 - 1/2*x^2 + 1/16
sage: solve(SR(charpoly), SR(x))
[x == -1/2*sqrt(2), x == 1/2*sqrt(2)]
```

An diesem Beispiel kann man sehen, dass die Wurzeln von χ die Abszissen der Punkte von $V(J_2)$ sind.

Für ein beliebiges Hauptideal J setzen wir $\chi(\lambda) = 0$ voraus mit $\lambda \in \mathbb{C}$. Daher ist λ ein Eigenwert von m_x. Sei $p \in \mathbb{Q}[x,y]\backslash J$ Repräsentant eines λ zugeordneten Eigenvektors: wir haben $xp = \lambda p + q$ für ein $q \in J$. Wegen $p \notin J$ können wir $(x_0, y_0) \in V(J)$ finden, sodass $p(x_0, y_0) \neq 0$ ist und haben dann

$$(x_0 - \lambda)p(x_0, y_0) = q(x_0, y_0) = 0,$$

woraus $\lambda = x_0$ folgt.

Übung 37 Seite 211. Die Ausdrücke sin, cos, $\sin(2\theta)$ und $\cos(2\theta)$ sind verknüpft durch die klassischen geometrischen Formeln

$$\sin^2\theta + \cos^2\theta = 1, \quad \sin(2\theta) = 2(\sin\theta)(\cos\theta), \quad \cos(2\theta) = \cos^2\theta - \sin^2\theta.$$

Zur Vereinfachung der Schreibweise setzen wir $c = \cos\theta$ und $s = \sin\theta$. Das Hauptideal

$$\langle u - (s + c), v - (2sc + c^2 - s^2), s^2 + c^2 - 1\rangle$$

von $\mathbb{Q}[s, c, u, v]$ bringt die Definitionen von $u(\theta)$ und $v(\theta)$ der Aufgabe sowie die Beziehung zwischen sin und cos zum Ausdruck. Für eine Anordnung der Monome, die vorrangig s und c entfernt, gibt die kanonische Form von s^6 modulo dieses Hauptideals das gesuchte Ergebnis.

```
sage: R.<s, c, u, v> = PolynomialRing(QQ, order='lex')
sage: Rel = ideal(u-(s+c), v-(2*s*c+c^2-s^2), s^2+c^2-1)
sage: Rel.reduce(s^6)
1/16*u^2*v^2 - 3/8*u^2*v + 7/16*u^2 + 1/8*v^2 - 1/8*v - 1/8
```

A.10. Differentialgleichungen und rekursiv definierte Folgen

Übung 38 Seite 222. *(Differentialgleichungen mit trennbaren Variablen)*

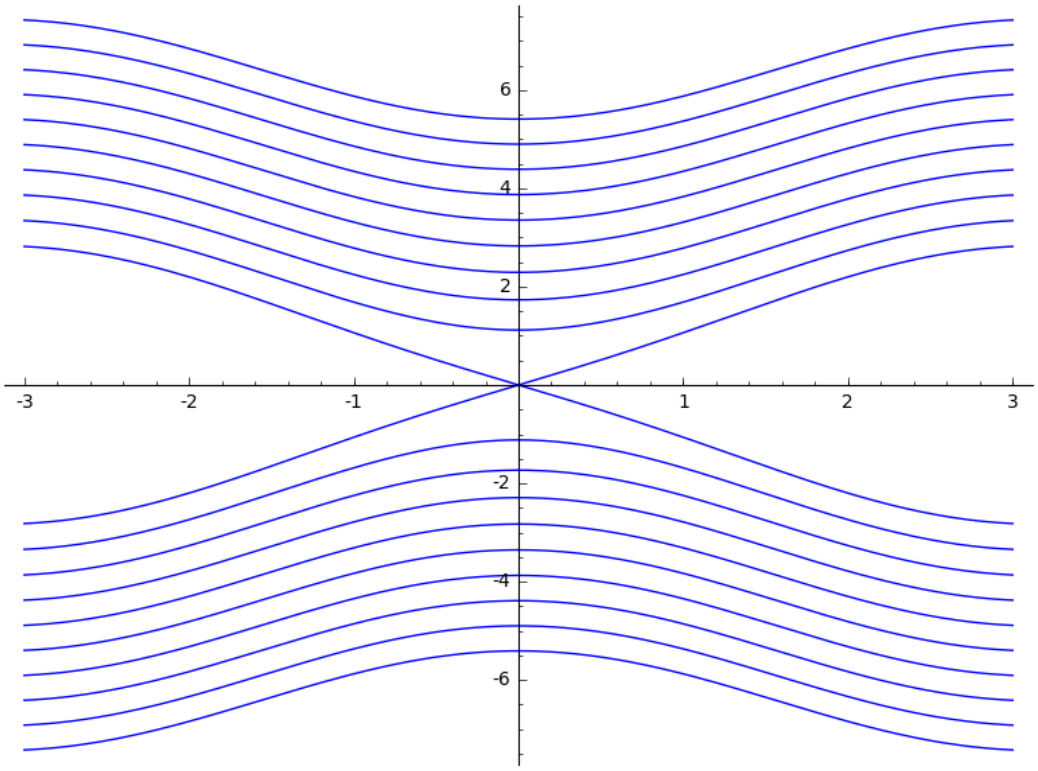
1. Wir wenden dieselbe Methode an wie in Unterabschnitt 10.2.1:

   ```
   sage: x = var('x')
   sage: y = function('y')(x)
   sage: ed = (desolve(y*diff(y,x)/sqrt(1+y^2) == sin(x),y)); ed
   sqrt(y(x)^2 + 1) == _C - cos(x)
   ```

   Da taucht das gleiche Problem auf: Wir legen fest, dass `_C-cos(x)` positiv ist:

   ```
   sage: _C = ed.variables()[0]
   sage: assume(_C-cos(x) > 0)
   sage: sol = solve(ed,y); sol
   [y(x) == -sqrt(_C^2 - 2*_C*cos(x) + cos(x)^2 - 1),
    y(x) == sqrt(_C^2 - 2*_C*cos(x) + cos(x)^2 - 1)]

   sage: P = Graphics()
   sage: for j in [0,1]:
   ....:     for k in range(0,20,2):
       :         P += plot(sol[j].subs(_C == 2+0.25*k).rhs(), x,-3,3)
   sage: P
   ```



2. Dasselbe Verfahren:

```
sage: solu = desolve(diff(y,x) == sin(x)/cos(y),y, show_method = True)
sage: solu
[sin(y(x)) == _C - cos(x), 'separable']
sage: solve(solu[0],y)
[y(x) == arcsin(_C - cos(x))]
```

Übung 39 Seite 223. *(Homogene Gleichungen)* Wir verifizieren, dasss die auf $]0,+\infty[$ und auf $]-\infty,0[$ definierte Gleichung $xyy' = x^2 + y^2$ bestimmt homogen ist. Dann versuchen wir, sie durch Veränderung einer unbekannten Funktion zu lösen, wie das im Unterabschnitt 10.1.2 im Beispiel gezeigtworden ist.

```
sage: x = var('x')
sage: y = function('y')(x)
sage: id(x) = x
sage: u = function('u')(x)
sage: d = diff(u*id,x)
sage: DE = (x*y*d == x**2+y**2).subs(y == u*id)
sage: equ = desolve(DE,u)
sage: solu = solve(equ,u)
sage: solu
[u(x) == -sqrt(2*_C + 2*log(x)), u(x) == sqrt(2*_C + 2*log(x))]
sage: Y = [x*solu[0].rhs() , x*solu[1].rhs()]
sage: Y[0]
-sqrt(2*_C + 2*log(x))*x
```

Wir könen für x Bedingungen hinzufügen (mit `assume`), um in Erinnerung zu rufen, dass die Gleichung für $x = 0$ nicht definiert ist.

A.11. Fließpunktzahlen

Übung 40 Seite 241. Wir schlagen zwei Lösungen vor.

1. Wir führen die Rechnung ohne die Methoden der Klasse `RealField` aus, welche Mantiise und Exponent einer Zahl ausgeben. Wir verifizieren zunächst $2^{99} < 10^{30} < 2^{100}$ ($10^{30} = (10^3)^{10} \approx (2^{10})^{10}$).

```
sage: R100=RealField(100)
sage: x=R100(10^30)
sage: x>2^99
True
sage: x<2^100
True
```

   Wir berechnen dann die Mantisse von x:

```
sage: e = 2^100
sage: s1 = 10^30
sage: mantisse = []
sage: anzziff = 0 # Zähler der notwendiger Ziffern
sage: while s1 > 0:
....:     e /= 2
....:     if e <= s1:
....:         mantisse.append(1)
....:         s1 -= e
....:     else:
....:         mantisse.append(0)
....:     anzziff += 1
sageprint mantisse
[1, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 0,
 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0,
 0, 1, 1, 0, 0, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1, 0, 1,
 1, 1, 1, 0, 1, 0, 1, 0, 0, 1]
sage: print "Anzahl der notwendigen Ziffern ", anzziff
Anzahl der notwendigen Ziffern  70
```

   Die binären Ziffern der Mantisse sind oberhalb der siebzigsten gleich null. Die zu 10^{30} nächstgelegene Zahl bekommt man daher durch Addition von 2^{-100} zur Mantisse, woraus sich als Resultat für $x.ulp()$ der Wert $2^{-100} \cdot 2^{100} = 1$ ergibt.

2. Mit der Methode `sign_mantissa_exponent()` der Klasse `RealField` erhalten wir unmittelbar

```
sage: R100=RealField(100)
sage: x=R100(10^30)
sage: s,m,e = x.sign_mantissa_exponent()
```

```
sage: print s,m,e
1 10000000000000000000000000000000 0
```

Der Befehl `m.binary()` erlaubt die Feststellung, dass wir in beiden Fällen die gleiche Mantisse erhalten.

Übung 41 Seite 244.

1. Wir suchen zunächst die Werte von α, β und γ in der Gleichung

$$u_n = \frac{\alpha 100^{n+1} + \beta 6^{n+1} + \gamma 5^{n+1}}{\alpha 100^n + \beta 6^n + \gamma 5^n}. \tag{A.1}$$

   Warum machen wir das nicht mit Sage? Wir bedienen uns der Werte u_0, u_1 und u_2, um ein Gleichungssystem zu bekommen und zu lösen, dessen Unbekannte α, β und γ sind. Wir formulieren die rekursiv definierte Folge und definieren die allgemeine Lösung:

```
sage: var("u0 u1 u2 alpha beta gamma n")
(u0, u1, u2, alpha, beta, gamma, n)
sage: recurrence = lambda a,b: 111-1130/a+3000/(a*b)
sage: gener1 = lambda n :(alpha*100^n+beta*6^n+gamma*5^n)
sage: solGen = lambda n: gener1(n+1)/gener1(n)
```

   Wir berechnen u_2 als Funktion von u_1 und u_0 und stellen das System auf:

```
sage: u2 = recurrence(u1,u0)
sage: s = [u2==solGen(2),u1==solGen(1),u0==solGen(0)]
sage: t = [s[i].substitute(u0=2,u1=-4) for i in range(0,3)]
```

   Dann lösen wir es:

```
sage: solve(t,alpha,beta,gamma)
[[alpha == 0, beta == -3/4*r1, gamma == r1]]
```

   Die Rechnung zeigt uns, dass γ unbestimmt bleibt.

   Wir müssen noch verifizieren, dass wir tatsächlich eine allgemeine Lösung erhalten, d.h. dass die Gleichung (A.1) für jedes n erfüllt ist.

```
sage: alpha=0
sage: beta = -3/4*gamma
sage: final = solGen(n) - recurrence(solGen(n-1), solGen(n-2))
sage: print final.simplify_full()
0
```

   Für γ können wir einen beliebigen Wert einsetzen, z.B. $\gamma = 4$, dann haben wir $\beta = -3$ und $\alpha = 0$.

2. Wir definieren eine Funktion für die rekursiv definierte Folge mit exakten Koeffizienten, damit wir sie mit unterschiedlichen Genauigkeiten verwenden können:

```
sage: def recur(x1,x0):
....:     return 111 - 1130/x1 + 3000/(x0*x1)
```

   Die Anfangsbedingungen werden hingegen aus `RealField()` genommen, sodass die Rechnung auf dieser Definitionsmenge erfolgt.

```
sage: u0 = 2.
sage: u1 = -4.
sage: for i in range(1,25):
....:     x = recur(u1,u0)
....:     print i,x
....:     u0 = u1
....:     u1 = x 1 18.5000000000000
2 9.37837837837838
3 7.80115273775217
4 7.15441448097533
5 6.80678473692481
6 6.59263276872179
..................
23 99.9999986592167
24 99.9999999193218
```

Offensichtlich konvergiert die Folge gegen 100.

3. Für die Erklärung des Verhaltens ist wichtig, dass die Werte u_n nicht exakt berechnet werden: wenn die Folge mit Anfangsbedingungen u_{n-1} und u_{n-2} definiert würde, die mit einem Rundungsfehler behaftet wären, dann lieferte die Gleichung keine allgemeine Lösung mehr.

   Suchen wir nun die stationären Werte der Folge:

   ```
   sage: var("x")
   x
   sage: solve(x==recurrence(x,x),x)
   [x == 100, x == 5, x == 6]
   ```

   Es gibt drei stationäre Werte: 100, 5 und 6. Die Konvergenz gegen 100, die bei Anwesenheit von Rundungsfehlern beobachtet wird, veranlasst Überlegungen zur Stabilität dieser drei Werte (Überlegungen, die jenseits der Grenzen dieser Übung angesiedelt sind).

4. Die Erhöhung der Genauigkeit bringt nichts, die Folge konvergiert immer gegen 100:

   ```
   sage: RL = RealField(5000)
   sage: u0 = RL(2)
   sage: u1 = RL(-4)
   sage: for i in range(1,2500):
   ....:     x = recur(u1,u0)
   ....:     u0 = u1
   ....:     u1= x
   sage: x
   100.000000000000000000000000000000000000000000000000000000000...
   ```

   Es genügt, dass ein einziges u_i nicht exakt berechnet wird, damit die Folge divergiert (α ist nicht mehr null).

5. Das Programm erfährt nur geringfügige Änderungen. Wir lassen die Variablen u_0 und u_1 ganzzahlig initialisieren:

```
sage: u0 = 2
sage: u1 = -4
sage: for i in range(1,2500):
....:     x = recur(u1,u0)
....:     u0 = u1
....:     u1 = x
sage: float(x)
6.0
```

Wir finden wirklich den Wert 6.0, doch wenn wir die rationale Zahl x ausgeben, sehen wir die enorme Menge an Information, die für die Berechnung benötigt wird (die Ausgabe wird hier nicht wiedergegeben!). Bei Ausgabe von $x-6$ erkennt man, dass der Grenzwert nicht verändert worden ist: die Größe der rechnerinternen Darstellung von x wird nicht kleiner, wenn wir die Iterationen fortsetzen.

A.12. Nichtlineare Gleichungen

Übung 42 Seite 263. Wir haben gesehen, dass das Schlüsselwort `return` die Ausführung der Funktion beendet. Es reicht deshalb aus zu testen, ob $f(0)$ null ist oder nicht. Um die mehrmalige Auswertung der Funktion f an der Stelle u zu vermeiden, speichern wir ihren Wert in einer Variablen. Mit dieser Änderung bekommen wir die folgende Funktion:

```
sage: def intervalgen(f, phi, s, t):
....:     assert (f(s) * f(t) < 0), \
....:            'Wrong arguments: f(%s) * f(%s) >= 0)'%(s, t)
....:     yield s
....:     yield t
....:     while 1:
....:         u = phi(s, t)
....:         yield u
....:         fu = f(u)
....:         if fu == 0:
....:             return
....:         if fu * f(s) < 0:
....:             t = u
....:         else:
....:             s = u
```

Testen wir diese Funktion mit einer Gleichung, deren Lösung wir kennen, beispielsweise mit einer linearen Funktion.

```
sage: f(x) = 4 * x - 1
sage: a, b = 0, 1
sage: phi(s, t) = (s + t) / 2
sage: list(intervalgen(f, phi, a, b))
[0, 1, 1/2, 1/4]
```

Übung 43 Seite 263. Die `intervalgen` als Parameter übergebene Funktion `phi` bestimmt die Stelle, an der ein Intervall geteilt werden soll. Wir müssen dieser Funktion also nur die passende Definition übergeben.

```
sage: f(x) = 4 * sin(x) - exp(x) / 2 + 1
sage: a, b = RR(-pi), RR(pi)
sage: def phi(s, t): return RR.random_element(s, t)
sage: random = intervalgen(f, phi, a, b)
sage: iterate(random, maxit=10000)
After 22 iterations: 2.15844376542606
```

Übung 44 Seite 271. Es bietet sich an, die Rechnungen mit `PolynomialRing(SR, 'x')` auszuführen. Es gibt jedoch eine technische Schwierigkeit: dieses Objekt nicht über die Methode `roors()`.

```
sage: basering.<x> = PolynomialRing(SR, 'x')
sage: p = x^2 + x
sage: p.parent()
Univariate Polynomial Ring in x over Symbolic Ring
sage: p.roots(multiplicities=False)
Traceback (most recent call last):
...
NotImplementedError
```

Achtung: Hier wird noch eine Änderung erfolgen, denn inzwischen ist Sage in der Lage, ein Ergebnis zu liefern: $[-1, 0]$.

Die Funktion `solve` ist nicht mehr eine Zusammenarbeit mit den Objekt `PolynomialRing(SR, 'x')`. Eine Alternative besteht in der Verwendung von `SR`, das diese Methode implementiert, doch bietet sie kein Äquivalent für die Methode `lagrange_polynomial()` von `PolynomialRing(SR, 'x')`. Daher werden wir zwischen diesen Objekten hin und her wechseln.

```
sage: from collections import deque
sage: basering = PolynomialRing(SR, 'x')
sage: q, method = None, None
sage: def quadraticgen(f, r, s):
....:     global q, method
....:     t = r - f(r) / f.derivative()(r)
....:     method = 'newton'
....:     yield t
....:     pts = deque([(p, f(p)) for p in (r, s, t)], maxlen=3)
....:     while True:
....:         q = basering.lagrange_polynomial(pts)
....:         p = sum([c*x^d for d, c in enumerate(q.list())])
....:         roots = [r for r in p.roots(x,multiplicities=False) \
....:                          if CC(r).is_real()]
....:         approx = None
....:         for root in roots:
....:             if (root - pts[2][0]) * (root - pts[1][0]) < 0:
....:                 approx = root
....:                 break
....:             elif (root - pts[0][0]) * (root - pts[1][0]) < 0:
....:                 pts.pop()
....:                 approx = root
....:                 break
....:         if approx:
```

```
....:             method = 'quadratic'
....:         else:
....:             method = 'dichotomy'
....:             approx = (pts[1][0] + pts[2][0]) / 2
....:         pts.append((approx, f(approx)))
....:         yield pts[2][0]
```

Jetzt ist es möglich, die ersten Terme der mit dem Verfahren von Brent rekursiv definierten Folge auszugeben. Aber Achtung: Die Rechnung ist ziemlich langwierig (und die Ausgabe des Ergebnisses passt nicht auf eine Seite dieses Buches...).

```
sage: basering = PolynomialRing(SR, 'x')
sage: a, b = pi/2, pi
sage: f(x) = 4 * sin(x) - exp(x) / 2 + 1
sage: generator = quadraticgen(f, a, b)
sage: print generator.next()
1/2*pi - (e^(1/2*pi) - 10)*e^(-1/2*pi)
```

Wärend der Ausführung des folgenden Codes wird der geduldige Leser die Parabelbögen sehen können, die bei der Berechnung der ersten Terme der Folge verwendet werden.

```
sage: generator = quadraticgen(f, a, b)
sage: g = plot(f, a, b, rgbcolor='blue')
sage: g += point((a, 0), rgbcolor='red', legend_label='0')
sage: g += point((b, 0), rgbcolor='red', legend_label='1')
sage: data = {'2': 'blue', '3': 'violet', '4': 'green'}
sage: for l, color in data.iteritems():
....:     u = RR(generator.next())
....:     print u, method
....:     g += point((u, 0), rgbcolor=color, legend_label=l)
....:     if method == 'quadratic':
....:         q = sum([c*x^d for d, c in enumerate(q.list())])
....:         g += plot(q, 0, 5, rgbcolor=color)
2.64959209030252 newton
2.17792417785930 quadratic
2.15915701651408 quadratic
sage: g.show()
```

A.13. Numerische lineare Algebra

Übung 45 Seite 279. Mit der Gleichung von Sherman und Morrison ist die Lösung von $Bx = f$ äquivalent zu derjenigen von $Ax = \sigma(I + u^t v A^{-1})f$ mit $\sigma = (1 + t^v A^{-1} u)^{-1}$. Wir gehen dann folgendermaßen vor.

1. Wir berechnen w als Lösung von $Aw = u$, dann $\sigma = (1 +^t vw)^{-1}$.
2. Wir berechnen z als Lösung von $Az = f$, dann $g =^t vz$ (das ist ein Skalar).
3. Wir berechnen dann $h = \sigma(f - gu)$ und lösen $Ax = h$; x ist die Lösung von $Bx = f$.

Wir stellen fest, dass wir drei lineare Systeme mit der zerlegten Matrix A gelöst haben, sprich die Lösung von sechs linearen Systemen mit Dreiecksmatrix. Jede dieser Lösungen erfordert Operationen der Ordnung n^2, viel weniger als der Aufwand für eine Zerlegung (die von der Ordnung n^3 ist). Um die Gleichung von Sherman und Morrison zu verifizieren, müssen wir nur das zweite Glied der Gleichung mit $A + u^t v$ (von rechts) multiplizieren und dann verifizieren, dass dieser Ausdruck gleich der identischen Matrix ist.

Übung 46 Seite 283. Wir betrachten die Cholesky-Zerlegung von $A = C^t C$, dann die Singulärwertzerlegung von C: $C = U\Sigma^t V$. Dann ist $X = U\Sigma^t U$. In der Tat: $A = C^t C = (U\Sigma^t V)(V\Sigma^t U) = USigma^t UUSigma^t U = X^2$.

Erstellen wir eine symmetrische positiv definite Zufallsmatrix:

```
sage: m = random_matrix(RDF,4)
sage: a = transpose(m)*m
sage: c = a.cholesky()
sage: U,S,V = c.SVD()
sage: X = U*S*transpose(U)
```

Verifizieren wir, dass $X^2 - a$ null ist (abgesehen von numerischen Fehlern):

```
sage: M = (X*X-a)
sage: all(abs(M[i,j]) < 10^-14
....:     for i in range(4) for j in range(4))
True
```

A.14. Numerische Integration und Differentialgleichungen

Übung 47 Seite 308. *(Berechnung der Koeffizienten von Newton-Cotes)*

1. Wenn wir bemerken, dass P_i $n-1$-ten Grades ist (worauf Gleichung (14.1) angewendet wird), und dass $P_i(j) = 0$ für $j \in \{0, \ldots, n-1\}$ und $j \neq i$, leiten wir daraus her:

$$\int_0^{n-1} P_i(x)\mathrm{d}x = w_i P_i(i)$$

   also

$$w_i = \frac{\int_0^{n-1} P_i(x)\mathrm{d}x}{P_i(i)}.$$

2. Einfach folgt daraus die Berechnung der Gewichte:

```
sage: x = var('x')
sage: def NCRule(n):
....:     P = prod([x - j for j in xrange(n)])
....:     return [integrate(P / (x-i), x, 0, n-1) \
....:             / (P/(x-i)).subs(x=i) for i in xrange(n)]
```

3. Durch einen simplen Variablentausch:

$$\int_a^b f(x)\mathrm{d}x = \frac{b-a}{n-1} \int_0^{n-1} f\left(a + \frac{b-a}{n-1}\right) \mathrm{d}u.$$

4. Wir wenden vorstehende Gleichung an und finden wir das folgende Programm:

```
sage: def QuadNC(f, a, b, n):
....:     W = NCRule(n)
....:     ret = 0
....:     for i in xrange(n):
....:         ret += f(a + (b-a)/(n-1)*i) * W[i]
....:     return (b-a)/(n-1)*ret
```

Bevor wir die Genauigkeit dieses Verfahrens mit der von anderen vergleichen, können wir bereits verifizieren, dass es keine inkohärenten Ergebnisse liefert:

```
sage: QuadNC(lambda u: 1, 0, 1, 12)
1
sage: N(QuadNC(sin, 0, pi, 10))
1.99999989482634
```

Nun vergleichen wir das erhaltene Verfahren kurz mit den Funktionen von GSL anhand der Integrale I_2 und I_3:

```
sage: numerical_integral(x * log(1+x), 0, 1)
(0.25, 2.7755575615628914e-15)
sage: N(QuadNC(lambda x: x * log(1+x), 0, 1, 19))
0.250000000000001
sage: numerical_integral(sqrt(1-x^2), 0, 1)
(0.7853981677264822, 9.042725224567119e-07)
sage: N(pi/4)
0.785398163397448
sage: N(QuadNC(lambda x: sqrt(1-x^2), 0, 1, 20))
0.784586419900198
```

Wir bemerken, dass die Güte des Ergebnisses von der Anzahl der verwendeten Punkte abhängt:

```
sage: [N(QuadNC(lambda x: x * log(1+x), 0, 1, n) - 1/4)
....:  for n in [2, 8, 16]]
[0.0965735902799726, 1.17408932921378e-7, 2.13476805677182e-13]
sage: [N(QuadNC(lambda x: sqrt(1-x^2), 0, 1, n) - pi/4)
....: for n in [2, 8, 16]]
[-0.285398163397448, -0.00524656673640445, -0.00125482109302663]
```

Ein interessanterer Vergleich zwischen den verschiedenen Integrationsfunktionen von Sage mit unserem Verfahren `QuadNC` würde die Umwandlung in ein adaptives Verfahren verlangen, welches das betrachtete Intervall wie `numerical_integral` automatisch unterteilt.

A.15. Aufzählende Kombinatorik

Übung 48 Seite 319. *(Wahrscheinlichkeit, beim Poker einen Vierling zu ziehen)* Wir bilden die Menge der Vierlinge:

```
sage: Farben = FiniteEnumeratedSet(["Karo","Herz","Pik","Kreuz"])
sage: Werte = FiniteEnumeratedSet([2, 3, 4, 5, 6, 7, 8, 9, 10,
                                   "Bube", "Dame", "Koenig", "Ass"])
sage: Vierlinge = cartesian_product([(Arrangements(Werte,2)), Farben])
```

Wir haben `FiniteEnumeratedSet` benutzt statt `Set`, um die Reihenfolge der Farben, der Werte und damit auch der Vierlinge festzulegen.

```
sage: Vierlinge.list()
[[[2, 3], 'Karo'],
[[2, 3], 'Herz'],
...
[['Ass', 'Koenig'], 'Kreuz']]
```

Diese Liste beginnt mit einem Zweier-Vierling und der Karo 3 und endet mit einem Vierling von Assen und dem Kreuz König. Insgesamt gibt es 624 Vierlinge.

```
sage: Carres.cardinality()
624
```

Bezogen auf die Anzahl möglicher Hände haben wir eine Chance von 1:4165, einen Vierling zu bekommen wenn ein Blatt zufällig gezogen wird.

```
sage: Karten = cartesian_product([Farben, Werte]).map(tuple)
sage: Haende = Subsets(Karten, 5)
sage: Vierlinge.cardinality() / Haende.cardinality()
1/4165
```

Übung 49 Seite 319. *(Wahrscheinlichkeit, beim Poker einen Straight Flush oder einen Flush zu ziehen)* Einen Straight Flush einzuordnen läuft zum Beispiel darauf hinaus, seine kleinste Karte zu bestimmen (zwischen 1 und 10) und seine Farbe. Davon gibt es demnach 40.

```
sage: StraightFlush = cartesian_product([srange(1, 11), Farben])
sage: StraightFlush.cardinality()
40
```

Und es gibt 5108 Flushes:

```
sage: AlleFlushes = cartesian_product([Subsets(Werte,5),Farben])
sage: AlleFlushes.cardinality() - StraightFlush.cardinality()
5108
```

Schließlich ist die Wahrscheinlichkeit, zufällig einen Flush zu ziehen, etwa zwei Promille:

```
sage: _ / Mains.cardinality()
1277/649740
sage: float(_)
0.0019654015452334776
```

Es wäre schön, die vorstehende Rechnung mit Mengenoperationen durchzuführen, indem die Menge der Flushes explizit als Differenz von `alleFlushes` und `StraightFlush` gebildet wird. Indes gibt es keinen effizienten generischen Algorithmus zur Berechnung der Differenz zweier Mengen $A\backslash B$: ohne zusätzliche Information gibt es kaum etwas Besseres als alle Elemente von A zu prüfen, ob sie in B enthalten sind. In obiger Rechnung haben wir die Tatsache benutzt, das B in A enthalten ist, was Sage vorher nicht wissen kann. Ein weiteres, leicht zu umgehendes Hindernis ist, dass die Elemente von B und A auf die gleiche Weise repräsentiert werden müssen.

Übung 50 Seite 319. Wir beschränken uns darauf den Fall eines Full House zu illustrieren, das aus einem Drilling und einem Paar besteht. Beginnen wir damit, eine Funktion zu schtreiben, die prüft, ob eine Hand ein Full House ist. Eine knappe Schreibweise erzielen wir mit dieser Methode:

```
sage: Word(['a','b','b','a','a','b','a']).evaluation_dict()
{'a': 4, 'b': 3}
```

```
sage: def is_Full_House(Hand):
....:     Farben = Word([Wert for (Farbe, Werte) in Hand])
....:     Wiederholungen = sorted(Farben.evaluation_dict().values())
....:     return Wiederholungen == [2,3]
sage: is_Full_House({('Karo', 5), ('Karo', 6), ('Herz', 6),
....:                ('Pique', 5), ('Pique', 1)})
False
sage: is_Full_House({('Kreuz', 3), ('Pik', 3), ('Herz', 3),
....:               ('Kreuz', 2), ('Pik', 2)})
True
```

Wir automatisieren jetzt die Abschätzung des Anteils von Full Houses. Allgemeiner, die folgende Funktion schätzt den Anteile jedes Elements der endlichen Menge `menge`, das einem `praedikat` genügt. Sie geht davon aus, dass die Menge mit einer Methode `random_element` versehen ist, die gleichverteiltes Ziehen implementiert:

```
sage: def schaetzung_anteil(menge, praedikat, n):
....:     zaehler = 0
....:     for i in range(n):
....:         if praedikat(menge.random_element()):
....:             zaehler += 1
....:     return zaehler/n
sage: float(schaetzung_anteil(Haende, is_Full_House, 10000))
0.0009
```

Kommen wir jetzt zur symbolischen Rechnung. Ein Full House zu bestimmen heißt, zwei verschiedene Farben anzugeben, die eine für den Drilling, die andere für das Paar, sowie eine Menge von drei Werten für den Drilling und eine andere mit zwei Werten für das Paar:

```
sage: Full_Houses = cartesian_product([Arrangements(Valeurs, 2),
....:     Subsets(Symboles, 3), Subsets(Symboles, 2)])
```

Hier beispielsweise ein Full House mit einem Drilling von Zweien und einem Paar von Dreien:

```
sage: Full_Houses.first()
[[2, 3], {'Kreuz', 'Pik', 'Karo'}, {'Pik', 'Kreuz'}]
```

Die Wahrscheinlichkeit für ein Full House ist dann:

```
sage: float(MainsPleines.cardinality() / Mains.cardinality())
0.0014405762304921968
```

Übung 51 Seite 320. *(Berechnung der vollständigen Binärbäume von Hand)* Es gibt einen vollständigen Binärbaum mit einem Blatt und einen mit zwei Blättern. Für $n = 3, 4$ und 5 Blätter findet man 2, 5 bzw. 14 Bäume (wegen $n = 4$ siehe Abbildung 15.1)

Übung 52 Seite 329. Die Kompositionen von n über k Teilen sind in Bijektion mit den Teilmengen der Größe k von $\{1, \ldots, n\}$: der Menge $\{i_1, \ldots, i_k\}$ mit $i_1 < i_2 < \ldots < i_k$ ordnen wir die Komposition $(i_1, i_2 - i_1, \ldots, n.i_k)$ zu und umgekehrt. Daraus leiten sich die Formeln für das Abzählen ab: es gibt 2^n Kompositionen von n. Davon sind *binomnk* Kompositionen mit k Teilen. Um herauszufinden, ob diese Gleichungen verwendet werden, können wir uns die Implementierung der Methode `cardinality` anschauen

```
sage: C = Compositions(5)
sage: C.cardinality??
```

Im zweiten Fall liefert schon der Name der intern verwendeten Methode, `_cardinality_from_iterator`, die Information: die Kardinalität wird - ineffizient - durch Iteration über alle Kompositionen berechnet.

```
sage: C = Compositions(5,length=3)
sage: C.cardinality
<bound method IntegerListsLex...._cardinality_from_iterator ...>
```

Übung 53 Seite 332. Einige Beispiele:

```
sage: IntegerVectors(5,3).list()
[[5, 0, 0], [4, 1, 0], [4, 0, 1], [3, 2, 0], [3, 1, 1], [3, 0, 2],
 ...
[0, 4, 1], [0, 3, 2], [0, 2, 3], [0, 1, 4], [0, 0, 5]]
```

```
sage: OrderedSetPartitions(3).cardinality()
13
sage: OrderedSetPartitions(3).list()
[[{1}, {2}, {3}], [{1}, {3}, {2}], [{2}, {1}, {3}], [{3}, {1}, {2}],
 ...
[{1, 2}, {3}], [{1, 3}, {2}], [{2, 3}, {1}], [{1, 2, 3}]]
sage: OrderedSetPartitions(3,2).random_element()
[{1, 3}, {2}]
```

```
sage: StandardTableaux([3,2]).cardinality()
5
sage: StandardTableaux([3,2]).an_element()
[[1, 3, 5], [2, 4]]
```

Übung 54 Seite 332. Im Kleinen bekommen wir diese Permutationsmatrizen:

```
sage: list(AlternatingSignMatrices(1))
[[1]]
sage: list(AlternatingSignMatrices(2))
[
[1 0]  [0 1]
[0 1], [1 0]
]
```

Das erste neue Element erscheint bei $n = 3$:

```
sage: list(AlternatingSignMatrices(3))
[
[1 0 0]  [0 1 0]  [1 0 0]  [ 0  1 0]  [0 0 1]  [0 1 0]  [0 0 1]
[0 1 0]  [1 0 0]  [0 0 1]  [ 1 -1 1]  [1 0 0]  [0 0 1]  [0 1 0]
[0 0 1], [0 0 1], [0 1 0], [ 0  1 0], [0 1 0], [1 0 0], [1 0 0]
]
```

Bei Betrachtung der Beispiele mit größerem n können wir sehen, dass es sich immer um Matrizen handelt mit Koeffizienten aus $\{-1, 0, 1\}$, sodass in jeder Zeile und jeder Spalte die von null verschiedenen Koeffizienten zwischen -1 und 1 alternieren, wobei sie mit 1 beginnen und mit 1 enden.

Übung 55 Seite 332. Es gibt 2^5 Vektoren in $(\mathbb{Z}/2\mathbb{Z})^5$:

```
sage: GF(2)^5
Vector space of dimension 5 over Finite Field of size 2
sage: _.cardinality()
32
```

Um eine invertierbare 3×3-Matrix mit Koeffizienten aus $\mathbb{Z}/2\mathbb{Z}$ zu bilden, reicht es hin, einen ersten von null verschiedenen Zeilenvektor (aus $2^3 - 1$) auszuwählen, dann einen zweiten Vektor (aus $2^3 - 2$), der nicht in der vom ersten aufgespannten Geraden liegt, dann einen dritten Zeilenvektor (aus $2^3 - 3$), der nicht in der von den ersten beiden aufgespannten Ebene liegt. Das macht

```
sage: (2^3-2^0)*(2^3-2)*(2^3-2^2)
168
```

Und wirklich:

```
sage: GL(3,2)
General Linear Group of degree 3 over Finite Field of size 2
sage: _.cardinality()
168
```

Dieselbe Überlegung führt auf die allgemeine Formel, die mittels `q_factorial` ausgedrückt wird:

$$\prod_{k=0}^{n-1}(q^n - q^k) = \frac{q^n}{(q-1)^n}[n]_q!$$

Also:

```
sage: from sage.combinat.q_analogues import q_factorial
sage: q = 2; n = 3
```

```
sage: q^n/(q-1)^n *q_factorial(n,q)
168
```

Übung 56 Seite 335. Im ersten Fall beginnt Python mit der Konstruktion der Liste aller Ergebnisse, bevor sie an `all` übergen wird. Im zweiten Fall übergibt der Iterator die Ergebnisse an `all` nach und nach; letztere kann deshalb terminieren, sobald ein Gegenbeispiel gefunden wird.

Übung 57 Seite 336. Die erste Zeile ergibt die Liste aller Kuben der ganzen Zahlen zwischen -999 und 999 einschließlich. Die beiden folgenden suchen nach einem Paar von Kuben, deren Summe gleich 218 ist. Die letzte Zeile ist effizienter in der Zeit, denn sie terminiert, sobald eine Lösung gefunden wird:

```
sage: cubes = [t**3 for t in range(-999,1000)]
sage: %time exists([(x,y) for x in cubes for y in cubes],
....:              lambda (x,y): x+y == 218)
CPU times: user 1.28 s, sys: 0.07 s, total: 1.35 s
Wall time: 1.35 s
(True, (-125, 343))
sage: %time exists(((x,y) for x in cubes for y in cubes),
....:              lambda (x,y): x+y == 218)
CPU times: user 0.88 s, sys: 0.02 s, total: 0.90 s
Wall time: 0.86 s
(True, (-125, 343))
```

Vor allem ist sie effizienter beim Speicherbedarf: wenn n die Länge der Liste der Kuben ist, ist die Größe des benutzten Speichers von der Größenordnung n statt n^2. Das wird besonders dann spürbar, wenn wir n verzehnfachen.

Übung 58 Seite 336.

- Berechne die erzeugende Reihe $\sum_{s\subset S} x^{|s|}$ aller Teilmengen von $\{1,\ldots,8\}$ als Funktion ihrer Kardinalität.
- Berechne die erzeugende Reihe der Permutationen von $\{1,2,3\}$ als Funktion der Anzahl ihrer Inversionen.
- Verifiziere die Tautologie $\forall x \in P,\ x \in P$ für die Menge P der Permutationen von $\{1,2,3,4,5\}$. Das ist ein guter Test für den inneren Zusammenhang der Iterationsfunktionen und für die Zugehörigkeit zur selben Menge. Er ist zudem in den generischen Test von Sage enthalten; siehe:

  ```
  sage: P = Partitions(5)
  sage: P._test_enumerated_set_contains??
  ```

  Die Tautologie $\forall x \notin P,\ x \notin P$ wäre gut geeignet. den Test auf Zugehörigkeit zu komplettieren. Allerdings müsste die Grundmenge präzisiert werden; und vor allem bräuchte es einen Iterator auf dem Komplement von P in dieser Grundmenge, was keine gewöhnliche Operation ist.
- Gib alle auf $\mathbb{Z}/2\mathbb{Z}$ invertierbaren 2×2-Matrizen aus.
- Gib alle Partitionen von 3 aus.
- Gib alle Partitionen der ganzen Zahlen (terminiert nicht!).

- Gib alle Primzahlen aus (terminiert nicht!).

- Suche eine Primzahl, deren zugehörige Mersenne-Zahl nicht prim ist.

- Iteriere über alle Primzahlen, deren zugehörige Mersenne-Zahl nicht prim ist.

Übung 59 Seite 338. Bilden wir `Leaf` und `Node` wie vorgeschlagen:

```
sage: Leaf = var('Leaf'); Node = function('Node', nargs=2)
```

Dann definieren wir unseren Iterator rekursiv:

```
sage: def C(n):
....:     if n == 1:
....:         yield Leaf
....:     elif n > 1:
....:         for k in range(1,n):
....:             for t1 in C(k):
....:                 for t2 in C(n-k):
....:                     yield Node(t1, t2)
```

Hier die kleinen Bäume:

```
sage: list(C(1))
[Leaf]
sage: list(C(2))
[Node(Leaf, Leaf)]
sage: list(C(3))
[Node(Leaf, Node(Leaf, Leaf)), Node(Node(Leaf, Leaf), Leaf)]
sage: list(C(4))
[Node(Leaf, Node(Leaf, Node(Leaf, Leaf))),
Node(Leaf, Node(Node(Leaf, Leaf), Leaf)),
Node(Node(Leaf, Leaf), Node(Leaf, Leaf)),
Node(Node(Leaf, Node(Leaf, Leaf)), Leaf),
Node(Node(Node(Leaf, Leaf), Leaf), Leaf)]
```

Wir treffen wieder auf die Folge von Catalan:

```
sage: [len(list(C(n))) for n in range(9)]
[0, 1, 1, 2, 5, 14, 42, 132, 429]
```

A.16. Graphentheorie

Übung 60 Seite350. *(Zirkulierende Graphen).* Zwei Schleifen reichen aus!

```
sage: def circulant(n, d):
....:     g = Graph(n)
....:     for u in range(n):
....:         for c in range(d):
....:             g.add_edge(u, (u+c)%n)
....:     return g
```

Übung 61 Seite 352. *(Graph von Kneser)* Das Einfachste ist, das Sage-Objekt `Subsets`zu verwenden. Wir listen dann alle Knotenpaare auf, um die adjacenten zu finden, doch das erfordert viel unnütze Rechnerei.

```
sage: def kneser(n,k):
....:     g = Graph()
....:     g.add_vertices(Subsets(n,k))
....:     for u in g:
....:         for v in g:
....:             if not u & v:
....:                 g.add_edge(u,v)
....:     return g
```

Wir können allerdings Zeit gewinnen, wenn wir nur die adjazenten Knoten auflisten.

```
sage: def kneser(n,k):
....:     g = Graph()
....:     sommets = Set(range(n))
....:     g.add_vertices(Subsets(sommets,k))
....:     for u in g:
....:         for v in Subsets(sommets - u,k):
....:             g.add_edge(u,v)
....:     return g
```

Übung 62 Seite 367. *(Optimale Reihenfolge bei gieriger Färbung)* Die Methode `coloring` liefert eine Färbung als Liste von Listen: die Liste der Knoten mit der Farbe 0, die Liste der Knoten mit der Farbe 1 usw. Um auf den Knoten eine Ordnung zu erhalten, die mit dem gierigen Algorithmus eine optimale Färbung liefert, genügt es, die Knoten mit der Farbe 0 aufzulisten (egal in welcher Reihenfolge), dan die mit der Farbe 1 uns so fort! So erhalten wir für den Petersen-Graphen:

```
sage: g = graphs.PetersenGraph()
sage: def ordre_optimal(g):
....:     ordre = []
....:     for classe_de_couleur in g.coloring():
....:         for v in classe_de_couleur:
....:             ordre.append(v)
....:     return ordre
sage: ordre_optimal(g)
[1, 3, 5, 9, 0, 2, 6, 4, 7, 8]
```

A.17. Lineare Programmierung

Übung 63 Seite 380. *(Teilsummenproblem)* Jedem Element der Menge weisen wir eine binäre Variable zu, die anzeigt, ob das Objekt zur Teilmenge mit der Summe null gehört oder nicht. Dann gibt es zwei Bedingungen:

- die Summe der zugehörigen Elemente muss null sein,
- die Menge darf nicht leer sein.

Das macht nun der folgende Code:

```
sage: l = [28, 10, -89, 69, 42, -37, 76, 78, -40, 92, -93, 45]
sage: p = MixedIntegerLinearProgram()
sage: b = p.new_variable(binary = True)
sage: p.add_constraint(p.sum([v*b[v] for v in l]) == 0)
sage: p.add_constraint(p.sum([b[v] for v in l]) >= 1)
sage: p.solve()
0.0
sage: b = p.get_values(b)
sage: print [v for v in b if b[v] == 1]
[-93, 10, 45, 78, -40]
```

Zu beachten ist, dass keine Zielfunktion definiert werden musste.

Übung 64 Seite 381. *(Dominierende Menge)* Die Bedingungen dieses ganzzahligen linearen Programms entsprechen einem Überdeckungsproblem: eine Menge S von Knoten eines Graphen ist eine dominierende Menge genau dann, wenn für jeden Knoten v gilt $(\{v\} \cup N_G(v)) \cap S \neq \emptyset$, wobei $N_G(v)$ die Menge der Nachbarn von v in G bezeichnet. Daraus leiten wir folgenden Code her:

```
sage: g = graphs.PetersenGraph()
sage: p = MixedIntegerLinearProgram(maximization = False)
sage: b = p.new_variable(binary = True)
sage: for v in g:
....:     p.add_constraint(p.sum([b[u] for u in g.neighbors(v)]) + b[v] >= 1)
sage: p.set_objective(p.sum([b[v] for v in g]))
sage: p.solve()
3.0
sage: b = p.get_values(b)
sage: print [v for v in b if b[v] == 1]
[0, 2, 6]
```

B. Bibliographie

[AL03] Journaïdi Abdeljaoued und Henri Lombardi, *Méthodes matricielles — Introduction à la complexité algébrique.* Springer, 2003, ISBN 3540202471.

[AP98] Uri M. Ascher und Linda R. Petzold, *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations.* Society for Industrial and Applied Mathematics, 1998, ISDBN 0898714128.

[AS] Noga Alon und Joel H. Spencer, *The Probalistic Method.* Wiley-Interscience, 2000, ISBN 0471370460.

[Bea] Bernard Beauzamy, *Robust Mathematival Methods for Extremely Rare Events.* Online, 2009. `http://www.scmsa.eu/RMM/BB_rare_events_2009_08.pdf`, 20 Seiten

[BZ10] Richard P. Brent und Paul Zimmermann, *Modern Computer Arithmetic.* Cambridge University Press, 2010, ISBN 0521194693. `http://www.loria.fr/~zimmerma/mca/pub226.html`.

[Cia82] Philippe G. Ciarlet, *Introduction à l'analyse numérique matricielle et à l'optimation.* Mathematiques appliquées pour la maîtrise. Masson, 1982, ISBN 2225688931.

[CLO07] David Cox, John Little und Donal O'Shea, *Ideals, Varieties, and Algorithms.* Undergraduate Texts in Mathematics. Springer-Verlag, 3. Auflage 2007, ISBN 0387946802.

[CM84] Michel Crouzeix und Alain L. Mignot, *Analyse numérique des équations différentielles.* Mathématiques appliquées pour la maîtrise. Masson 1984, ISBN 2225773416.

[Coh] Henri Cohen, *A Course in Computational Algebraic Number Theory.* Nummer 138 in *Graduate Texts in Mathematics.* Springer-Verlag, 1993, ISBN 3540556400.

[CP01] Richard Crandall und Carl Pomerance, *Prime Numbers: A Computational Perspective.* Springer-Verlag, 2001, ISBN 0387947779.

[DGSZ95] Philippe Dumas, Claude Gomez, Bruno Salvy und Paul Zimmermann, *Calcul formel : mode d'emploi.* Masson, 1995, ISBN 2225847800.

[Edm65] Jack Edmonds, *Paths, Trees, and Flowers.* Canadian Journal of Mathematics, 17(3):449-467, 1965.

[EM07] Mohamed Elkadi und Bernard Mourrain, *Introduction à la résolution des systèmes polynomiaux.* Nummmer 59 in *Mathématiques et Applications.* Springer-Verlag, 2007, ISBN 3540716467.

[FS09] Philippe Flajolet und Robert Sedgewick, *Analytic Combinatorics.* Cambridge University Press, 2009, ISBN 0521898065.

[FSED09] Jean Charles Faugère und Mohab Safey El Din, *De l'algèbre linéaire à la résolution des systèmes polynomiaux.* In Jacques Arthur Weil und Alain Yger (Herausgeber), *Mathématiques appliquées L3,* Seiten 331-388. Pearson Education, 2009, ISBN 2744073520.

[Gan90] Félix Rudimovich Gantmacher, *Théorie des Matrices.* Éditions Jacques Gabay, 1990, ISBN 2876470357.

[Gol91] David Goldberg, *What Every Computer Scientist Should Know About Floating Point Arithmetic.* ACM Computing Surveys, 23(1):5-48, 1991.

[GVL96] Gene H. Golub und Charles F. Van Loan, *Matrix Computations.* Johns Hopkins Studies in the Mathematical Sciences. Johns Hopkin University Press, 3^e édition, 1996, ISBN 0801854149.

[Hig93] Nicholas J. Higham, *The Accuracy of Floating Point Summation.* SIAM Journal on Scientific Computing, 14(4):783-799, 1993, ISSN 1064-8275.

[HLW02] Ernst Hairer, Christian Lubich und Gerhard Wanner, *Geometric Numerical Integration.* Springer-Verlag, 2002, ISBN 3662050200.

[HT04] Florent Hivert und Nicolas M. Thiéry, *MuPAD-Combinat, an Open-Source Package for Research in Algebraic Combinatorics.* Séminaire lotharingien de combiuatoire, 51:B51z, 2004. `http://www.mat.univie,ac.at/~slc/wpapers/s51thiery.html.`

[LT93] Patrick Lascaux und Raymond Théodor, *Analyse, numérique matricielle appliquée à l'art de l'ingénieur,* Band 1. Masson, 2. Auflage, 1993, ISBN 2225841225.

[LT94] Patrick Lascaux und Raymond Théodor, *Analyse, numérique matricielle appliquée à l'art de l'ingénieur,* Band 2. Masson, 2. Auflage, 1994, ISBN 2225845468.

[Mas13] Thierry Massart, *Syllabus INFO-F-101 Programmation.* Online, 2013. `http://www.ulb.ac.be/di/verif/tmassart/Prog/,` version 3.2.

[Mat03] Jiří Matoušek, *Using the Borsuk-Ulam Theorem: Lectures on Topological Methods in Combinatorics and Geometry.* Springer-Verlag, 2003, ISBN 3540003625.

[MBdD+10] Jean Michel Muller, Nicolas Brisebarre, Florent de Dinechin, Claude Pierre Jeannerod, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé et Serge Torres, *Handbook of floating-point arithmetic.* Birkhäuser, 2010, ISBN 0817647049.

[Mor05] Masatake Mori. *Discovery of the Double Exponential Transformation and Its Developments.* Publications of the Research Institute for Mathematical Sciences. 41:897-935, 2005.

[NO] Sampo Niskancn und Patric R. J. Östergård, *Cliquer — Routines for Clique Searching,* `http://users.tkk.fi/pat/cliquer.html.`

[PWZ96] Marko Petkovšek, Herbert S. Wilf und Doron Zeilberger, $A = B$. A K Peters Ltd., 1996, ISBN 1568810636.

[Sch91] Michelle Schatzman, *Analyse numérique.* InterEditions, 1991, ISBN 2729603731.

[SwiOO] Gérard Swinnen, *Apprendre à programmer avec Python.* Eyrolles, 2009, ISBN 2212124743. `http://inforef.be/swi/python.htm.`

[Swil2] Gérard Swinnen, *Apprendre à programmer avec Python 3.* Eyrolles, 2012, ISBN 2212134346. `http://inforef.be/swi/python.htm.`

[TMF00] Gerald Tenenbaum und Michel Mendès France, *Les nombres premiers.* Que sais-je? P.U.F.. 2000, ISBN 2130483992.

[TSM05] Ken'ichiro Tanaka, Masaaki Sugihara und Kazuo Murota, *Numerical Indefinite Integration by Double Exponential sinc Method.* Mathematics of Computation, 74(250):655-679, 2005.

[Vie07] Xavier Viennot, *Leonhard Euler, père de la combinatoire contemporaine.* Exposé à la journée « Leonhard Euler, mathémati. cien universel », IHÉS. Bures-sur-Yvette, mai 2007. `http://www.xavierviennot.org/xgv/exposes_files/Euler_IHES_web.pdf.`

[vzGG03] Joachim von zur Gathen und Jürgen Gerhard, *Modern Computer Algebra.* Cambridge University Press, 2. Auflage, 2003, ISBN 0521826462. `http://www-math.uni-paderborn.de/mca.`

[Zei96] Doron Zeilberger, *Proof of the Alternating Sign Matrix Conjecture.* Electronic Journal of Combinatorics, 3(2), 1996.

C. Register

Index