pyAPP7 Documentation

Release 1.1

ALR Aerospace

Aug 19, 2020

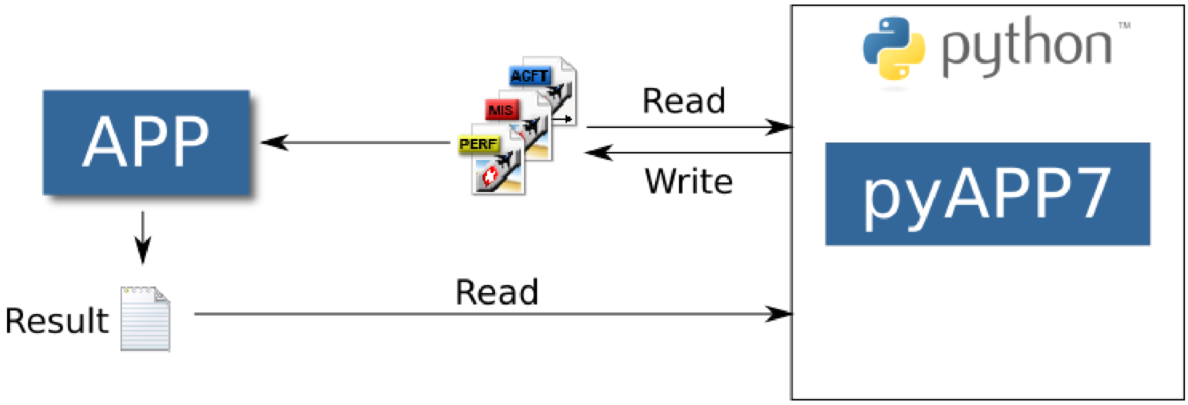
Contents

1 Introduction

pyAPP7 is a Python package to interact with ALR's Aircraft Performance Program APP.

The current implementation of pyAPP7 is a file based interface to APP. pyAPP7's classes enable to read and write APP7 files. Together with the command line interface of APP, pyAPP7 allows for automation of mission and point performance computations. The results that are calculated and written by APP (as text files) can be read with pyAPP7.

pyAPP7 is made to work with both Python 2 and Python 3. This was tested using Python 2.7 and Python 3.6.



2 Installation

If you do not already have a Python distribution installed, ALR recommends to use a distribution that is pre-built for windows and includes common modules such as numpy, scipy and matplotlib. Such a distribution is Anaconda, available here: https://www.continuum.io/downloads

Installing pyAPP7 is straight forward, as for any python packge. Navigate to the pyAPP7 folder and open a Windows command line (cmd). Install pyAPP7 by executing:

```
python setup.py install
```

This will add pyAPP7 to your active python distribution. To test the sucessful installation, open a python shell by entering:

```
python
```

and then type:

```
>>> import pyAPP7
```

If no error message appears, the installation was sucessful.

A second method is to either add the path to the pyAPP7 root folder in each script by modifying sys.path or to add the path to the PYTHONPATH environment variable. For futher information, consult the official python documentation on how to install modules: https://docs.python.org/2/install/#modifying-python-s-search-path

3 APP Command Line

APP7 offers a command line mode to execute a computation without using the Graphical User Interface (GUI). pyAPP7 has two classes that simplify the execution of APP from a Python script.

The command line mode of APP7 writes the results in ASCII format into a text file (.txt). This file can then be read by pyAPP7.

To calculate a mission saved as myMission.mis, type:

```
App7.exe -m myMission.mis
```

To calculate a Performance Chart saved as myChart.perf, type:

```
App7.exe -pp myChart.perf
```

4 Upgrading from pyAPP6

If you have existing pyAPP6 based scripts, updating to APP7 and pyAPP7 usually requires only the two following steps:

- Make sure your APP aircraft, performance and mission files were saved with the latest APP7 GUI version.
- Replace all occurrences of pyAPP6 in the import statements (“from pyAPP6 import ...” or “import pyAPP6”) with pyAPP7.

With these two changes, most code should be backwards compatible. However, in order to improve the code readability, quality and usability of pyAPP, some breaking changes were necessary.

The following list shows some of the issues that might be encountered when porting your code from pyAPP6 to pyAPP7:

- The unit conversion variables in the module “Units.py” have been renamed (the underscore prefix from pyAPP6 was removed in pyAPP7).
- The APP custom data type classes have been moved from “Files.py” to a new module “Datatypes.py”.
- The name of the variable for the path to the APP execuatble was renamed from APP6Dir to APP7Dir (in the Mission and Performance module).
- All functions related to file parsing in “Files.py” and “Global.py” have been moved to the module “FileFunctions.py”.
- Variable indices for performance and mission results have been changed as more variables have been added to the APP parameter list from APP6 to APP7. The database class automatically handles this, but any script with indices hard coded in will need to be updated during a version upgrade from pyAPP6 to pyAPP7.
- Propeller data is stored within the PropThrust class in pyAPP6 while pyAPP7 has a unique propeller class which is a member of the PropThrust and PropElectricThrust classes.
- In the class MisOptData, the variable “optimizer” has been renamed to “solver”. The old function names and dictionary keys to access the data have been kept for backwards compatibility.
- Functions to access the mass have been moved from the class “Mass” to “Config”.

5 Authors

ALR-Aerospace:

- Marc Immer
- Micha Brunner
- Philipp Juretzko
- Vito Colangelo

6 User Guide

This user guide to pyAPP7 is structured into four parts. First, an overview over the *package strcuture* is provided. The second section describes how to *read and write APP files* (aircraft, missions and performance charts) using Python. The last two sections describe how to execute APP's *mission computations* and *performance chart computations* by using Python and parse the results written by APP.

6.1 Package Structure

The pyAPP7 package comprises the following modules:

Files Classes for *Reading and Writing APP Files*.

Datatypes Classes defining APP sepcific, custom data types

Mission Classes for executing *Mission Computations* and reading the results.

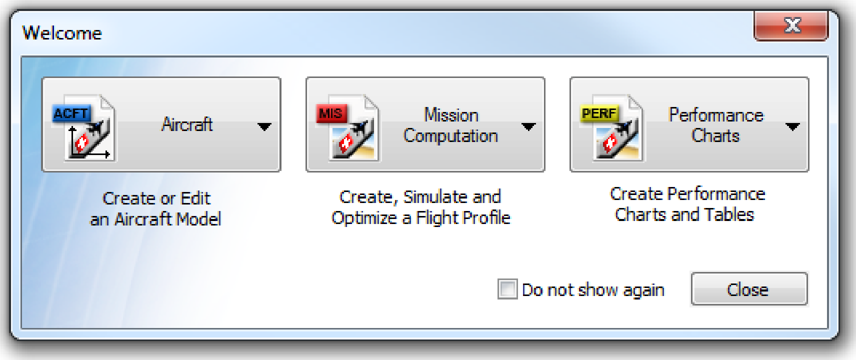
Performance Classes for executing *Performance Charts* computations and reading the results.

Database Helper class to read APP's string table.

Global Constants as used in APP.

Units Unit conversion factors as used in APP.

6.2 Reading and Writing APP Files



For each APP7 filetype, pyAPP7 offers a class to read, manipulate and write a file. The classes are located in the Files module:

APP File	pyAPP7 Class
Aircraft (.acft)	Files.AircraftModel
Mission Computation (.mis)	Files.MissionComputationFile
Performance Charts (.perf)	Files.PerformanceChartFile

The classes are located in the Files module:

class `pyAPP7.Files.`**`AircraftModel`**

Holds the APP7 aircraft model that is used to read and write APP .acft files

Each type of data (Mass&Limits, Aerodynamcis, Propulsion, Stores) is stored in two lists: one list containing names and one list containing data. These two lists have to have the same length. The configurations are built by using these list indices. Take proper care when manipulating these lists manually and update the 'ProjectAircraft' (m_Prj).

Examples

The best way to create an instance of an AircraftModel is to use the classmethod fromFile:

```
from pyAPP7 import Files

acft = Files.AircraftModel.fromFile(r'myAircraft.acft')
```

Variables

- **m_GeneralData** (GeneralData) – General data about the aircraft
- **text** (Text) – Content of the comment text box in 'General Data'
- **configName** (*list[str]*) – list holding the names of the Mass&Limits datasets ('Config' classes)
- **aeroName** (*list[str]*) – list holding the names of the Aerodynamics datasets ('Aero' classes)
- **propulsionName** (*list[str]*) – list holding the names of the Propulsion datasets ('PropulsionData' child classes)
- **storeName** (*list[str]*) – list holding the names of the Store datasets ('Store' classes)
- **m_config** (*list[Config]*) – list of the Mass&Limits datasets ('Config' classes)
- **m_aero** (*list[Aero]*) – list of the Aerodynamics datasets ('Aero' classes)
- **m_propulsion** (*list[PropulsionData]*) – list of the Propulsion datasets ('PropulsionData' child classes)
- **m_store** (*list[Store]*) – list of the Store datasets ('Store' classes)
- **m_Prj** (ProjectAircraft) – Contains the Configurations and Store Configurations

class pyAPP7.Files.**MissionComputationFile**

Reads an APP .mis file

This class reads an APP mission computation file. Most of the data is stored in a ProjectAircraftSetting object (aircraft configuration and stores) and a MissionDefinition object (initial conditions, list of segments). When manipulating mission files, consult the source code and documentation of these two classes.

Note: Data for APP's "Parameter Study" computation mode is read as well (into the variationData attribute). However, APP's command line mode does not support this computation type

Examples

The best way to create an instance of a MissionComputationFile is to use the classmethod fromFile:

```
from pyAPP7 import Files

mis = Files.MissionComputationFile.fromFile(r'myMission.mis')
```

Variables

- **text** (Text) – Description text
- **name** (*str*) – name of the mission computation
- **author** (*str*) – name of the author of the mission file

- **aircraftpath** (*str*) – path to the aircraft, either relative (to the location of the mis file) or absolute
- **projectAircraftSetting** (ProjectAircraftSetting) – holds the used configuration of the aircraft and settings of stores
- **misDef** (MissionDefinition) – Holds the initial conditions and the list of segments
- **resData** (ResArrayData) – Holds the computation type (CMP_MISSION or CMP_MISSIONVAR)
- **variationData** (VariationData) – Holds data for the Parameter Study mission computation type

class pyAPP7.Files.**PerformanceChartFile**

Reads an APP .perf file

This class reads an APP performance chart file. Most of the data is stored in a ProjectAircraftSetting object (aircraft, configuration and stores), a FlightData object (initial conditions and flight state) and a PointPerfSolver child class object (specific data, related to the type of performance chart).

Note: Not all types of point performance charts can be computed by the APP command line mode. See documentation for valid types.

Examples

The best way to create an instance of a PerformanceChartFile is to use the classmethod fromFile:

```
from pyAPP7 import Files

chart = Files.PerformanceChartFile.fromFile(r'myPerfFile.perf')
```

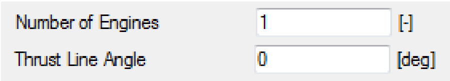
Variables

- **text** (Text) – Description text
- **name** (*str*) – name of the mission computation
- **author** (*str*) – name of the author of the mission file
- **aircraftpath** (*str*) – path to the aircraft, either relative (to the location of the perf file) or absolute
- **projectAircraftSetting** (ProjectAircraftSetting) – holds the used configuration of the aircraft and settings of stores
- **flightData** (FlightData) – holds the flight state (initial conditions)
- **perf** (PointPerfSolver) – instance of a child class of PointPerfSolver, defines the type of performance chart

NExtReal

APP defines two custom data types: NExtReal and XTables. When using pyAPP7 to manipulate APP files, it is important to understand these data types.

NExtReals are recognizable in the APP user interface by a text followed by a value and a unit:



class pyAPP7.Datatypes.**NExtReal**

APP datatype that wraps a float and allows to specify a label, type of variable (through an index string) and indicate if the value is a limiter

Variables

- **xx** (*float*) – value of variable
- **label** (*str*) – label of the value, e.g. '[Mach]'
- **realIdx** (*str*) – index (type) of variable, e.g. 'REAL_MACH'
- **limitActive** (*int*) – 0 or 1, depends on whether the variable has an active limit. E.g used for Max. Take-Off Mass

Note: use readASCIILimited and writeASCIILimited if the variable is a limited value.

Examples

When using pyAPP7 to read APP files, usually no direct use of this type is needed. This information is mostly for developers/maintainers. The text format of a simple, non-limited NExtReal looks like this:

```
[Mach]
REAL_MACH
0.985
```

This is parsed using readASCII with the flag full=True. The full flag has to be set to True to read the index string 'REAL_MACH'.

```
>>> val = NExtReal()
>>> f = open('path to text file')
>>> val.readASCII(f, full=True)
```

resulting in the following attributes:

```
val.xx = 0.985
val.realIdx = 'REAL_MACH'
val.label = '[Mach]'
val.limitActive = 0
```

If the text format has no index string,

```
[Mach]
0.985
```

readASCII is called with with the flag full=False:

```
>>> val = NExtReal()
>>> f = open('path to text file')
>>> val.readASCII(f)
```

XTable

XTables are used everywhere you see a spreadsheet-like table in APP. pyAPP7 uses the **numpy** module to store data tables. **numpy** offers a lot of functionality to manipulate arrays. pyAPP7 defines four different tables, with increasing dimensionality: X0Table, X1Table, X2Table and X3Table.

The X0Table is used for one-column data ranges, for example in performance charts for the **X-Range** and **Parameter** range.

class `pyAPP7.Datatypes.`**`X0Table`**

Holds a 1D table (data range)

Variables

- **`data`** (*`list[str]`*) – Table data with table factor and interpolation settings
- **`table`** (*`ndarray`*) – numpy array of shape (N,1)
- **`label`** (*`str`*) – Header string
- **`X0Typ`** (*`str`*) – APP variable type

The X1Table is a simple two-column table. An example would be the Mach limit or CLmax table. The data is stored in a two-dimensional numpy array.

class `pyAPP7.Datatypes.`**`X1Table`**

Holds a 2D table

Variables

- **`data`** (*`list[str]`*) – Table data with table factor and interpolation settings
- **`table`** (*`ndarray`*) – numpy array of shape (N,2)
- **`label`** (*`str`*) – Header string

The X2Table is a list of two-column tables. An example would be the induced drag tables or the max. thrust tables. The data is stored in a list of two-dimensional numpy arrays (*table* attribute). Each table also has a value (for the induced drag table that would be a Mach number). The values are stored in the *value* list. The *table* and *value* list have the same length and same ordering.

class `pyAPP7.Datatypes.`**`X2Table`**(*embedded=False*)

Holds a list of 2D tables

Variables

- **`data`** (*`list[str]`*) – Table data with table factor and interpolation settings
- **`table`** (*`list[ndarray]`*) – list of numpy arrays of shape (N,2)
- **`value`** (*`list[float]`*) – value of each table
- **`label`** (*`str`*) – Header string
- **`embedded`** (*`bool`*) – True if table is embedded in an 'X3Table'. Disables reading/writing of header (data and label)

The X3Table class is used in APP for the fuel flow table. The X3Table consists of a list of X2Tables and corresponding values.

class `pyAPP7.Datatypes.`**`X3Table`**

Holds a list of X2Tables.

This class holds a list of X2Tables and a value for each table.

Variables

- **`data`** (*`list[str]`*) – Table data with table factor and interpolation settings
- **`table`** (*`list[X2Table]`*) – list of X2Table instances
- **`value`** (*`list[float]`*) – value of each table

- **label** (*str*) – Header string

6.3 Variables

When executing APP via the command line, the user can specify what variables will be written in the output result file. By default, pyAPP7 uses the included *ParameterList_All.par* file to specify the variables. The ouput data is stored in a large numpy table, and the variable can be best accessed by it's index. The following table presents the current mapping of indices to the variables when using the default parameter list file.

Index	Variable Name	(SI)	(British)
0	Acceleration	[m/sec2]	[ft/sec2]
1	SEP (Accel)	[m/sec2]	[KTS/s]
2	X-Acc.	[m/sec2]	[ft/sec2]
3	Z-Acc.	[m/sec2]	[ft/sec2]
4	Advance Ratio	[-]	[-]
5	Altitude	[m]	[ft]
6	AoA	[deg]	[deg]
7	Attitude	[deg]	[deg]
8	Battery Energy	[kJ]	[Wh]
9	Battery SOC	[%]	[%]
10	Propeller Beta	[deg]	[deg]
11	CAS	[m/sec]	[nm/hr]
12	CD	[-]	[-]
13	CD0	[-]	[-]
14	CDi	[-]	[-]
15	CDs	[-]	[-]
16	CL	[-]	[-]
17	CL/CD	[-]	[-]
18	CLmax	[-]	[-]
19	CO2 Mass	[kg]	[lbs]
20	Thrust cos(AoA+sigma)	[N]	[lbf]
21	CP	[-]	[-]
22	(M/SFC)(L/D)	[-]	[-]
23	CT	[-]	[-]
24	Density	[kg/m3]	[slug/ft3]
25	Distance	[km]	[nm]
26	Drag	[N]	[lbf]
27	Drag Area	[m2]	[ft2]
28	dT	[K]	[K]
29	EAS	[m/sec]	[nm/hr]
30	Energy Height	[m]	[ft]
31	Ekin	[Nm]	[lbf ft]
32	Epot	[Nm]	[lbf ft]
33	Energy Specific Range	[m/J]	[m/J]
34	Etot	[Nm]	[lbf ft]
35	Friction Force	[N]	[lbf]
36	Fuel Flow	[kg/sec]	[lbs/hr]
37	Fuel Mass	[kg]	[lbs]
38	Fuel Percent	[%]	[%]
39	Fuel Percent (Internal)	[%]	[%]
40	Climb Angle	[deg]	[deg]
41	Generator Power	[%]	[%]
42	Ground Force	[N]	[lbf]
43	Load Factor	[-]	[-]

continues on next page

Table 1 – continued from previous page

44	Lift	[N]	[lbf]
45	Lift Area	[m2]	[ft2]
46	Placard Mach	[-]	[-]
47	Mach	[-]	[-]
48	Mass	[kg]	[lbs]
49	Max. Thrust	[N]	[lbf]
50	Min. Thrust	[N]	[lbf]
51	Motor Eta	[%]	[%]
52	Motor Torque	[Nm]	[lbf ft]
53	Payload	[%]	[%]
54	Power Setting	[%]	[%]
55	Power Consumption	[W]	[W]
56	Power Required	[W]	[shp]
57	Pull-Up Rate	[deg/sec]	[deg/sec]
58	Pressure Altitude	[m]	[ft]
59	Pressure	[N/m2]	[lbf/ft2]
60	Propeller Efficiency	[%]	[%]
61	Dynamic Pressure	[N/m2]	[lbf/ft2]
62	Engine Revolution	[rpm]	[rpm]
63	Seg. CO2 Mass	[kg]	[lbs]
64	Seg. Dist.	[km]	[nm]
65	Seg. Fuel	[kg]	[lbs]
66	Seg. Time	[min]	[min]
67	SEP	[m/sec]	[ft/sec]
68	Configuration Nr.	[-]	[-]
69	SFC	[kg/(sec N)]	[lbs/(hr lbf)]
70	Shaft Power	[W]	[shp]
71	Speed of Sound	[m/sec]	[ft/sec]
72	Specific Range	[km/kg]	[nm/lbs]
73	Reference Area	[m2]	[ft2]
74	TAS	[m/sec]	[nm/hr]
75	Temperature	[K]	[K]
76	Thrust	[N]	[lbf]
77	Time	[sec]	[sec]
78	Turn Radius	[m]	[ft]
79	Turn Rate	[deg/sec]	[deg/sec]
80	T/Tmax	[-]	[-]
81	Turns	[turn]	[turn]
82	Velocity	[m/sec]	[nm/hr]
83	Minimum Unstick Speed	[m/sec]	[nm/hr]
84	Minimum Unstick Speed (CAS)	[m/sec]	[nm/hr]
85	Stall Speed	[m/sec]	[nm/hr]
86	Stall Speed (CAS)	[m/sec]	[nm/hr]
87	Vx	[m/sec]	[nm/hr]
88	Climb Speed	[m/sec]	[ft/sec]

6.4 Mission Computations

The Mission module, specifically the class *MissionComputation*, is used to run the APP command line mode for mission computations and parse the result text file.

class pyAPP7.Mission.**MissionComputation**(*APP7Directory='C:\\Program Files (x86)\\ALR Aerospace\\APP 7 Professional Edition'*)

Class to execute APP and subsequently load the results.

This class is a helper class to execute APP mission computations. After creating an instance of this object, execute the 'run' function. The result will be loaded into the 'result' attribute. 'result' is of type MissionResult, see the documentation of the MissionResult class for further details.

Note: The 'Parameter Study' computation type can not be computed with the APP command line mode.

Examples

This example shows how to run a mission computation and obtain an instance of the mission result:

```
from pyAPP7 import Mission

misCmp = Mission.MissionComputation()
misCmp.run(r'myMission.mis')
result = misCmp.getResult()
```

This example assumes APP is installed in the default directory.

Variables

- **output** (*str*) – Path to the text file with the mission results written by APP
- **result** (MissionResult) – The result of the mission computation, parsed from the 'output' text file
- **misCompFile** (Files.MissionComputationFile) – Instance of a MissionComputationFile (APP .mis file). Is available once the method run was called
- **db** (*Database*) – Instance of a Database object
- **inputfile** (*str*) – Path to the APP mis file

A result from an APP command line computation can also be directly read by using the MissionResult class.

class pyAPP7.Mission.**MissionResult**

This class can read the APP mission result text file.

Examples

This example shows how to read a mission result directly from a text file. This is useful to read results from past mission computations, for example when conduction batch simulations:

```
from pyAPP7 import Mission

res = Mission.MissionResult.fromFile(r'myMission.mis_ouput.txt')
```

Variables

- **output** (*dict*) – Dictionary containing the mission flags, error text, number- and list of variables

- **segments** (*list [MissionResultSegment]*) – A list of MissionResultSegment class instances, holding the results of each segment
- **initialSettings** (*MissionResultSegment ()*) – The initial settings of the mission

6.5 Performance Charts

The Performance module, specifically the class *PerformanceChart*, is used to run the APP command line mode for performance chart computations and parse the result text file.

class pyAPP7.Performance.**PerformanceChart** (*APP7Directory='C:\\Program Files (x86)\\ALR Aerospace\\APP 7 Professional Edition'*)

Helper class to execute a performance chart computation from an existing .perf file.

Variables

- **inputfile** (*str*) – Path to the input .perf file
- **perfFile** (*PerformanceChartFile*) – Parsed input APP7 .perf file
- **output** (*str*) – Path to the resulting txt file
- **result** (*PerformanceChartResult*) – Result
- **APP7Path** (*str*) – Full path to the APP7 executable

Parameters **APP7Directory** (*str, optional*) – Path to the location of the APP7 executable.

Raises **ValueError** – If the APP7 executable is not found in the specified directory

A result from an APP command line computation can also be directly read by using the PerformanceChartResult class.

class pyAPP7.Performance.**PerformanceChartResult**

Reads a result txt file written by the APP7 command line mode for a performance chart.

Examples

This example shows how to read a performance chart result directly from a text file. This is useful to read results from past computations, for example when conduction batch computations:

```
from pyAPP7 import Performance

res = Performance.PerformanceChartResult.fromFile(r'myChart.perf_ouput.txt')
```

Variables

- **output** (*dict*) – stores the result meta-data
- **lines** (*List [ResultLine]*) – holds the data of each line of a performanc chart

7 pyAPP7 Examples

Content: These examples are grouped into three main sections:

- *Files*
- *Mission Computation*
- *Performance Charts*

Version: pyAPP7 version 1.0

Note: This example was written as a jupyter notebook (version 4.4.0), and has been tested with Python 2.7.16 |Anaconda (64-bit). The notebook file is available in the *Examples* directory of the pyAPP7 distribution.

7.1 Imports & Constants

Imports for plotting (matplotlib) and arrays (numpy):

[1]:
```
import matplotlib.pyplot as plt
import numpy as np
```

Jupyter Notebook specific imports:

[2]:
```
%matplotlib inline
```

Constants:

[3]:
```
APP7DIR = r'C:\Program Files (x86)\ALR Aerospace\APP 7 Professional Edition'
```

7.2 Files

The Files module is used to open, change and save APP Files. It can be used for: * *acft* (Aircraft) * *mis* (Mission Computation) * *perf* (Performance Charts)

file types.

It is recommended to create a new file using the APP GUI and subsequenty modify this file using Python/pyAPP7, instead of creating a file from scratch with pyAPP7.

Import the pyAPP7 modules

[4]:
```
from pyAPP7 import Files
from pyAPP7 import Database
from pyAPP7 import Units
```

The Units and Database modules are imported as well for this example. They are useful to convert units and translate APP indices to human-readable text

Aircraft File (*.acft)

To load an APP aircraft model, the class *AircraftModel* is used. A new instance can be created directly with the *fromFile* class method:

[5]:
```
aircraftpath = r'data\\LWF.acft'
acft = Files.AircraftModel.fromFile(aircraftpath)
```

Now we have the aircraft file available in the *acft* variable. All data within the aircraft can be accessed through class member variables directly, or by using *get* functions. This examples shows how to access fields in the *General Data* tab of APP's aircraft model GUI:

[6]:
```
data = acft.getGeneralData()
print ('Aircraft Name:', data.m_sAircraftName)
print ('Author:', data.m_sAuthor)
```

```
('Aircraft Name:', 'LWF')
('Author:', 'ALR')
```

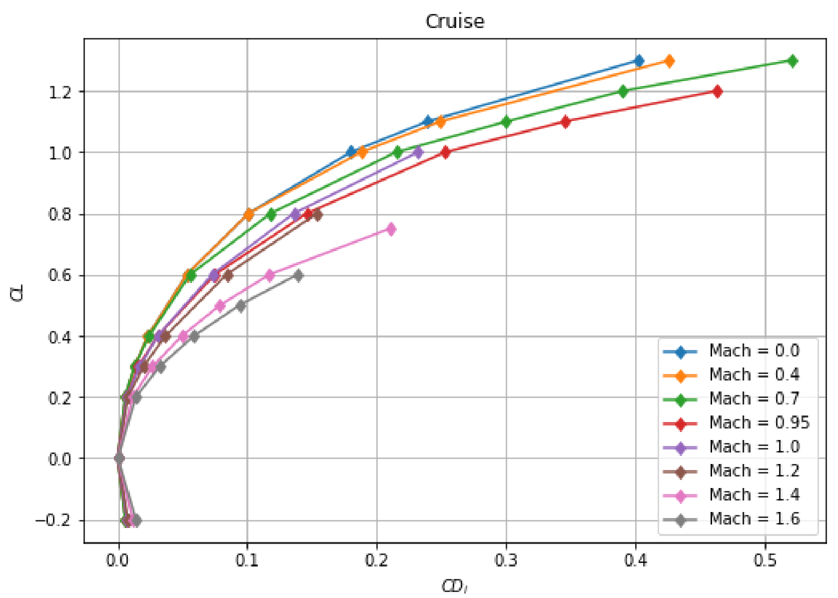
Getter functions exist for all the main datasets. To print lists of the available data sets, use:

[7]:
```
print(acft.getMassLimitsNames())
print(acft.getAeroNames())
print(acft.getPropulsionNames())
print(acft.getStoreNames())
```

```
['Standard']
['Cruise', 'TO Flaps 27\xb0']
['LWF']
['AIM-9 Wingtip']
```

This example demonstrates how to loop through an X2Table (in this case the CL/CDi table) and correctly lable the drag polars:

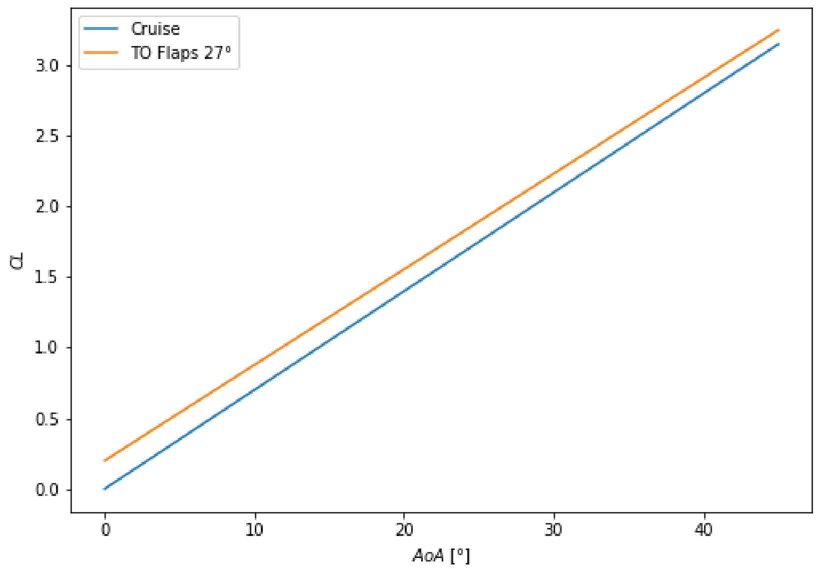
[8]:
```
i = 0
aero = acft.getAero(i) #get the first aerodanymic dataset, in this case 'Cruise'

fig = plt.figure(figsize=(8.3, 5.8)) #A5 landscape figure, size is in inches
ax = plt.subplot(1,1,1)

for val, table in zip(aero.cdITable.value,aero.cdITable.table):
    ax.plot(table[:,1], table[:,0], 'd-', label='Mach = '+str(val))

# adjust Axis properties
ax.set_title(acft.getAeroName(i))
ax.legend(loc='best')
ax.set_xlabel('$CD_i$')
ax.set_ylabel('$CL$')
ax.grid()
```



For an detailed explaination of the XTables classes, consult the pyAPP user guide.

A more involved example would be to compare lift curves of all available aero datasets:

```
[9]: fig = plt.figure(figsize=(8.3, 5.8)) #A5 landscape figure, size is in inches
ax = plt.subplot(1, 1, 1)
for aero, aeroName in zip(acft.getAeroList(), acft.getAeroNames()):
    ax.plot(aero.clTable.table[0][:,0]*Units.DEG, aero.clTable.table[0][:,1],␣
↪label=aeroName.decode('cp1252'))

ax.set_xlabel(u'$AoA$ [°]')
ax.set_ylabel(u'$CL$')
leg = ax.legend(loc=2)
#fig.savefig('CL_comparison.png',dpi=200)
```



Additionally, this example demonstrates the use of the *Units* module to convert from radians to degrees.

Note: In oder for the legend label for the *TO Flaps 27°* setting to be printed correctly, the *aeroName* string has to be converted to unicode with the enconding of the original text file, in this case *cp1252*. In addition, to print the ° sign in the x-axis label, the string has to be unicode and is typed with the prefix 'u'

Mission File (*.mis)

The mission file is loaded using the classmethod *fromFile* in the MissionComputationFile class:

```
[10]: missionpath = r'data\\LWF Air Combat Mission RoA.mis'
missionFile = Files.MissionComputationFile.fromFile(missionpath)
```

We have now the mission file as a python variable *missionFile* in the memory ready to be be examined and changed.

For example, *getInitialCondition()* can be used to access the initial conditions. The return value is of type *Files.FlightData*

```
[11]: initFd = missionFile.getInitialCondition()
print(initFd.alt.xx) #altitude in meters
print(initFd.fuel.xx) #initial fuel as a factor [0...1]

0.0
1.0
```

To loop through the segments, use *getSegmentList()* to access the list of segments. The following code prints the segment index (identifier) of each segment:

[12]:
```
for segment in missionFile.getSegmentList():
    print(segment.segmentIndex)
```

```
SEG_GROUNDOP
SEG_TAKEOFF
SEG_CLIMB
SEG_BESTCLIMBRATE
SEG_ACCELERATION
SEG_TARGETMACHCRUISE
SEG_MANEUVRE
SEG_STOREDROP
SEG_MANEUVRE
SEG_STOREDROP
SEG_LOITER
SEG_SPECIFICRANGE
SEG_DECELERATION
SEG_CASDESCENT
SEG_LANDINGROLL
```

In order to display the label of each segment instead of the index string, we can use the Database class:

[13]:
```
db = Database.Database()
```

[14]:
```
for segment in missionFile.getSegmentList():
    print(db.GetTextFromID(segment.segmentIndex))
```

```
Ground Operation
Takeoff
Climb
Climb at Best Rate
Acceleration
Cruise at Mach
Maneuver at Max. LF
Store Drop
Maneuver at Max. LF
Store Drop
Loiter
Cruise at Best SR
Deceleration
Descent at CAS
Landing Roll
```

Similarly, the type and value of the segment end condition can shown:

[15]:
```
for segment in missionFile.getSegmentList():
    print(db.GetTextFromID(segment.endValue1.realIdx),':', segment.endValue1.xx, )
```

```
('Seg. Time', ':', 600.0)
('Velocity', ':', 75.4455900943)
('Altitude', ':', 500.0)
('Altitude', ':', 9500.0)
('Mach', ':', 0.9)
('Seg. Dist.', ':', 320053.202172)
('Turns', ':', 12.5663706144)
('Seg. Time', ':', 100.0)
('Turns', ':', 6.28318530718)
('Seg. Dist.', ':', 100.0)
('Seg. Time', ':', 600.0)
('Seg. Dist.', ':', 402135.694779)
('CAS', ':', 102.888888976)
('Altitude', ':', 500.0)
('Velocity', ':', 0.01)
```

In the following code examples we show how to make changes to the mission and save it to a new file.

The frist example shows how to change the initial fuel mass to 80% and the initial altitude to 1000 m:

[16]:
```
initFd = missionFile.getInitialCondition()
initFd.fuel.xx = 0.8
initFd.alt.xx = 1000.0
```

Next, we change parameters of a segment, in this example the altitude (stop condition) of the segment "Climb at Best Rate" (segment index 3) from 9500m to 7000m:

[17]:
```
print(missionFile.getSegment(3).endValue1.xx)
missionFile.getSegment(3).endValue1.xx = 7000.0
print(missionFile.getSegment(3).endValue1.xx)
```

```
9500.0
7000.0
```

In addition, we change the altitude of the initial climb after takeoff (Segment index 2) to 500m above the starting altitude.

[18]:
```
print(missionFile.getSegment(2).endValue1.xx)
missionFile.getSegment(2).endValue1.xx = initFd.alt.xx + 500.0
print(missionFile.getSegment(2).endValue1.xx)
```

```
500.0
1500.0
```

Finally, we save the changed mission to a new file.

[19]:
```
missionpath_mod = r'data\\LWF Air Combat Mission RoA_mod.mis'
missionFile.saveToFile(missionpath_mod,overwrite=True)
```

Performance Chart File (*.perf)

A PerformanceChartFile is instantiated via the fromFile classmethod:

[20]:
```
chartpath = r'data\\LWF Climb Rate Chart 50% Fuel.perf'
chart = Files.PerformanceChartFile.fromFile(chartpath)
```

This example shows how to change the flight state (initial condition). The function *getInitialCondition* returns an instance of type FlightData:

[21]:
```
fd = chart.getInitialCondition()
```

[22]:
```
print fd.alt.xx
print fd.speed.xx, db.GetTextFromID(fd.speed.realIdx) #Mach Number
print fd.fuel.xx, db.GetTextFromID(fd.fuel.realIdx)
```

```
0.0
0.0 Mach
0.5 Fuel Percent
```

Note: the speed variable can be either **Mach** or **TAS**. Check the corresponding *realIdx* string. Similarly, the variables *payload*, *climb*, *thrust* and *pull* can be of different type

Change the fuel from the current state (50%) to 100%

[23]:
```
print(fd.fuel.xx)
```

```
0.5
```

[24]:
```
fd.fuel.xx = 1.0
```

To change the aircraft **Configuration**, for example from Dry (configuration index 0) to Reheat (configuration index 1), access the *ProjectAircraftSetting* class. To see what configurations are available, open the aircraft model.

[25]:
```
configNames = acft.getConfigurationNames()
print 'Configurations in the aircraft model:\n', configNames, '\n'

cfg = chart.getAircraftConfiguration()
print cfg.activeSetting, configNames[cfg.activeSetting]
cfg.activeSetting = 1
print cfg.activeSetting, configNames[cfg.activeSetting]
```

```
Configurations in the aircraft model:
['Cruise, Dry', 'Cruise, Reheat', 'TOL, Reheat', 'TOL, Dry']

0 Cruise, Dry
1 Cruise, Reheat
```

Similarly, *External Store Configurations* can be changed:

[26]:
```
storeConfigNames = acft.getStoreConfigurationNames()
print 'Store configurations in the aircraft model:\n',storeConfigNames,'\n'

cfg = chart.getAircraftConfiguration()
print cfg.activeStoreSetting, storeConfigNames[cfg.activeStoreSetting]
cfg.activeStoreSetting = -1 #use -1 for no external stores (clean)
```

```
Store configurations in the aircraft model:
['Air-to-Air']

0 Air-to-Air
```

To access the computation, use the *getComputation* method. The type of performance chart can be checked with the *CompType* variable. In the case of a **Point Performance Computation**, the type of equation solved is stored in *resData.CmpType*.

[27]:
```
comp = chart.getComputation()
print db.GetTextFromID(comp.CompType)
print db.GetTextFromID(comp.resData.CmpType)
```

```
Point Performance Computation
Climb
```

The *resData* attribute also holds the data ranges for the chart in two *X0Tables*, one for the **X-Range** the other for the **Parameter**:

[28]:
```
print comp.resData.X1Range.X0Typ
print comp.resData.X1Range.table

print comp.resData.X2Range.X0Typ
print comp.resData.X2Range.table
```

```
REAL_MACH
[ 0.2   0.25  0.3   0.35  0.4   0.45  0.5   0.55  0.6   0.65  0.7   0.75
  0.8   0.85  0.9   0.95]
REAL_ALT
[     0.   2500.   5000.   7500.  10000.]
```

For example, to change the computed altitudes, replace the *table* with a new numpy array:

[29]:
```
comp.resData.X2Range.table = np.linspace(0.0, 10000.0, 3)
print comp.resData.X2Range.table
```

```
[     0.   5000.  10000.]
```

or, add values manually (as *floats*):

[30]:
```
comp.resData.X2Range.table = np.array([0.0, 10000.0])
print comp.resData.X2Range.table
```

```
[     0.  10000.]
```

Save your modified file:

[31]:
```
chartpath_mod = r'data\\LWF Climb Rate Chart 100% Fuel.perf'
chart.saveToFile(chartpath_mod, overwrite=True)
```

7.3 Mission Computation

Import the *Mission* module from pyAPP7:

[32]:
```
from pyAPP7 import Mission
```

In order to run APP mission computations, create an instance of the *MissionComputation* class. The path to the directory where the APP executable can be found has to be provided

[33]:
```
misCmp = Mission.MissionComputation(APP7Directory = APP7DIR)
```

[34]:
```
misCmp.run(missionpath)
```

[34]:
```
True
```

[35]:
```
res = misCmp.result
```

Access data by looping through the segments. To get a specific variable, find the index of the variable by using the function *getVariableIndex*. To access the data of the segment, use *getData*. *getData* returns a 2D numpy array, with the first dimension being the datapoint and the second dimension the variable. For example, the variable *Fuel Mass* at the end of each segment can be obtained by using:

[36]:
```
idx_fuel = res.getVariableIndex('Fuel Mass')

for seg in res.getSegmentList():
    print res.getVariableName(idx_fuel),':',seg.getData()[-1,idx_fuel]
```

```
Fuel Mass [kg] : 1896.218
Fuel Mass [kg] : 1859.49218997
Fuel Mass [kg] : 1779.27456082
Fuel Mass [kg] : 1532.93143492
Fuel Mass [kg] : 1520.88752268
Fuel Mass [kg] : 1123.86312629
Fuel Mass [kg] : 914.583951541
Fuel Mass [kg] : 914.583951541
Fuel Mass [kg] : 815.505620848
Fuel Mass [kg] : 815.505620848
Fuel Mass [kg] : 619.613596679
Fuel Mass [kg] : 225.152121272
Fuel Mass [kg] : 222.695756009
Fuel Mass [kg] : 104.648393277
Fuel Mass [kg] : 100.120415794
```

Instead of using getData to access the raw output, we can call *getVariableData* and get a list of numpy arrays for the output of a specific variable:

[37]:
```
var_name, mission_data = res.getVariableData('Fuel Mass')
print var_name
print len(mission_data)
print mission_data[0] #time-dependent data of the first segment
```

```
Fuel Mass [kg]
15
[ 2000.       1998.2703  1996.5406  1994.8109  1993.0812  1991.3515
  1989.6218  1987.8921  1986.1624  1984.4327  1982.703   1980.9733
  1979.2436  1977.5139  1975.7842  1974.0545  1972.3248  1970.5951
  1968.8654  1967.1357  1965.406   1963.6763  1961.9466  1960.2169
  1958.4872  1956.7575  1955.0278  1953.2981  1951.5684  1949.8387
  1948.109   1946.3793  1944.6496  1942.9199  1941.1902  1939.4605
  1937.7308  1936.0011  1934.2714  1932.5417  1930.812   1929.0823
  1927.3526  1925.6229  1923.8932  1922.1635  1920.4338  1918.7041
  1916.9744  1915.2447  1913.515   1911.7853  1910.0556  1908.3259
  1906.5962  1904.8665  1903.1368  1901.4071  1899.6774  1897.9477
  1896.218   1896.218 ]
```

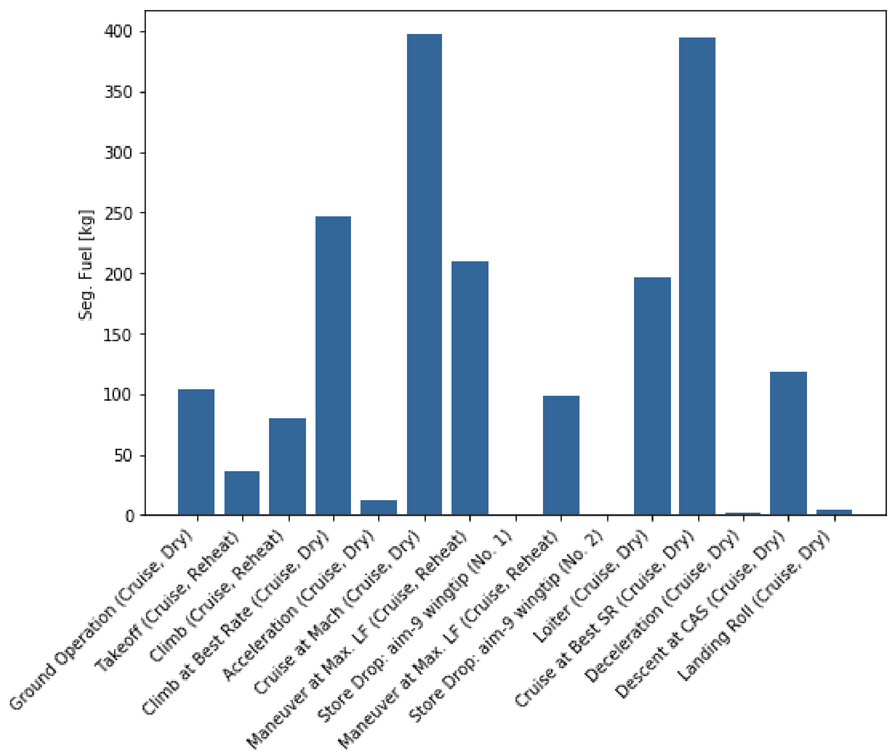
A list of the fuel consumed per segment can be easily ontained using a list comprehension:

[38]:
```
idx_segFuel = res.getVariableIndex('Seg. Fuel')

segFuelList = [seg.getData()[-1,idx_segFuel] for seg in res.getSegmentList()]
print segFuelList
```

```
[103.782, 36.7258100321, 80.2176291448, 246.34312590799999, 12.043912239799999, 
↪397.02439638800001, 209.279174746, 0.0, 99.0783306923, 0.0, 195.892024169, 394.
↪46147540700002, 2.4563652631499999, 118.047362732, 4.5279774831899999]
```

[39]:
```
fig = plt.figure(figsize=(8.3, 5.8)) #A5 landscape figure, size is in inches
ax = plt.subplot(1,1,1)
ax.bar(range(len(segFuelList)),
       segFuelList,
       align='center',
       color='#336699')
ax.set_xticks(range(len(segFuelList)))
ax.set_xticklabels(res.getSegmentNameList(), rotation=45, ha='right')
ax.set_ylabel(res.getVariableName(idx_segFuel))
```

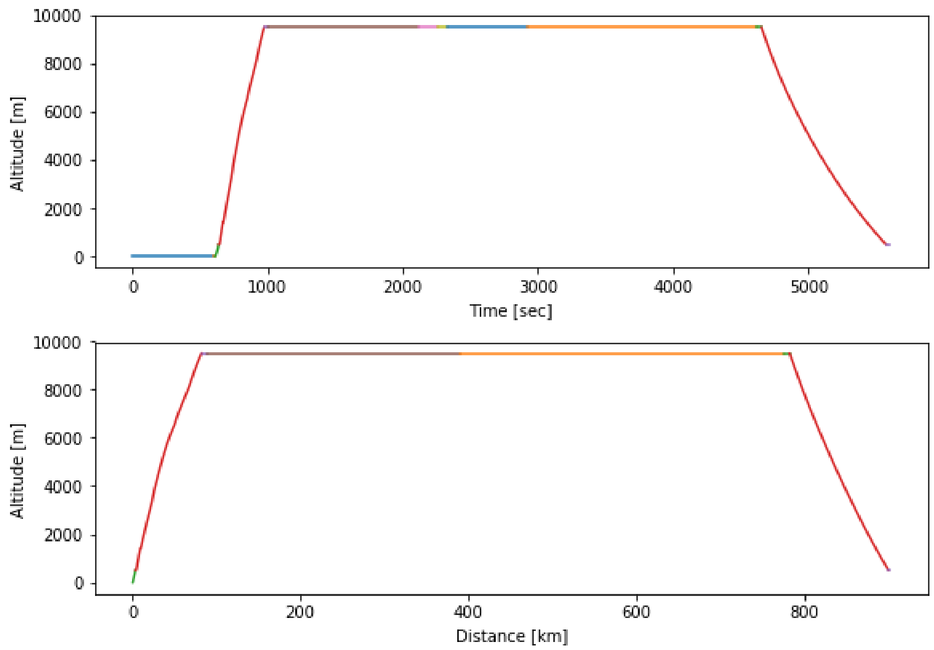
[39]:
```
Text(0,0.5,'Seg. Fuel [kg]')
```



Looping through the segments can also be useful to plot the mission profile:

```
[40]: idx1 = res.getVariableIndex('Time')
idx2 = res.getVariableIndex('Distance')
idx3 = res.getVariableIndex('Altitude')
```

```
[41]: fig = plt.figure(figsize=(8.3, 5.8)) #A5 landscape figure, size is in inches

ax = plt.subplot(2,1,1)
for seg in res.getSegmentList():
    ax.plot(seg.getData()[:,idx1],seg.getData()[:,idx3])

ax.set_xlabel(res.getVariableName(idx1))
ax.set_ylabel(res.getVariableName(idx3))

ax = plt.subplot(2,1,2)
for seg in res.getSegmentList():
    ax.plot(seg.getData()[:,idx2],seg.getData()[:,idx3])

ax.set_xlabel(res.getVariableName(idx2))
ax.set_ylabel(res.getVariableName(idx3))

plt.tight_layout()
```



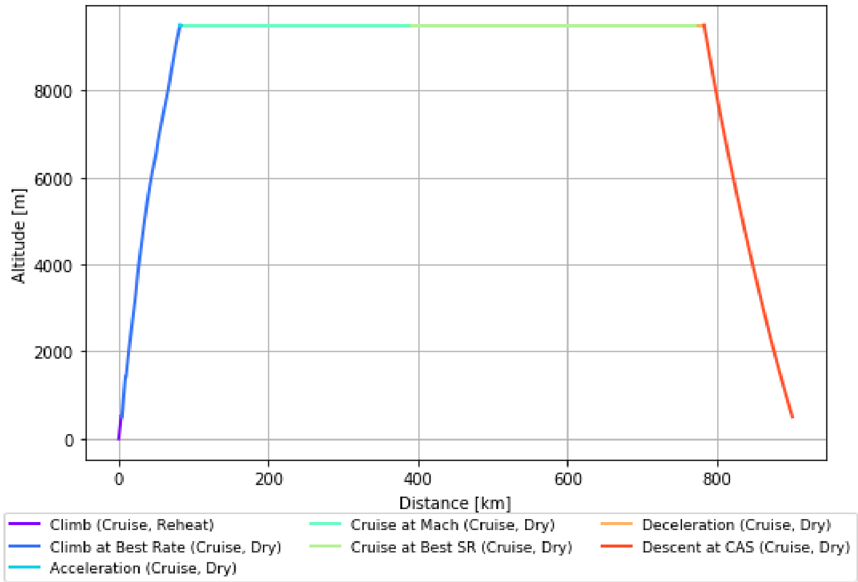
Matplotlib offers a lot of formatting options for legends: http://matplotlib.org/api/legend_api.html#matplotlib.legend.Legend

```
[42]: idx1 = res.getVariableIndex('Distance')
idx2 = res.getVariableIndex('Altitude')
idx_segDst = res.getVariableIndex('Seg. Dist')

fig = plt.figure(figsize=(8.3, 5.8)) #A5 landscape figure, size is in inches
ax = plt.subplot(1,1,1)

colormap = plt.cm.rainbow
ax.set_prop_cycle('color',[colormap(i) for i in np.linspace(0, 0.9, 7)])

for i,seg in enumerate(res.getSegmentList()):
    if seg.getData()[-1,idx_segDst]>2.0:
```

(continues on next page)

(continued from previous page)

```
        ax.plot(seg.getData()[:,idx1], seg.getData()[:,idx2], label=seg.getName(),
↪lw=2.0)

ax.set_xlabel(res.getVariableName(idx1))
ax.set_ylabel(res.getVariableName(idx2))
plt.subplots_adjust(bottom=0.2)
ax.legend(bbox_to_anchor=(1.05,-0.1), ncol=3, fontsize = 9, handlelength = 2.0)
ax.grid()
```

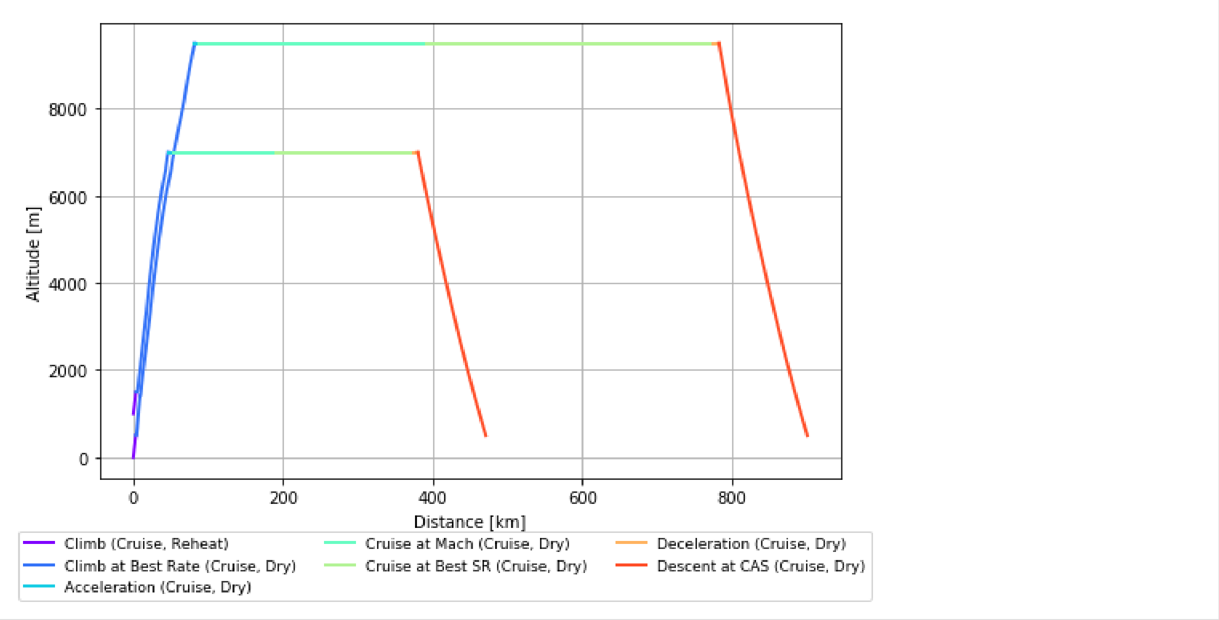


[43]:
```
misCmp_mod = Mission.MissionComputation(APP7Directory = APP7DIR)
misCmp_mod.run(missionpath_mod)
```

[43]:
```
True
```

[44]:
```
res_mod = misCmp_mod.result
idx_segDst = res.getVariableIndex('Seg. Dist')
```

[45]:
```
fig = plt.figure(figsize=(8.3, 5.8)) #A5 landscape figure, size is in inches
ax = plt.subplot(1,1,1)

colormap = plt.cm.rainbow
ax.set_prop_cycle('color',[colormap(i) for i in np.linspace(0, 0.9, 7)])

for i,seg in enumerate(res.getSegmentList()):
    if seg.getData()[-1,idx_segDst]>2.0:
        ax.plot(seg.getData()[:,idx1], seg.getData()[:,idx2], label=seg.getName(),
↪lw=2.0)

for i,seg in enumerate(res_mod.getSegmentList()):
    if seg.getData()[-1,idx_segDst]>2.0:
        ax.plot(seg.getData()[:,idx1], seg.getData()[:,idx2],lw=2.0)

ax.set_xlabel(res.getVariableName(idx1))
ax.set_ylabel(res.getVariableName(idx2))
plt.subplots_adjust(bottom=0.2)
ax.legend(bbox_to_anchor=(1.05,-0.1), ncol=3, fontsize = 9, handlelength = 2.0)
ax.grid()
```



The result of a mission computation can also be loaded from the result text-file after the computation:

[46]:
```
resfile = r'data\\LWF Air Combat Mission RoA.mis_output.txt'
res = Mission.MissionResult.fromFile(resfile)
```

Complex Mission Loop

[47]:
```
cap_path = r'data\\LWF CAP Loop.mis'
cap_path_mod = r'data\\LWF CAP Loop_mod.mis'
```

[48]:
```
mis = Files.MissionComputationFile.fromFile(cap_path)
[(i, seg.getName()) for i, seg in enumerate(mis.getSegmentList())]
```

[48]:
```
[(0, 'SEG_GROUNDOP'),
 (1, 'SEG_TAKEOFF'),
 (2, 'SEG_CLIMB'),
 (3, 'SEG_BESTCLIMBRATE'),
 (4, 'SEG_ACCELERATION'),
 (5, 'SEG_TARGETMACHCRUISE'),
 (6, 'SEG_LOITER'),
 (7, 'SEG_STOREDROP'),
 (8, 'SEG_STOREDROP'),
 (9, 'SEG_MANEUVRE'),
 (10, 'SEG_SPECIFICRANGE'),
 (11, 'SEG_NOCREDIT')]
```

[49]:
```
idx_loiter = 6
idx_combat = 9
```

[50]:
```
range_combat = np.linspace(0, 10, 6) # minutes
```

Read a CAP mission from an existing file, adjust the end-value of the combat segment and save the mission to another file. Afterwards, run the mission, extract the result and store it to a list (i.e. *loiter_time*).

[51]:
```
loiter_time = []
for i in range_combat:
    misFile = Files.MissionComputationFile.fromFile(cap_path)
```

(continues on next page)

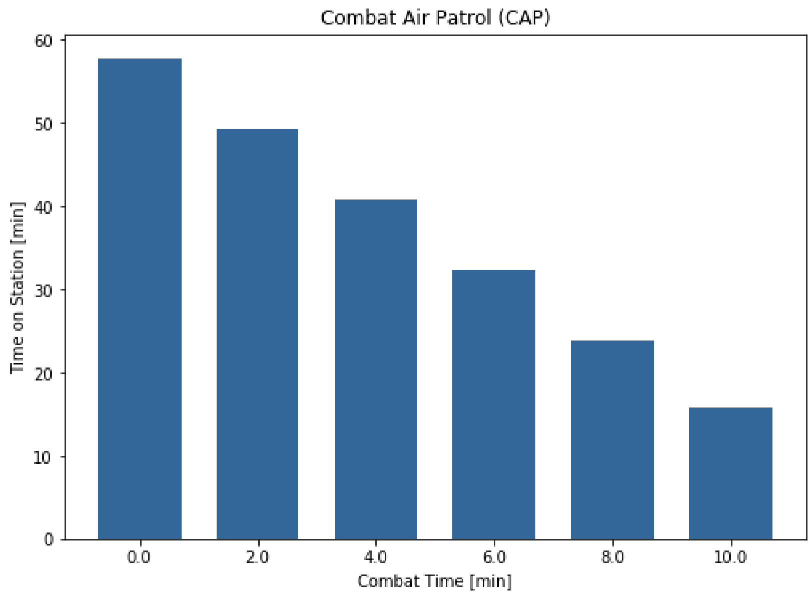
(continued from previous page)

```
    combat = misFile.getSegment(idx_combat)
    combat.endValue1.xx = i*60.0 # convert minutes to seconds
    misFile.saveToFile(cap_path_mod, overwrite=True)

    mis = Mission.MissionComputation(APP7DIR)
    mis.run(cap_path_mod)

    res = mis.getResult()

    idx_segTime = res.getVariableIndex('Seg. Time')
    loiter_time.append(res.getSegment(idx_loiter).getData()[-1,idx_segTime])
```

Plot the results as a bar-chart.

[52]:
```
fig = plt.figure(figsize=(8.3, 5.8)) #A5 landscape figure, size is in inches
ax = plt.subplot(1,1,1)
width = 1.4
ax.bar(x=range_combat-2.0*width,
       height=loiter_time, width=width,
       tick_label=[str(c) for c in range_combat],
       align='center',
       color='#336699')
ax.set_title('Combat Air Patrol (CAP)')
ax.set_xlabel('Combat Time [min]')
ax.set_ylabel('Time on Station [min]')
```

[52]:
```
Text(0,0.5,'Time on Station [min]')
```



Note: input data is always in SI units (e.g. the combat time segment *endValue* is in seconds), but the output values are formatted (e.g. loiter time is in minutes)

7.4 Performance Charts

Import the Performance module from pyAPP7

[53]:
```
from pyAPP7 import Performance
```

[54]:
```
perf = Performance.PerformanceChart(APP7Directory=APP7DIR)
perf.run(chartpath)
```

[54]:
```
True
```

The result is loaded into a *PerformanceChartResult* instance:

[55]:
```
res = perf.result
```

A *PerformanceChartResult* contains a list of *ResultLine* objects. The *ResultLine* contains the data as a 2d numpy array, with the first dimension being the datapoints and the second dimension the variable index:

[56]:
```
line = res.getLine(0)
data = line.getData()
print data.shape
print data
```

```
(16L, 88L)
[[   0.            0.            0.          ...,   64.27329108
    64.93182676   20.39259343]
 [   0.            0.            0.          ...,   64.27329108   79.2340213
    30.97531047]
 [   0.            0.            0.          ...,   64.27329108
    94.60491582   38.3654778 ]
 ...,
 [   0.            0.            0.          ...,   68.52726956
   288.03991245   26.42931541]
 [   0.            0.            0.          ...,   68.23628197
   306.24737031    3.24777545]
 [   0.            0.           -0.          ...,   69.04558153
   315.66211611  -69.76337364]]
```

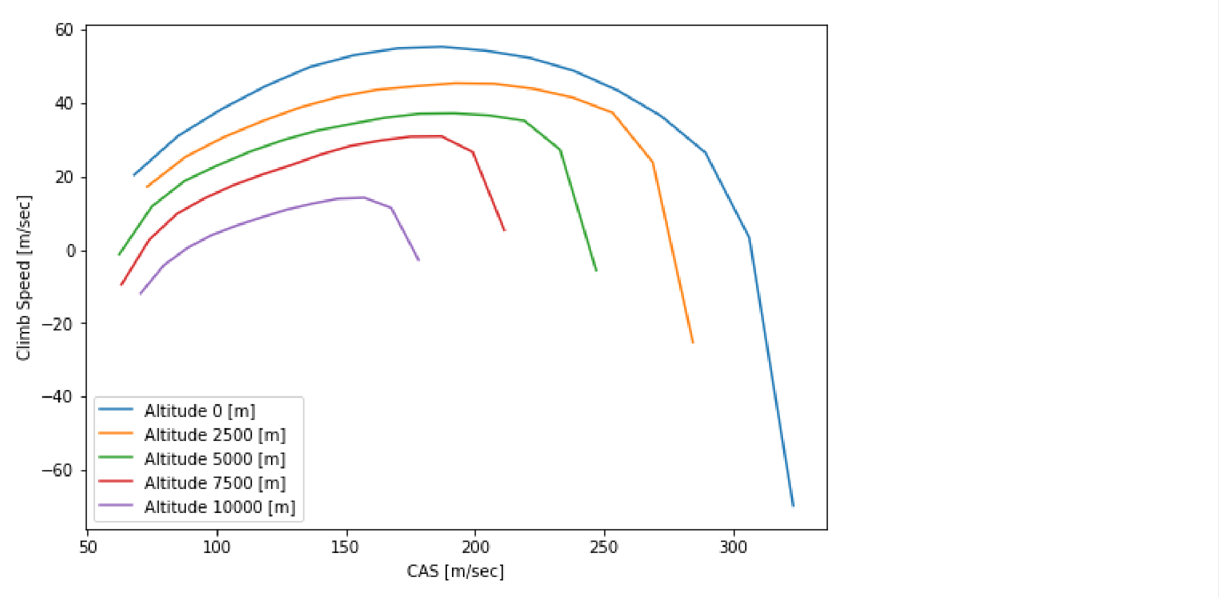
To find the index of the desired variable, use the *getVariableIndex* function:

[57]:
```
idx1 = res.getVariableIndex('CAS')
idx2 = res.getVariableIndex('Climb Speed')
```

The lines can then be plotted using Matploltib:

[58]:
```
fig = plt.figure(figsize=(8.3, 5.8)) #A5 landscape figure, size is in inches
ax = plt.subplot(1,1,1)

#Plot the lines
for line in res.getLineList():
    ax.plot(line.getData()[:,idx1],line.getData()[:,idx2], label=line.getLabel())

ax.legend(loc=3)
ax.set_xlabel(res.getVariableName(idx1))
ax.set_ylabel(res.getVariableName(idx2))
```

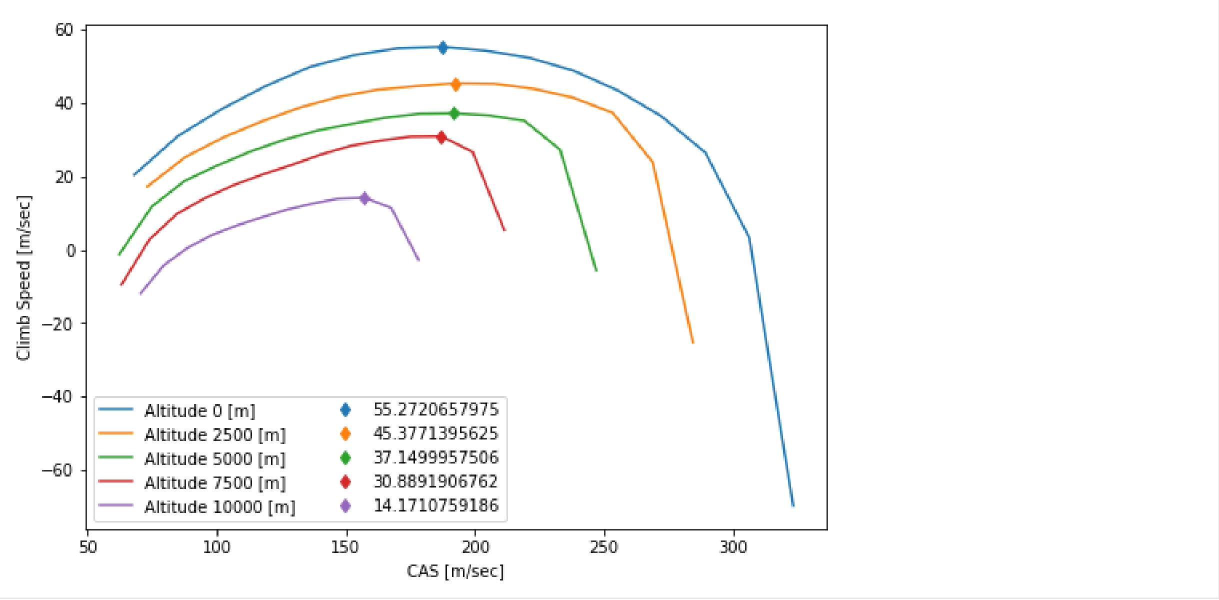
[58]:
```
Text(0,0.5,'Climb Speed [m/sec]')
```



Since each data line is a numpy array, data can easily be processed using the powerful functions of numpy. This example extracts the maxima of each line and plots them. **Note:** the line contains NaNs, therefore the function *np.nanargmax* is used to extract the maxima.

[59]:
```
fig = plt.figure(figsize=(8.3, 5.8)) #A5 landscape figure, size is in inches
ax = plt.subplot(1,1,1)

#Plot the lines
for line in res.getLineList():
    ax.plot(line.getData()[:,idx1], line.getData()[:,idx2], label=line.getLabel())

ax.set_prop_cycle(None) #Resets the color cycle

#Plot the maxima
for line in res.getLineList():
    xdata = line.getData()[:,idx1]
    ydata = line.getData()[:,idx2]
    idx_max = np.nanargmax(ydata) #find the location of the maximum
    ax.plot(xdata[idx_max], ydata[idx_max], 'd', label=str(ydata[idx_max]))

ax.legend(loc=3, numpoints=1, ncol=2)
ax.set_xlabel(res.getVariableName(idx1))
ax.set_ylabel(res.getVariableName(idx2))
```

[59]:
```
Text(0,0.5,'Climb Speed [m/sec]')
```



Climb Speed [m/sec]
CAS [m/sec]
Altitude 0 [m]
Altitude 2500 [m]
Altitude 5000 [m]
Altitude 7500 [m]
Altitude 10000 [m]
55.2720657975
45.3771395625
37.1499957506
30.8891906762
14.1710759186

8 Developer Interface

This part of the documentation details the classes and functions available within pyAPP7

8.1 AircraftModel

class pyAPP7.Files.**AircraftModel**

Holds the APP7 aircraft model that is used to read and write APP .acft files

Each type of data (Mass&Limits, Aerodynamcis, Propulsion, Stores) is stored in two lists: one list containing names and one list containing data. These two lists have to have the same length. The configurations are built by using these list indices. Take proper care when manipulating these lists manually and update the 'ProjectAircraft' (m_Prj).

Examples

The best way to create an instance of an AircraftModel is to use the classmethod fromFile:

```
from pyAPP7 import Files

acft = Files.AircraftModel.fromFile(r'myAircraft.acft')
```

Variables

- **m_GeneralData** (GeneralData) – General data about the aircraft
- **text** (Text) – Content of the comment text box in 'General Data'
- **configName** (*list[str]*) – list holding the names of the Mass&Limits datasets ('Config' classes)
- **aeroName** (*list[str]*) – list holding the names of the Aerodynamics datasets ('Aero' classes)
- **propulsionName** (*list[str]*) – list holding the names of the Propulsion datasets ('PropulsionData' child classes)
- **storeName** (*list[str]*) – list holding the names of the Store datasets ('Store' classes)
- **m_config** (*list[Config]*) – list of the Mass&Limits datasets ('Config' classes)
- **m_aero** (*list[Aero]*) – list of the Aerodynamics datasets ('Aero' classes)
- **m_propulsion** (*list[PropulsionData]*) – list of the Propulsion datasets ('PropulsionData' child classes)
- **m_store** (*list[Store]*) – list of the Store datasets ('Store' classes)
- **m_Prj** (ProjectAircraft) – Contains the Configurations and Store Configurations

classmethod fromFile(*filename*)

Creates a new AircraftModel instance from the path 'filename'

Raises

- **ValueError** – If a parsing error occurs. The aircraft file seems to be corrupted
- **IOError** – If the file cannot be opened

load(*f*)

load an aircraft from a file handle f. Low level function, use fromFile or loadFromFile.

Raises ValueError – If a parsing error occurs. The aircraft file seems to be corrupted

loadFromFile (*filename*)
load an aircraft from a file path 'filename'

Raises

- **ValueError** – If a parsing error occurs. The aircraft file seems to be corrupted
- **IOError** – If the file cannot be opened

saveToFile (*filename*, *overwrite=False*)
Write the APP .acft aircraft file.

Raises ValueError – If file exists but overwrite was set to False

class pyAPP7.Files.**GeneralData**
Class used in 'AircraftModel' to store general data.

Variables

- **m_sAircraftName** (*str*) – Name of the aircraft model ('Model' field in APP)
- **m_sManufacturer** (*str*) – Name of the manufacturer
- **m_sVariant** (*str*) – Name of a specific variant for this aircraft
- **m_sYear** (*str*) – Year
- **m_sAuthor** (*str*) – Name of the author of the APP model
- **m_sVersion** (*str*) – Version description of the APP model
- **m_sDate** (*str*) – Date of the APP model. Format: DD/MM/YYYY (e.g. 24/03/2016)

class pyAPP7.Files.**Config**
Class used in 'AircraftModel', holds Mass&Limits data

Variables

- **text** (Text) – Description
- **mass** (Mass) – Class holding mass data
- **battery** (Battery) – Class holding battery data
- **gear** (Gear) – Class holding gear data
- **tolParameter** (TOLParameter) – Class holding parameters for take-off and landing
- **nEngines** (NExtReal) – Number of engines
- **thrustMult** (NExtReal) – Thrust multiplier
- **fuelFlowMult** (NExtReal) – Fuel flow multiplier
- **relAoA** (NExtReal) – Thrust line angle
- **dDragArea** (NExtReal) – Delta drag area
- **dragMult** (NExtReal) – Drag multiplier
- **posLimitLF** (NExtReal) – Positive limit load factor
- **negLimitLF** (NExtReal) – Negative limit load factor
- **limitAoAMax** (NExtReal) – Maximum AoA Limit
- **limitAoAMin** (NExtReal) – Minimum AoA Limit
- **limitMass** (NExtReal) – Maximum Take-Off Mass limiter (optional)
- **limitMachTable** (X1Table) – Mach limiter table (altitide, Mach)
- **limitAoAGTable** (X1Table) – AoA-G limiter table (AoA, g)

class pyAPP7.Files.**Mass**

Class used in the 'Config' (Mass&Limits) class for the 'AircraftModel'

This class holds the mass breakdown. A minimal dataset should have values for the structure, payload and internalFuel entries.

Variables

- **structure** (NExtReal) – Structure mass
- **propulsionGroup** (NExtReal) – Propulsion group mass
- **equipment** (NExtReal) – Equipment mass
- **massDeviations** (NExtReal) – Mass deviation
- **fixedOperatingEquipment** (NExtReal) – Fixed op. equipment mass
- **unusableFuelAndOil** (NExtReal) – Unusable fuel and oil mass
- **gun** (NExtReal) – Gun mass
- **removableOperatingEquipment** (NExtReal) – Removable op. equipment mass
- **usableOil** (NExtReal) – Usable oil mass
- **crew** (NExtReal) – Crew mass
- **specMissionEquipment** (NExtReal) – Spec. mission euqipment mass
- **ammunition** (NExtReal) – Ammunition mass
- **payload** (NExtReal) – Payload mass
- **internalFuel** (NExtReal) – Fuel mass (internal fuel)

class pyAPP7.Files.**Battery**

Class used in the 'Config' (Mass&Limits) class for the 'AircraftModel'

This class holds the battery properties.

Variables

- **batteryEnergy** (NExtReal) – Energy storage capacity
- **batterySpecificEnergy** (NExtReal) – Specific energy
- **nu_discharge** (NExtReal) – Discharge efficiency
- **nu_charge** (NExtReal) – Charging efficiency

class pyAPP7.Files.**Gear**

Class used in the 'Config' (Mass&Limits) class for the 'AircraftModel'

Variables

- **cdGearArea** (NExtReal) – Gear drag area
- **aoaGround** (NExtReal) – AoA on Ground
- **isFixedGear** (Boolean) – Fixed gear

class pyAPP7.Files.**TOLParameter**

Class used in the 'Config' (Mass&Limits) class for the 'AircraftModel'

Variables

- **tailstrikeAngle** (NExtReal) – Tailsrike angle
- **maxTireSpeed** (NExtReal) – currently unused

class pyAPP7.Files.**Aero**

Class used in 'AircraftModel', holds aerodynamics data

Variables

- **text** (Text) – Description
- **aspectRatio** (NExtReal) – Aspect ratio
- **Sref** (NExtReal) – Reference area
- **cd0Table** (X2Table) – Table holding the zero lift drag CD0
- **cdITable** (X2Table) – Table holding the induced drag CDI
- **clmaxTable** (X1Table) – Table holding the maximum CL (CLmax)
- **cl0Table** (X1Table) – Table holding the Cl0 (DCL, i.e. CL for minimum drag)
- **clTable** (X2Table) – Table holding the lift curves (CL)

class pyAPP7.Files.**PropulsionData**

Base class for propulsion datasets. Use the class method 'fromIndex' to create child classes.

classmethod fromIndex(*index*)

Creates a PropulsionData child class using the propulsion type (index)

Parameters index (*str*) – The currently available types are 'PROPULSION_JET' and 'PROPULSION_PROP'

class pyAPP7.Files.**JetPropulsionData**

Class used in 'AircraftModel', holds jet propulsion data

Variables

- **m_manufacturer** (*str*) – Manufacturer of the engine
- **m_variant** (*str*) – Variant of the engine
- **nthrustData** (*int*) – Number of thrust characteristics. Equals the length of the thrustData list
- **thrustData** (*list[JetThrust]*) – List containing the thrust characteristics (JetThrust)
- **nfuelData** (*int*) – Number of thrust characteristics. Equals the length of the thrustData list
- **fuelData** (*list[JetFuel]*) – List containing the fuel flow data (JetFuel)
- **m_index** (*str*) – Type of the propulsion, PROPULSION_JET

class pyAPP7.Files.**JetThrust**

Class used in 'JetPropulsionData', holds jet thrust data

Variables

- **name** (*str*) – Name of the dataset
- **text** (Text) – Description
- **maxThrustTable** (X2Table) – Table holding the max. thrust data
- **minThrustTable** (X2Table) – Table holding the min. thrust data
- **fuelFlowFileName** (*str*) – Name of the fuel flow data associated with this thrust dataset

class pyAPP7.Files.**JetFuel**

Class used in 'JetPropulsionData', holds jet fuel flow data

Variables

- **name** (*str*) – Name of the dataset
- **text** (Text) – Description

- **fuelTable** (X3Table) – Table holding the fuel flow data

class pyAPP7.Files.**PropPropulsionData**
Class used in 'AircraftModel', holds propeller propulsion data

class pyAPP7.Files.**PropThrust**
Class used in 'PropPropulsionData', holds propeller and power data

class pyAPP7.Files.**PropFuel**
Class used in 'PropPropulsionData', holds fuel flow data

class pyAPP7.Files.**Propeller**
Class used in 'PropThrust', holds propeller data

class pyAPP7.Files.**ElectricPropulsionData**
Class used in 'AircraftModel', holds electric propeller propulsion data

class pyAPP7.Files.**PropElectricThrust**
Class used in 'PropPropulsionData', holds propeller and power data

class pyAPP7.Files.**GenericElectricPropulsionData**
Class used in 'AircraftModel', holds generic electric propulsion data

class pyAPP7.Files.**GenericElectricThrust**
Class used in 'JetPropulsionData', holds generic electric thrust data

The only difference to JetThrust is the class label.

class pyAPP7.Files.**GenericElectricFuel**
Class used in 'JetPropulsionData', holds generic electric fuel flow data

class pyAPP7.Files.**RangeExtenderPropulsionData**
Class used in 'AircraftModel', holds range extender propulsion data

class pyAPP7.Files.**RangeExtenderThrust**
Class used in 'RangeExtenderPropulsionData', holds range extender and power data

class pyAPP7.Files.**Store**
Class used in 'AircraftModel', holds store data

class pyAPP7.Files.**ProjectAircraft**
Class used in 'AircraftModel', holds the configurations and store configurations

Variables

- **storeConfigName** (*list[str]*) – List of the store configuration names
- **storeConfigList** (*list[StoreDataList]*) – List of the store configurations
- **text** (Text) – Description. Currently unused
- **nrOfProjects** (*int*) – Number of aircraft configurations
- **nrOfStoreSettings** (*int*) – Number of store configurations
- **settingName** (*list[str]*) – List of the aircraft configuration names
- **configName** (*list[str]*) – List of the mass and limit dataset names
- **aeroName** (*list[str]*) – List of the aerodynamic dataset names
- **propulsionName** (*list[str]*) – List of the propulsion dataset names
- **thrustName** (*list[str]*) – List of the thrust rating dataset names

checkSettings()
Check the project for consistency

Raises AssertionError – If any of the lists do not have the same length as the project

class pyAPP7.Files.**StoreDataList**
Holds a list of StoreData. Used in ProjectAircraft and ProjectAircraftSetting.

class pyAPP7.Files.**StoreDataList**

Holds a list of StoreData. Used in ProjectAircraft and ProjectAircraftSetting.

class pyAPP7.Files.**StoreData**

Holds the state of a store. The correpsonding 'Store' data is identified by its name

Variables

- **name** (*string*) – Name of the 'Store' data
- **autodrop** (*int*) – set to 1 if the store should be dropped when empty, 0 otherwise (if the store is a fuel tank)
- **storestate** (*int*) – Indicates if the store is dropped (1) or attached (0)

8.2 MissionComputationFile

class pyAPP7.Files.**MissionComputationFile**

Reads an APP .mis file

This class reads an APP mission computation file. Most of the data is stored in a ProjectAircraftSetting object (aircraft configuration and stores) and a MissionDefinition object (initial conditions, list of segments). When manipulating mission files, consult the source code and documentation of these two classes.

Note: Data for APP's "Parameter Study" computation mode is read as well (into the variationData attribute). However, APP's command line mode does not support this computation type

Examples

The best way to create an instance of a MissionComputationFile is to use the classmethod fromFile:

```
from pyAPP7 import Files

mis = Files.MissionComputationFile.fromFile(r'myMission.mis')
```

Variables

- **text** (Text) – Description text
- **name** (*str*) – name of the mission computation
- **author** (*str*) – name of the author of the mission file
- **aircraftpath** (*str*) – path to the aircraft, either relative (to the location of the mis file) or absolute
- **projectAircraftSetting** (ProjectAircraftSetting) – holds the used configuration of the aircraft and settings of stores
- **misDef** (MissionDefinition) – Holds the initial conditions and the list of segments
- **resData** (ResArrayData) – Holds the computation type (CMP_MISSION or CMP_MISSIONVAR)
- **variationData** (VariationData) – Holds data for the Parameter Study mission computation type

checkAircraftPath()

Check if the aircraft file specified in aircraftpath exists

classmethod fromFile(*filename*)

Creates a new MissionComputationFile instance from the path 'filename'

Raises **IOError** – If the file cannot be opened

getAbsoluteAircraftPath(*misFilePath*)
: If the aircraftpath is relative, this function returns the absolute path with respect to misFilePath

getOptimizerSettings()
: kept for backwards compatability

load(*f*)
: Loads a mis file using an existing open file handle f. To read from a file path, use the fuction loadFromFile or the classmethod fromFile

class pyAPP7.Files.**MissionDefinition**

Class is used in 'MissionComputationFile'. Holds the initial conditions and the list of segments.

Variables

- **initialFd** (FlightData) – Initial conditions of the mission
- **segments** (*list [MissionSegment]*) – List of segments
- **opt** (MisOptData) – Solver settings

class pyAPP7.Files.**MissionSegment**

Used in the class 'MissionDefinition', holds all data that describes a segment

Variables

- **segmentIndex** (*str*) – type of the segment, e.g. 'SEG_TAKEOFF' or 'SEG_CLIMB'. Refer to the documentation for valid strings
- **versionString** (*list [str]*) – class name and version, set by APP7
- **segFd** (FlightData) – parameters of the segment. Not all segments use all data.
- **Timestep** (NExtReal) – timestep of the segment, in seconds
- **endValue1, endValue2** (NExtReal) – Segment stop conditions. See documentation for valid NExtReal.realIdx strings
- **comparatorType1, comparatorType2** (*int*) – Comparator for each segment stop condition. less=0, greater=1
- **increaseX, increaseY, increaseZ** (*int*) – flags for x,y and z integration (the z value is currently unused)
- **specialValue1, specialValue2** (NExtReal) – some segments use additional data. Refer to the documentation
- **specialInteger** (*int*) – some segments use additional data. Refer to the documentation

class pyAPP7.Files.**MisOptData**

Holds mission solver data, used in 'MissionDefinition'.

class pyAPP7.Files.**ProjectAircraftSetting**

Saves the index of the active configuration and store configuration and holds the initial state of the stores within the selected store configuration

Used in 'PerformanceChartFile' and 'MissionComputationFile'

class pyAPP7.Files.**VariationData**

Holds mission variation data, used in 'MissionComputationFile'.

8.3 PerformanceChartFile

class pyAPP7.Files.**PerformanceChartFile**

Reads an APP .perf file

This class reads an APP performance chart file. Most of the data is stored in a ProjectAircraftSetting object (aircraft, configuration and stores), a FlightData object (initial conditions and flight state) and a PointPerfSolver child class object (specific data, related to the type of performance chart).

Note: Not all types of point performance charts can be computed by the APP command line mode. See documentation for valid types.

Examples

The best way to create an instance of a PerformanceChartFile is to use the classmethod fromFile:

```
from pyAPP7 import Files

chart = Files.PerformanceChartFile.fromFile(r'myPerfFile.perf')
```

Variables

- **text** (Text) – Description text
- **name** (*str*) – name of the mission computation
- **author** (*str*) – name of the author of the mission file
- **aircraftpath** (*str*) – path to the aircraft, either relative (to the location of the perf file) or absolute
- **projectAircraftSetting** (ProjectAircraftSetting) – holds the used configuration of the aircraft and settings of stores
- **flightData** (FlightData) – holds the flight state (initial conditions)
- **perf** (PointPerfSolver) – instance of a child class of PointPerfSolver, defines the type of performance chart

classmethod fromFile(*filename*)

Creates a new PerformanceChartFile instance from the path 'filename'

Raises IOError – If the file cannot be opened

getAbsoluteAircraftPath(*perfFilePath*)

If the aircraftpath is relative, this function returns the absolute path with respect to the misFilePath

class pyAPP7.Files.**PointPerfSolver**

Base class for a performance chart (PerformanceChartFile) type. Do not use directly, use the class 'PointPerfHelper' to generate child classes.

class pyAPP7.Files.**PointSolveParaStudy**

'Point Performance Computation' performance chart type, used in 'PerformanceChartFile'

class pyAPP7.Files.**PointSolveLFEnvelope**

'G-Envelope' performance chart type, used in 'PerformanceChartFile'

class pyAPP7.Files.**PointSolveSEPEnvelope**

'SEP-Envelope' performance chart type, used in 'PerformanceChartFile'

class pyAPP7.Files.**PointSolveAccelEnvelope**

'SEP-Envelope (Accel)' performance chart type, used in 'PerformanceChartFile'

class pyAPP7.Files.**PointSolveSEPTurnRate**
‘Turn-Rate Chart (SEP)’ performance chart type, used in ‘PerformanceChartFile’

class pyAPP7.Files.**PointSolveAccelTurnRate**
‘Turn-Rate Chart (SEP)’ performance chart type, used in ‘PerformanceChartFile’

class pyAPP7.Files.**PointSolveAltTurnRate**
‘Turn-Rate Chart (Altitude)’ performance chart type, used in ‘PerformanceChartFile’

class pyAPP7.Files.**PointSolveAltSEP**
‘SEP Chart (Altitude)’ performance chart type, used in ‘PerformanceChartFile’

class pyAPP7.Files.**PointSolveThrustDrag**
‘Thrust and Drag’ performance chart type, used in ‘PerformanceChartFile’

8.4 Common Classes

class pyAPP7.Files.**FlightData**
Holds all data that defines a flight state.

class pyAPP7.Files.**ResArrayData**
Holds data for ranges used in performance charts (‘PointPerfSolver’)

class pyAPP7.Files.**Text**
Multi-line text, used in ‘Description’ fields of APP

Variables **text** (*list[str]*) – lines of the text. An empty line is written with a single ‘%’ character

Example

```
>>> comment=Text()
>>> comment.text=['This is a multi-line comment.','%','This is another line']
>>> comment.writeASCII(sys.stdout)
[OBJECT VERSION]
CText        1
[USER TEXT]
3
This is a multi-line comment.
%
This is another line
```

8.5 Supporting Classes

class pyAPP7.Files.**PointPerfHelper**
Factory class to generate ‘PointPerfSolver’ child classes corresponding to a specified performance chart type.

Valid chart types are:

- CMP_POINT_PERF
- CMP_G_ENVELOPE
- CMP_SEP_ENVELOPE
- CMP_SEP_ENVELOPE_ACCEL
- CMP_TURNRATE_SEP_CHART
- CMP_TURNRATE_ACCEL_CHART
- CMP_TURNRATE_ALT_CHART

- CMP_THRUSTDRAG_CHART
- CMP_SEP_ALT_CHART

Variables **cmpType** (*string*) – type of performance chart. See the class method 'newSolver' for a list of valid types.

classmethod fromType (*cmpType*)
Creates a new 'PointPerfSolver' instance with type cmpType.

Raises

- **NotImplementedError** – If the 'cmpType' has not yet been implemented into pyAPP7
- **ValueError** – If the 'cmpType' is not a valid chart type.

newSolver (*cmpType*)
Returns a child class instance of base type 'PointPerfSolver' by using the attribure 'cmpType'. cmpType is set using SetType().

Raises

- **NotImplementedError** – If the 'cmpType' has not yet been implemented into pyAPP7
- **ValueError** – If the 'cmpType' is not a valid chart type.

8.6 Data Types

class pyAPP7.Datatypes.**NExtReal**
APP datatype that wraps a float and allows to specify a label, type of variable (through an index string) and indicate if the value is a limiter

Variables

- **xx** (*float*) – value of variable
- **label** (*str*) – label of the value, e.g. '[Mach]'
- **realIdx** (*str*) – index (type) of variable, e.g. 'REAL_MACH'
- **limitActive** (*int*) – 0 or 1, depends on whether the variable has an active limit. E.g used for Max. Take-Off Mass

Note: use readASCIILimited and writeASCIILimited if the variable is a limited value.

Examples

When using pyAPP7 to read APP files, usually no direct use of this type is needed. This information is mostly for developers/maintainers. The text format of a simple, non-limited NExtReal looks like this:

```
[Mach]
REAL_MACH
0.985
```

This is parsed using readASCII with the flag full=True. The full flag has to be set to True to read the index string 'REAL_MACH'.

```
>>> val = NExtReal()
>>> f = open('path to text file')
>>> val.readASCII(f, full=True)
```

resulting in the following attributes:

```
val.xx = 0.985
val.realIdx = 'REAL_MACH'
val.label = '[Mach]'
val.limitActive = 0
```

If the text format has no index string,

```
[Mach]
0.985
```

readASCII is called with with the flag full=False:

```
>>> val = NExtReal()
>>> f = open('path to text file')
>>> val.readASCII(f)
```

class pyAPP7.Datatypes.**Boolean**
Wrapper to read/write an APP boolean

8.7 Tables

class pyAPP7.Datatypes.**X0Table**
Holds a 1D table (data range)

Variables

- **data** (*list[str]*) – Table data with table factor and interpolation settings
- **table** (*ndarray*) – numpy array of shape (N,1)
- **label** (*str*) – Header string
- **X0Typ** (*str*) – APP variable type

class pyAPP7.Datatypes.**X1Table**
Holds a 2D table

Variables

- **data** (*list[str]*) – Table data with table factor and interpolation settings
- **table** (*ndarray*) – numpy array of shape (N,2)
- **label** (*str*) – Header string

class pyAPP7.Datatypes.**X2Table**(*embedded=False*)
Holds a list of 2D tables

Variables

- **data** (*list[str]*) – Table data with table factor and interpolation settings
- **table** (*list[ndarray]*) – list of numpy arrays of shape (N,2)
- **value** (*list[float]*) – value of each table
- **label** (*str*) – Header string
- **embedded** (*bool*) – True if table is embedded in an 'X3Table'. Disables reading/writing of header (data and label)

clear()
Remove all elements from the table

getIndex(*value*)
Returns index of table with value "value"

Parameters **value** (*float*) – value of the table

Returns index of table with "value"

Return type int

Raises **IndexError** – If table value is not in the list

insertTable (*value*, *data*)
Insert a new table (value, data) pair

Parameters

- **value** (*float*) – value of table to add
- **data** (*ndarray*) – data table as a numpy array with shape (N,2)

Raises

- **ValueError** – If table with value 'value' already exists
- **ValueError** – If data is not of shape N

remove (*i*)
Remove table of index i

class pyAPP7.Datatypes.**X3Table**
Holds a list of X2Tables.

This class holds a list of X2Tables and a value for each table.

Variables

- **data** (*list[str]*) – Table data with table factor and interpolation settings
- **table** (*list[X2Table]*) – list of X2Table instances
- **value** (*list[float]*) – value of each table
- **label** (*str*) – Header string

clear ()
Remove all elements from the table

insertTable (*value*, *x2Table*)
Insert a new table (value, x2Table) pair

Parameters

- **value** (*float*) – value of table to add
- **x2Table** (X2Table) – X2Table to insert

Raises

- **ValueError** – If table with value 'value' already exists
- **ValueError** – If x2Table is not of type X2Table

remove (*i*)
Remove table of index i

8.8 Mission Computations

class pyAPP7.Mission.**MissionComputation**(*APP7Directory='C:\\Program Files (x86)\\ALR Aerospace\\APP 7 Professional Edition'*)

Class to execute APP and subsequently load the results.

This class is a helper class to execute APP mission computations. After creating an instance of this object, execute the 'run' function. The result will be loaded into the 'result' attribute. 'result' is of type MissionResult, see the documentation of the MissionResult class for further details.

Note: The 'Parameter Study' computation type can not be computed with the APP command line mode.

Examples

This example shows how to run a mission computation and obtain an instance of the mission result:

```
from pyAPP7 import Mission

misCmp = Mission.MissionComputation()
misCmp.run(r'myMission.mis')
result = misCmp.getResult()
```

This example assumes APP is installed in the default directory.

Variables

- **output** (*str*) – Path to the text file with the mission results written by APP
- **result** (MissionResult) – The result of the mission computation, parsed from the 'output' text file
- **misCompFile** (Files.MissionComputationFile) – Instance of a MissionComputationFile (APP .mis file). Is available once the method run was called
- **db** (*Database*) – Instance of a Database object
- **inputfile** (*str*) – Path to the APP mis file

printSegmentNames()

Prints the name of the segments

printStores()

Prints the name of the stores used in the mission

run(*inputfile, imperial=False, suffix='_output', ParameterList='ParameterList_All.par'*)

This method runs APP7 using the command line mode and loads the results.

After the APP7 computation has terminated, the result is read into 'result'.

Parameters

- **inputfile** (*str*) – path to the APP7 .mis file
- **imperial** (*bool, optional*) – set False for SI units, True for imperial units
- **suffix** (*string, optional*) – suffix of the written result text filename
- **ParameterList** (*string, optional*) – filename of the parameter file. Has to be in the pyAPP7 directory.

Returns True if successful, False otherwise.

Return type bool

Raises

- **IOError** – If the mission file (inputfile) does not exists
- **IOError** – If the aircraft file specified in the mission does not exists or if no aircraft path was provided
- **ValueError** – If the computation type of the mission file is not set to 'Single Mission'

class pyAPP7.Mission.**MissionResult**

This class can read the APP mission result text file.

Examples

This example shows how to read a mission result directly from a text file. This is useful to read results from past mission computations, for example when conduction batch simulations:

```
from pyAPP7 import Mission

res = Mission.MissionResult.fromFile(r'myMission.mis_ouput.txt')
```

Variables

- **output** (*dict*) – Dictionary containing the mission flags, error text, number- and list of variables
- **segments** (*list [MissionResultSegment]*) – A list of MissionResultSegment class instances, holding the results of each segment
- **initialSettings** (*MissionResultSegment ()*) – The initial settings of the mission

classmethod fromFile(*filename*)

Creates a new MissionResult instance from the path 'filename'

Raises IOError – If the file cannot be opened

getVariableData(*name*)

Returns an array with all data of the variable starting with the name 'name'

Use np.hstack() on the return value 'mission_data' to get a single array.

Parameters name (*str*) – name of the variable

Returns

- **var_name** (*str*) – Full name of the variable name
- **mission_data** (*ndarray*) – Variable data for all mission segments as a numpy array.

Raises ValueError – If the variable with name 'name' does not exists

getVariableIndex(*name*)

Returns the variable index starting with the name 'name'

Parameters name (*str*) – name of the variable

Raises

- **ValueError** – If multiple variables starting with 'name' exist
- **ValueError** – If the variable with name 'name' does not exists

getVariableList()

Returns an ordered list of the variable names

getVariableName(*idx*)

returns the name of the variable at index idx

class pyAPP7.Mission.**MissionResultSegment**

Class used to store the result of a single mission segment. This class is used in the MissionResult class to parse each segment.

Variables

- **name** (*str*) – Name of the segment
- **data** (*ndarray*) – Data table as a numpy array with shape (ndata,n_var). n_var is stored in the MissionResult.output['n_var']
- **ndata** (*int*) – Number of datapoints in the segment

8.9 Performance Chart Computations

class pyAPP7.Performance.**PerformanceChart**(*APP7Directory='C:\\Program Files (x86)\\ALR Aerospace\\APP 7 Professional Edition'*)

Helper class to execute a performance chart computation from an existing .perf file.

Variables

- **inputfile** (*str*) – Path to the input .perf file
- **perfFile** (PerformanceChartFile) – Parsed input APP7 .perf file
- **output** (*str*) – Path to the resulting txt file
- **result** (PerformanceChartResult) – Result
- **APP7Path** (*str*) – Full path to the APP7 executable

Parameters **APP7Directory** (*str, optional*) – Path to the location of the APP7 executable.

Raises **ValueError** – If the APP7 executable is not found in the specified directory

run(*inputfile, imperial=False, suffix='_output', par_file=None*)

This method runs APP7 using the command line mode and load the results.

After the APP7 computation has terminated, the result is read into 'result'.

Parameters

- **inputfile** (*str*) – path to the APP7 .perf file
- **imperial** (*bool, optional*) – set False for SI units, True for imperial units
- **suffix** (*string, optional*) – suffix of the written result text filename

Returns True if successful, False otherwise.

Return type bool

Raises

- **IOError** – If the performance file (inputfile) does not exists
- **IOError** – If the aircraft file specified in the performance file does not exists or if no aircraft path was provided

class pyAPP7.Performance.**PerformanceChartResult**

Reads a result txt file written by the APP7 command line mode for a performance chart.

Examples

This example shows how to read a performance chart result directly from a text file. This is useful to read results from past computations, for example when conduction batch computations:

```
from pyAPP7 import Performance

res = Performance.PerformanceChartResult.fromFile(r'myChart.perf_ouput.txt')
```

Variables

- **output** (*dict*) – stores the result meta-data
- **lines** (*List [ResultLine]*) – holds the data of each line of a performanc chart

classmethod fromFile (*filename*)
Creates a new PerformanceChartResult instance from the path 'filename'

Raises **IOError** – If the file cannot be opened

getErrorText ()
Returns the error text

getLine (*idx*)
Returns the ResultLine at index idx

getLineData (*idx*)

Parameters **idx** (*int*) – Index of line

Returns numpy array of all data points, with shape (n_points,n_variables)

Return type ndarray

getLineLabelList ()
Returns a list of all line labels

getLineList ()
Returns the list of ResultLines

getLineVariableData (*idx, varIdx*)

Parameters

- **idx** (*int*) – Index of line
- **varIdx** (*int*) – Index of variable

Returns numpy array with data points and shape (n,)

Return type ndarray

getVariableData (*name*)
Returns an array with all data of the variable starting with the name 'name'

Parameters **name** (*str*) – name of the variable

Returns

- **var_name** (*str*) – Full name of the variable name
- **line_data** (*ndarray*) – Variable data for all chart lines as a numpy array.

Raises **ValueError** – If the variable with name 'name' does not exists

getVariableIndex (*name*)
Returns the variable index starting with the name 'name'

Parameters **name** (*str*) – name of the variable

Raises

- **ValueError** – If multiple variables starting with 'name' exist
- **ValueError** – If the variable with name 'name' does not exists

getVariableList()
Returns an ordered list of the variable names

getVariableName(*idx*)
returns the name of the variable at index idx

isSuccessful()
Returns True if the performance result was computed sucessfully

loadFromFile(*filename*)
Read a APP7 performance chart result

Parameters **filename** (*str*) – path to the results txt file written by APP7

class pyAPP7.Performance.**ResultLine**
Represents a line in an APP7 performance chart.

Variables

- **label** (*str*) – Label of the line
- **value** (*str*) – Value of the line
- **ndata** (*int*) – Number of data points of the line
- **data** (*ndarray*) – Data array of all points with shape (ndata,nvariables)

load(*f*)
This function is called by the PerformanceChartResult class, do not use directly. Reads a line from the file handle f.

9 Version History

9.1 1.1 (2020-08-17)

- Database updated for APP 7.0.3.0
- Added capabilities to read and write new KTS/s TR and SEP envelope charts

9.2 1.0 (2019-06-17)

- Initial release, corresponds to APP 7.0.1.0

Index