Architecture about Dancing: Creating a Cross Environment, Cross Domain Framework for Creative Coding Musicians

Owen Green, Pierre Alexandre Tremblay, Ted Moore, James Bradbury, Jacob Hart, Alex Harker
Centre for Research in New Music
University of Huddersfield
{o.green, p.a.tremblay, t.moore, j.bradbury, j.hart2, a.harker}@hud.ac.uk

Gerard Roma
School of Computer Science
Leeds Trinity University
g.roma@leedstrinity.ac.uk

Abstract

In this paper, we offer reflections on the (still-forming) outcomes of a five-year project, situated in a context of artistic research around music technology, that seeks to facilitate the use of machine listening and machine learning techniques for creative coding musicians working in the Max, Pure Data and Supercollider environments. We have developed a suite of software extensions and learning materials, and, unusually, we have included community development efforts in our work. Whilst the project has no doubt differed in aims, methods and knowledge claims to how PPIG researchers may approach these topics, we feel there is significant common interest in a number of the emerging themes. We focus here on continuing attempts, by user-programmers and library programmers alike, to navigate various tensions thrown up by ambitions for the project's *accessibility*, *community* and *continuity*. Among these tensions are: cross environment legibility *vs* cross domain legibility between music and data-science *vs* environment idioms *vs* (unknown) idiosyncratic working patterns *vs* quick iterative design *vs* maintainability and longevity *vs* low cost of entry *vs* penetrability (Clarke & Becker, 2003).

1. Introduction

This paper offers some reflections from the late stages of a five-year project, *Fluid Corpus Manipulation* (FluCoMa), that has sought to make a toolkit of signal processing and data science extensions available to creative coding musicians who work in the Max (Zicarelli, 2002; Puckette, 2002), Pure Data (Puckette, 1997) or Supercollider (McCartney, 2002) languages. Our goal here is to bring into focus how a range of different priorities interacted to produce some specific design choices, and some consequences these priorities have had for the usability of the toolkit, based on the experiences of users.

We will briefly introduce the project and its disciplinary background. We place our efforts in the context of some earlier work in this area that serves to help explain how the priorities that steered the design and implementation process took root. We then present four vignettes describing aspects of the toolkit that continue to present difficulties to users, and point towards how our design priorities played a part in producing these issues. A takeaway is that whilst these priorities have served us well in getting this far, they may be coming to the end of their usefulness, both for dealing with remaining usability issues, and as the project transitions (we hope) into community development.

2. Background

We will sketch out the background context to the FluCoMa project here to the extent that we can illustrate how a particular set of design priorities came to inform development. For a fuller account of the toolkit's foundations and eventual form, interested readers can consult Green, Tremblay, and Roma (2018) and Tremblay, Roma, and Green (2022).

Whilst the bulk of computer based music making still happens in the context of digital audio workstations (DAWs) that mimic the functionality of mixing consoles and multitrack recorders, a long-running thread of creative work and research has focused on environments that are more open-ended in how they allow musicians and other artists to work with sound. As computing power has increased, musical

languages have augmented symbolic data processing capabilities with ways of working directly with audio, and with that has come an interest in getting and working with data *from* audio, ranging from very simple analyses (like tracking the energy of a signal) to the much more complex (like 'un-mixing' the voices of a polyphonic sound). Meanwhile, as machine learning techniques have become more tractable and accessible, there has been interest in how these can be used to help facilitate or organise some of this work.

Two particular strands of prior work in machine listening and learning for music had a formative influence on the types of task we wished to enable and explore with our toolkit. The first of these, pioneered by Rebecca Fiebrink's *Wekinator*, uses simple machine learning models as an alternative paradigm for programming *mappings* between input data and controls for synthesisers or other processes (Fiebrink, 2019). Crafting such mappings plays a very large role in the programming of interactive and generative music systems. Doing this work by hand can be tedious and unintuitive, as well as frequently producing brittle results, especially when using data derived from audio. The second strand, exemplified by Diemo Schwarz's *CataRT* (Schwarz, Beller, Verbrugghe, & Britton, 2006) uses audio feature analysis as a tool to discover and play with relationships between sounds in a corpus. Typically this involves an interface that uses a 2D scatter plot of segments of sound, arranged according to some audio descriptors or, in more recent manifestations, according to some dimensionality reduction process (Roma, Green, & Tremblay, 2019; Garber, Ciccola, & Amusategui, 2020)[1].

The currently dominant languages for creative-coding musicians—Max, Pure Data and SuperCollider—have a shared emphasis on working in real-time, making them useful not only for composing, but also for building custom instruments, installations or even co-players. All are also able to host extensions via plugins written in C or C++. Despite this common focus on real-time work, they are very different environments (see Nash, 2015, for a fuller comparison). Max and Pure Data are superficially similar data-flow-like languages that share the same inventor. A 'box-and-arrow' type patching idiom promotes a focus on processes rather than data, and the facilities for manipulating data structures are quite limited (though different) in each. However, Max and Pure Data have diverged somewhat from their common roots, both in terms of what is possible or easy to achieve in each, but also in terms of what is idiomatic to each.

SuperCollider, meanwhile, is a very different environment with a completely different architecture where the language ('sclang') runs in a separate process to real-time audio processing (the server), leading to very different usage experiences and constraints. sclang is a textual, dynamically-typed language, with some similarities to Smalltalk, and sophisticated facilities for manipulating data and expressing the timing and patterning of musical data. It communicates with the server (which actually does the sound production) via a network protocol (using UDP by default). The server is primarily focused on the playback of graphs of sound generators that have been specified in the client language, although there are some limited facilities for adding new 'commands' for offline processing. Facilities for communicating back from the server to the language are limited and awkward. Whilst there is a C-based SDK for Supercollider, this applies only to the server (for writing new sound generators or commands), and not to the language, so that all FluCoMa components need supporting scaffolding in sclang.

Whilst the built-in support for analysing and working with audio data has been quite limited in these environments, there have been various extensions and packages available for each language, such as the *MuBu and Friends* package for Max (Schnell et al., 2009). An impetus for the FluCoMa project was that, despite much promise, existing extensions in this area left a number of things to be desired from the point of view of supporting widespread uptake and long-term artistic research, such as:

1. Imposing a whole new language to learn on top of the host environment.

[1]For an impression of how our toolkit looks and works in practice, a video tutorial showing how to construct a CataRT-style exploration interface in Max can be found at https://learn.flucoma.org/learn/2d-corpus-explorer/. To compare the toolkit's feel in different languages, these videos demonstrate equivalent examples in Max (https://youtu.be/cjk9oHw7PQg) and SuperCollider (https://youtu.be/Y1cHmtbQPSk)

2. Having only sparse or expert-level documentation, whilst also introducing many new concepts.

3. Not releasing source code and / or being unmaintained after a short time.

4. Only being available for a single creative coding environment, inhibiting portability and cross-communication.

From this background, the FluCoMa project had three overarching thematic preoccupations that would crystallise into a set of design priorities as work progressed:

Accessibility Whilst recognising that we would be addressing a reasonably advanced subset of creative coding musicians with this toolkit, there is no particular reason to believe that being an expert user of one of these environments translates into an appetite for finding one's own documentation or for learning more new languages. Furthermore, we knew from experience that such appetite can vary quite markedly depending on where in the progress of a creative project one is.

Community A key rationale of the project is that it should deliver enabling conditions for more and better artistic research around data-driven music making, meaning that we are concerned not just with augmenting the possibilities of our own artistic practices but with establishing a community of interest around this topic. It was crucial for us that this community forming took place *alongside* technical work. This is because we were conscious that developing in isolation could shape the affordances of the toolkit too strongly around our own musical proclivities or ways of thinking, and result in more exclusive and less interesting work. Moreover, evaluating artistic research *requires* community: it is by nature discursive and qualitative, and poorly served by proxy measures.

Continuity Artistic research takes time: pieces take a long while to complete, instruments require many hours of practice and performance, and typically research questions and concrete musical goals solidify and emerge over the course of practical investigation rather than being clear at the outset of a project. The timescales involved will very often be far in excess of funding periods, so artistic researchers are justifiably nervous of tools that may stop working, or simply disappear.

These themes played a structuring role in the overall approach taken to development, as well as coming to inform (more or less explicitly) design choices made along the way. A core commitment of the project, serving both accessibility and community, was that the toolkit should target more than just a single host language, the better to reach a wide range of practitioners and benefit from the distinct ways of thinking about music and code that working in different languages might bring (McPherson & Tahıroğlu, 2020).

2.1. Project Phases

The early phase of the project was structured by two waves of professional commissions with public performances, ensuring that we would have some committed input from fellow experts prepared to endure using software in a state of flux whilst pushing some work through to a developed state over the course of some months[2]. As the toolkit matured towards public release, the later stages of the project have been marked by wider participation, both via our forum at `https://discourse.flucoma.org/`, and over 30 workshops delivered to a mixture of researchers, students and independent artists, mostly in Europe and North America.

Expanding our user pool in this later phase has also allowed us to dramatically improve our documentation based on being able to observe and discuss where people experienced problems. As well as platform-specific help files and reference material, we developed a web site of learning resources at `https://learn.flucoma.org/`, focussing on concepts and usage patterns rather than the specifics of particular components.

[2]Documentation of these commissions can be found at `https://www.flucoma.org/commissions/` along with some deeper analysis amongst the articles at `https://learn.flucoma.org/explore/`

2.2. Design Priorities

Certain design priorities fell out of this project structure quite naturally:

Rapid Development To make sure that we could be responsive to our commissioned artists' experiences with early versions of the tools, we tried to arrange matters to produce and trial new components rapidly, across each of our host environments. To support this way of working we made heavy use of C++ templates to establish a host-agnostic way to specify the form and behaviour of a component (a 'client') that would yield a well-formed extension in each of Max, Pure Data or SuperCollider (see Figure 1).

Maintainable Code Other priorities, meanwhile, needed to be observed *despite* this fast-paced, iterative way of working. Trying to ensure continuity for the project, for instance, has meant always bearing in mind what impact on future maintenance adding features might have, especially considering that we would ideally like this maintenance to be a communal affair in the future.

Idiomaticity A key feature of accessibility is whether an API respects established idioms for the language it targets, which may well not be enforced by the language, but reflect common working practices that in turn serve to support sharing and discussing code. At the same time, people pick up these particular languages in quite distinct ways and there may well be multiple idioms emerging from different enclaves or traditions. Furthermore, we are addressing artists, many of whom might well have playful and idiosyncratic ways of working, which we would not wish to hinder.

Legibility In the interests of accessibility and community, we have tried to ensure that both the functionality and form of the APIs provided are consistent between each of our target languages, the better to enable communication and comparison of practices, so that code using our toolkit in one language should be legible to users of another language. We are also, of course, trying to make legible concepts from outside the musical domain, imported from signal processing and data science. Both of these exist in tension with *idiomaticity*.

Complexity Accessibility is also served by trying to ensure that the cost of participation is low: it should be possible to do interesting things with a few components, with little tweaking and not too much recourse to documentation. Meanwhile, however, both community and continuity require that there is scope to do new and powerful things, and that more complex experimentation is both feasible and rewarding. The process of bringing a creative project to fruition always involves moving between different types of programming practices, and it is important that we can reward 'sketching' and 'tinkering' whilst also supporting more orthodox programming, as one 'toughens' a patch ready for performance or distribution (Bergström & Blackwell, 2016).

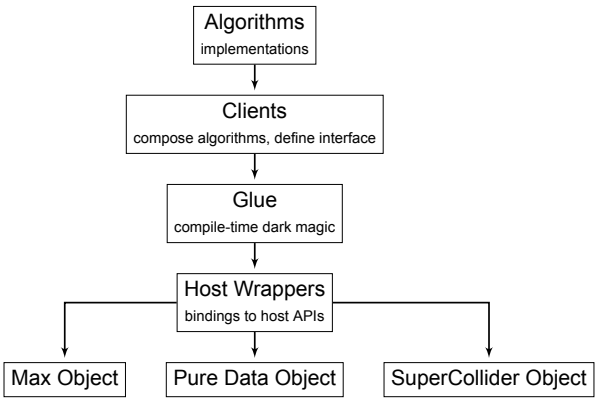

Figure 1 – The Supporting C++ Architecture

Clearly, it would be impossible to satisfy all these principles all the time because they pull against each other. Furthermore, we have to operate within the engineering constraints of what are actually pretty different languages and supporting platforms. This is especially true with respect to SuperCollider's distinctive architecture: because it is only possible to write C++ extensions for the server side, coming up with ways of working that satisfy both a sense of what is locally idiomatic and recognisably 'FluCoMa' is a delicate balancing act.

These tensions notwithstanding, we are reasonably happy that an approach of pragmatically balancing between these factors has served us well so far. The toolkit and its associated resources have been received well: people are using it in their work unprompted, and contributions of discussion, feedback, and code from new people are coming in. Nevertheless, the ways in which we have been weighing between these factors will probably have to change. The following section details four areas where we have noticed that people experience particular difficulty in using the tools (this is not an exhaustive list, mind you), and we reflect upon how these points of friction materialised.

3. Four Vignettes of Difficulty

We will present here four brief vignettes examining aspects that users have found it hard to learn or that are cumbersome in practice. We discuss how some of our original design decisions brought these about, and how / why it is hard to arrive at solutions that continue to balance the principles described in Section 2.2.

3.1. Components With Many Parameters

Dealing with an abundance of parameters is a routine problem in designing an API in any language. Many of the algorithms we have implemented in the toolkit have a great many and, when in doubt, we have often opted to expose things rather than hide them on the basis that any encapsulation can prematurely foreclose possibilities for creative experimentation. However, this can cause problems for users in knowing what might be most useful to reach for at first, or in grasping how different parameters might be related. In addition, parameters are exposed to users quite differently across our three environments, and the available mechanism to help users make sense of large sets of these things also vary (see Figure 2).

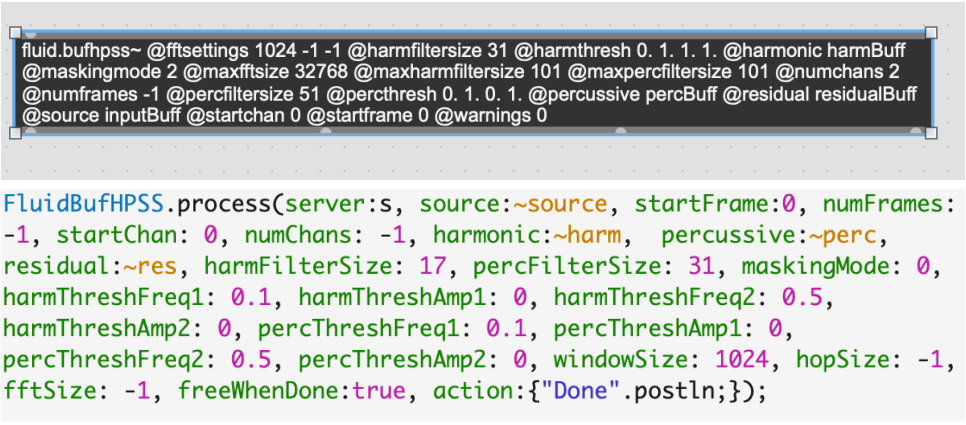

Figure 2 – Views of the same component with many parameters in Max, SuperCollider and Pure Data.

In principle, there are different tactics we could investigate to alleviate this. However, because of the varied ways the environments expose parameters, these tactics push against our desire to keep things consistent, and reduce maintainability by demanding environment specific approaches. For instance, we could try to group together parameters so that the 'primary' and 'tweaky' controls for an algorithm are clearly separated, or so that parameters addressing a given aspect are encapsulated together. In SuperCollider this could be done by bundling parameters into new sclang classes that signal their relatedness. In Max and Pure Data the mechanism isn't so clear. An alternative would be to encapsulate whole components into simpler interfaces for newcomers, and expose fewer moving parts. This could be done through abstractions in Max / PD and wrapper classes in SuperCollider, but at the cost of a great deal of duplication across environments that becomes hard to maintain as the number of components increases. A more tenable solution would involve returning to our C++ APIs and working on making it possible to compose 'clients' at this level.

3.2. Buffers as Universal Containers

Early on in development we realised that as well as having real-time processes, we needed processes that worked offline on stored pieces of audio data. Some algorithms only work this way (because they are not causal), but such a facility also enables one to process data in batches, often faster than real-time. This raised a question about what the output of such offline processors should be contained in when the result wasn't audio data (e.g. some kind of analysed feature like pitch or loudness). We settled on using the same containers that the host environments use to store audio, variously called 'buffers' (Max, SuperCollider) or 'arrays' (Pure Data). It seemed a straightforward decision: all three environments had such a component with which users would already be familiar, and they were also scalable up to very large sizes (unlike the 'list' types in Max and Pure Data).

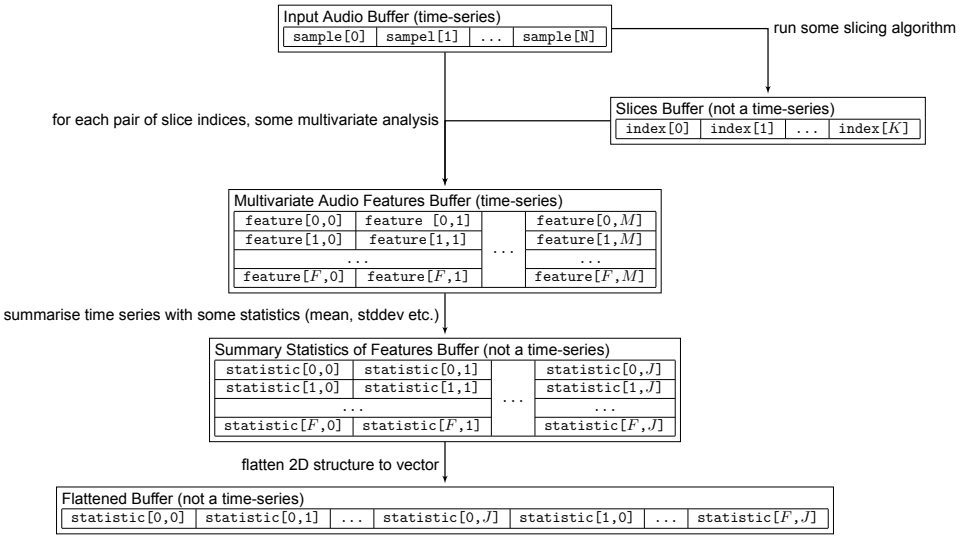

Figure 3 – Various different structures we impose on the same generic container, at different points in an audio analysis workflow

However, when we also encountered the need to output things that weren't simple time series, we stuck with buffers, primarily in the interests of being able to keep moving. This has contributed to a number of difficulties we see people having:

- The very fact of using buffers 'off-label' by putting non-audio data into them causes trouble for some users, though fortunately seldom long-lasting. Clearly, we violate a principle of least astonishment at some level for these users.
- More stubborn difficulties arise when people need to manipulate data in these buffer objects. These are two-fold. First is that reshaping operations, such as flattening a two-dimensional matrix

of values to a vector, can be hard to conceptualise in the absence of Matlab / NumPy-like facilities for inspecting the effects of such moves.

- All this is exacerbated by the extent to which we have 'overloaded' the usage of buffers to also contain things that either aren't time series (vectors of statistics or slice points), or try to express more dimensions than two in what is an inherently 2D structure.
- Slice point buffers are especially vexing, because there's not a nice way to deal with them across platforms. Typically, we wish to iterate over pairs of points to analyse sections of an audio buffer, and this currently always pushes error-prone boilerplate on to the user: iteration in Max and Pure Data is famously awkward. Meanwhile, in SuperCollider, the same iteration involves bringing the buffer back to the language to be able to iterate over it, and then launching a stream of processes back on the server. Because this is all asynchronous, it is all too easy to end up with code full of nested callbacks that are hard to reason about.

A big part of the problem here is that the environments' native buffer / array components don't expose much direct machinery for doing things beyond playing back audio. In Max and Pure Data we can copy into a native 'list', which allows for more direct manipulation, but is limited in both size and representative power (flat and one-dimensional only). In SuperCollider, we can talk to Buffers in the context of Synths on the server (which is awkward), or stream them back to the language (which is slow *and* awkward).

3.3. Iterative Workflows in Machine Learning

An important set of components in our toolkit provides building blocks for some basic machine learning tasks, such as supervised and unsupervised learning, and some data pre-processing. Figure 4 shows an example pipeline that combines some of these techniques. The shape of our API is heavily based on scikit-learn (Buitinck et al., 2013), a machine learning toolkit for Python, which gave us something solid and proven to go on quickly, and meant that there was a useful documentation resource available whilst our own learning resources caught up.

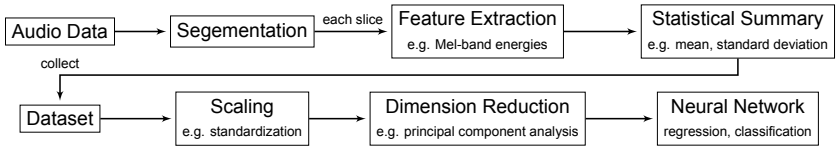

Figure 4 – An example machine learning pipeline

Whilst this building-block approach works well for being able to quickly gather data and assemble chains of algorithms, it has become apparent that it doesn't really communicate a core expectation of machine learning development practices: the process of building up a useful model is always *iterative*, both in the senses of tweaking or perhaps swapping out each stage of a pipeline, but also that one will return frequently to one's data to experiment with its composition and analysis. In this sense there is a moderately normative aspect of the practice that our API wishes to support that is left implicit by the interface, and the costs of it not being apparent might be that newcomers simply get discouraged by poor initial results.

Our target environments contribute different sorts of difficulty to this sort of iterative experimentation, which suggests that solutions that maintain a consistent API might not be forthcoming. In the patching languages (Max and Pure Data), assembling a pipeline is easy enough, but repeatedly re-assembling or re-organising can be a drag as things tend to get messy, and there are a great many mouse clicks involved. In SuperCollider, by contrast, sclang makes it much easier to reorganise things and also has more powerful tools for manipulating data structures. However, the data and algorithms are all on the server, meaning that people can end up having to manage a great deal more traffic between language and server than is desirable.

For many musical users who are used to working in digital audio workstations or synthesisers, the basic idea of such an iterative approach to finding a sweet spot would be familiar enough from the practice of designing sounds through chains of audio generators and processors. However, what is conspicuously different here is the immediacy of feedback and, crucially, the ease with one can make sense of it. This leads us on to the issue of how one can evaluate results in machine learning pipelines and the question of interpretation.

3.4. Making Sense: Evaluation and Interpretability

Sometimes all that is needed to verify that one is happy with how a model is performing in a musical context is to play with it and decide if one likes the results. However, if one isn't happy, then it can be hard to know what to do about it. So far, we haven't built in typical supporting machinery for model evaluation, such as mechanisms for performing cross-validation or comparing performance on training data to performance on held-out test data. In part this is because of the usage and implementation complexity they would add, but also because the results of these processes still require careful interpretation. Where this is most evident is with our neural network components, where we have noticed that new users can become overly focused on isolated values for the training error, which signals—at the very least—that we need to better document what limited sense can be made from this number.

More generally, we notice that people struggle with the increasing abstraction away from perceptually explicable quantities as they move through pipelines like that shown in Figure 4. Supervised processes, like neural networks, are hard to look 'into', whereas the output of unsupervised algorithms like dimension reduction can be hard (or impossible) to interpret in sonic terms. Even certain audio analysis features can be difficult to relate directly to aural experience, such as Mel frequency cepstral coefficients (MFCCs), which are commonly used but rarely well explained[3].

Because all the components of such pipelines interact, difficulties in interpretation become compounded and innocuous-seeming steps, like whether or how to normalise the ranges of input data, have presented obstacles to some users. The 'correct' choice for such a step cannot be given *a priori* because it depends both on the condition of the input data and often also on the assumptions of algorithms further downstream. For instance, any algorithm that involves a distance calculation is likely to do better if all input dimensions are uniformly scaled, and other algorithms may also depend on the data being centred. Using the sample mean and standard deviation for this centring and scaling is very much standard practice in many machine learning workflows, but whether it actually makes sense to use these statistics depends on the data itself. What if the distribution is not Gaussian, for instance? Again, we see implicit conditions on what effective usage patterns might be that are not directly expressed by the API, and whose explanations can get very technical quite quickly.

Established and emerging mechanisms for evaluating and interpreting machine learning models tend to be numerical or visual. So far, we've hesitated to introduce more than quite minimal visual facilities into the toolkit because, besides being hard to program well, they are non-portable between our hosts and therefore represent a real maintenance headache.

4. Discussion

With these four vignettes our intention has been to develop a critical assessment of our toolkit from a user perspective, and to illustrate some concrete challenges that arose from trying to balance contradictory priorities over the course of development. In ending up having components with a great many parameters, presented homogeneously, we opted to err on the side of exposing more moveable parts in the interests of not imprinting the API with whatever we happened to think was musically important, and to leave room for experimentation. Ways of mitigating this that also preserve consistency between the APIs exposed in different hosts have so far eluded us. Opting to use our hosts' audio buffer components in a variety of ways unintended by their original authors served the interests of rapid development and of accessibility, insofar as users didn't have to learn a whole new container, but nevertheless is a persistent

[3] We have had a go: https://learn.flucoma.org/reference/mfcc/.

source of confusion for new users that, again, resists easy fixes that would also preserve the consistency of experience between environments.

Meanwhile, taking a building block approach to making some data science facilities available was motivated by wishing to balance immediate usability against scope for deeper experimentation, whilst not foreclosing unforeseen musical outcomes by making advance judgements about what is important or desirable. What we have ended up with resembles quite closely standard machine learning toolkits like Python's scikit-learn. On the plus side, this provides a proven basis of an effective API, and a smooth route for users wishing to experiment in a different setting. However, there are aspects of common practice that remain implicit in our implementation, and this can be confusing. Both dimensions of this that we described—the expectation of an iterative workflow and the difficulties in interpreting what's happening—point not just at the possible fruitfulness of offering some alternative abstractions but, more nebulously, at a need to try and make the statistical languages and practices of data science more legible for musicians, perhaps even aurally.

While none of these challenges are insurmountable, addressing them may well mean relaxing adherence to our guiding principles. In particular, striving to maintain consistency of both functionality and form across three quite different environments finds itself in increasing tension with the ambition that the tools should slot neatly into the idioms of each host, and that they should be accessible both to less experienced (or, indeed, less patient) users whilst also offering a rich palette for experimentation. Of course, a further difficulty with this ambition for cross-host consistency is that individual users may well not care: if something's difficult to use, it's possibly not of much comfort to learn that this difficulty stems from a lofty principle, however well-intentioned.

Furthermore, as the project ends its initial phase of development and moves into what we hope will be a more communal mode of maintenance and enhancement, it seems clear that the balance of priorities and principles would have to shift in any case. The idea of cross-host consistency has been possible to try and stick to as long as we were a small group of developers working full-time. Insisting that any and all future community contributions place the same level of focus on this is unrealistic, especially if we don't wish to deter people from getting involved. Similarly, our ideas of what is and is not maintainable will probably shift. At present, these estimations are very much conditioned by the very fact that we are so few people, as well as what kinds of change the existing C++ framework makes more or less difficult, itself a function of how we have balanced priorities to date. One especially pressing body of work left to address *before* the project moves into community development is the maintainability of the C++ code itself, parts of which are suffering from repeated duct-tape solutions and over-obscurity, generally made in the interests of trying to keep development iterations reasonably rapid. Crucially, just as we have tried to create scope for progressively deeper engagement with the toolkit, we want something similar for potential contributors of new algorithms and components in C++, in the form of a entry-level API that tries to minimise the barriers to contribution.

If we are to be optimistic (and why not?), we may well hope that a wider pool of contributors will be enabling in other ways. For instance, in this initial phase we have been wary of encapsulating *too* much for fear that we would imprint our own, limited musical priorities on the toolkit too strongly. However, a more distributed mode of development opens up more scope for a variety of different encapsulations and ways-in that can address different musical practices and technical proclivities. In this way, we might also hope that some of the problems of legibility pointed at in sections 3.3 and 3.4 could be tackled in varied ways. If one dimension of the problems described is that there are implicit workflow expectations that aren't clearly illustrated by having all these separate building blocks, another is that the tools we have so far borrowed from data science remain generic and that we don't yet have a vocabulary for articulating what is characteristic of different types of musical data, or for relating our aural experiences of these data to statistical explanations.

5. Conclusion

Over the course of five years of research and development, the Fluid Corpus Manipulation project has put together a toolkit for creative coding musicians that consolidates and builds upon a range of prior work that provides functionality for audio decomposition, analysis and basic machine learning workflows in Max, Pure Data and SuperCollider. The toolkit has been successfully used to produce a number of complete pieces, has been enthusiastically received by user communities, and successfully taught to newcomers in over 30 workshops, which gives us some confidence that our ambitions for the learnability, utility, performance and robustness of the tools are being fulfilled. This has been achieved in part by following a pattern of (relatively) rapid, iterative software development, combined with efforts to produce expansive learning resources, and to bring forth a community of interest around the project's topic.

We have highlighted some wrinkles in usability, and explored how these have their roots in the interaction between some principles that guided development. In particular, we have tried to balance the accessibility and flexibility of the tools, whilst also making efforts to keep a sense of consistency across three quite distinct target languages, and ensure a basis for continued development once funding has finished.

Whilst this work has taken place in an artistic research setting, with motivations and methods that may be unfamiliar to some readers, we feel that this account points to questions and areas for further work that are possibly interesting to the PPIG community. On one front, it charts the conduct of a moderately-sized software project in the wild that has had to contend with the affordances of a range of languages simultaneously, and reveals some of the dance between pragmatism and principle involved in making this work. Meanwhile, answers to questions about how creative coding musicians might relate to and play with concepts from data science might well be usefully approached in the future by some combination of the release-it-and-see-what-happens approach we take here, and more controlled experimentation.

Acknowledgement

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 725899).

6. References

Bergström, I., & Blackwell, A. F. (2016). The practices of programming. In *2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (pp. 190–198). IEEE Computer Society. doi: 10.1109/VLHCC.2016.7739684

Buitinck, L., Louppe, G., Blondel, M., Pedregosa, F., Mueller, A., Grisel, O., ... Varoquaux, G. (2013). API design for machine learning software: Experiences from the scikit-learn project. In *ECML PKDD workshop: Languages for data mining and machine learning* (pp. 108–122).

Clarke, S., & Becker, C. (2003). Using the Cognitive Dimensions Framework to evaluate the usability of a class library. In *Psychology of Programming Interest Group (PPIG) 2003.*

Fiebrink, R. (2019). Machine Learning Education for Artists, Musicians, and Other Creative Practitioners. *ACM Transactions on Computing Education*, *19*(4), 1–32. doi: 10.1145/3294008

Garber, L., Ciccola, T., & Amusategui, J. C. (2020). AudioStellar, an Open Source Corpus-Based Musical Instrument for Latent Sound Structure Discovery and Sonic Experimentation. In *Proceedings of the ICMC 2020.* Santiago, Chile: Pontificia Universidad Católica de Chile.

Green, O., Tremblay, P. A., & Roma, G. (2018). Interdisciplinary Research as Musical Experimentation: A case study in musicianly approaches to sound corpora. In *Electroacoustic Studies Network Conference.* Florence, Italy.

McCartney, J. (2002). Rethinking the Computer Music Language: SuperCollider. *Computer Music Journal*, *26*(4), 61–68. doi: 10.1162/014892602320991383

McPherson, A., & Tahıroğlu, K. (2020). Idiomatic Patterns and Aesthetic Influence in Computer Music Languages. *Organised Sound*, *25*(1), 53–63. doi: 10.1017/S1355771819000463

Nash, C. (2015). The cognitive dimensions of music notations. In *International Conference on Tech-*

nologies for Music Notation and Representation (TENOR) (pp. 191–203).

Puckette, M. (1997). Pure Data: Another Integrated Computer Music Environment. In *International Computer Music Conference* (pp. 224–227).

Puckette, M. (2002). Max at Seventeen. *Computer Music Journal*, *26*(4), 31–43. doi: 10.1162/014892602320991356

Roma, G., Green, O., & Tremblay, P. A. (2019). Adaptive Mapping of Sound Collections for Data-driven Musical Interfaces. In *Proceedings of the International Conference on New Interfaces for Musical Expression* (pp. 313–318).

Schnell, N., Röbel, A., Schwarz, D., Peeters, G., Borghesi, R., & Pompidou, I. C. (2009). Mubu & Friends- Assembling Tools for Content Based Real-Time Interactive Audio Processing in Max/Msp. In *ICMC*.

Schwarz, D., Beller, G., Verbrugghe, B., & Britton, S. (2006). Real-Time Corpus-Based Concatenative Synthesis with CataRT. In *Digital Audio Effects (DAFx)* (pp. 279–282). Montreal, Canada.

Tremblay, P. A., Roma, G., & Green, O. (2022). Enabling Programmatic Data Mining as Musicking: The Fluid Corpus Manipulation Toolkit. *Computer Music Journal*, *45*(2), 9–23. doi: 10.1162/comj_a_00600

Zicarelli, D. (2002). How I Learned to Love a Program That Does Nothing. *Computer Music Journal*, *26*(4), 44–51. doi: 10.1162/014892602320991365