Revista Vórtex | Vortex Music Journal | ISSN 2317–9937 | http://vortex.unespar.edu.br/
D.O.I.: https://doi.org/10.33871/23179937.2021.9.2.18

Overview of the CEAMMC PureData distribution and library

Serge Poltavsky, Alex Nadzharov

CEAMMC | Russia

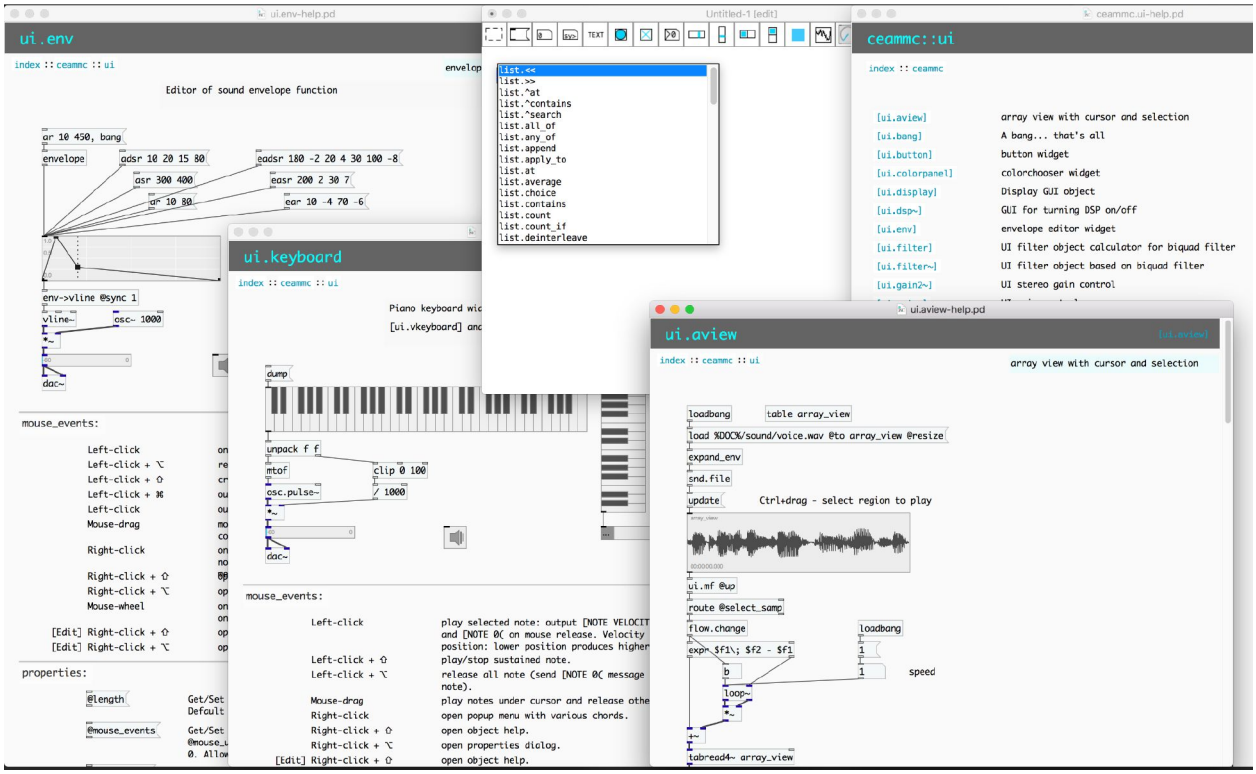
Abstract: This text is a report about the CEAMMC PureData distribution - also known as “Pd-ceammc” - and the CEAMMC external library for the official Pure Data distribution by Miller Puckette - also known as “Pd Vanilla”. The CEAMMC Pure Data repository is available at https://github.com/uliss/pure-data. This distribution is a software package that comprises the original Pure Data distribution with several user interface enhancements and the CEAMMC object library plus yet other external libraries. The CEAMMC library itself is also available via the Deken package manager so it can be installed in Pd Vanilla. This paper focuses on our main concepts for this work, current features and future plan.

Keywords: PureData, Externals library, CEAMMC

Received on July 1st, 2021. Approved on: November 1st, 2021. Available online on: 30/11/2021. Guest Editor: Dr. Alexandre Torres Porres.

The CEAMMC distribution and library is an idea of Serge Poltavsky and it was first released in 2016. The figure below is an overview of Pd-ceammc distribution and shows us the Help system, the object name auto-completion feature and several UI (user interface) objects from the CEAMMC library.

FIGURE 01 – Screenshot of the CEAMMC PureData. Help System, object name auto-completion, several UI objects[1]



Our distribution goal was to have a friendly distro for beginner students, a helpful and reliable product for ourselves that could be used in live concerts allowing us to solve our common music tasks quickly.

When you look at a popular, well-designed, and successful programming language like Python, you can see that besides the neat language syntax itself it provides a rich standard library with reasonable and predictable naming of modules and functions.

If we treat Pd as a visual language it is clear that the standard library cannot be considered as rich and consistent. The other visual programming language - MaxMSP, the Pd competitor, offers

[1] All images were provided by the authors.

much more built-in objects, but by reasons of legacy and compatibility the used naming scheme also is far from ideal.

Attempting to make a robust expansion for the standard Pd library we strongly needed good documentation, this led us to develop a documentation generator and documentation storage format based on XML.

The properties used in many languages also are a very convenient concept and were introduced to our library of externals. Also, many programming languages allow the use and construct arbitrary data structures. The first effort was done in our library in that direction: some extra data types were added beside standard Pd floats, symbols, and lists.

Object naming

Pd-vanilla has a small set of objects out of the box. For historic reasons some objects have ingenious names, but they are non-intuitive, a classic example is [moses]. If a student just wants to split numeric data flow by some condition, he should read all the documentation to find this object. This process draws the education process. There is no search in the help files like in almost any language, for example SuperCollider. All popular languages like Python have naming conventions for modules and functions within it. It seems that the only naming convention in Pd is the tilde symbol.

We think that objects should be grouped by categories they operate. This simplifies the search of needed functions especially for beginners. These categories give the object prefix. For example: all list related objects are named with the “list.” prefix (for instance: [list.equal], [list.insert], etc).

Documentation

Patches traditionally serve as documentation in Pd. These have pros and cons. It’s convenient to run visual pieces of code just in place. But searching object functionality by keywords, description in a bunch of Pd patches is impossible at this moment. Also, the lack of HTTP-accessible documentation leads to difficulty in internet discussions. While with many other

languages one can just put a link to a specific module documentation, there's no such possibility in Pd yet[2].

Also, the visual nature of Pd language leads to manual editing of documentation, which works well when you have 50 objects, and it becomes extremely difficult with more than 500. So, one of the first projects that came together with the CEAMMC library/distribution is the pddoc[3] documentation generator. The object documentation is stored in XML format and by specified templates pddoc utils generate Pd help patches, categories, and library indexes. Markdown format also is possible, which allows us to provide HTML documentation as well.[4]

EXAMPLE 01 – pddoc file

```
<?xml                          version="1.0"                          encoding="utf-8"?>
<pddoc                                                                    version="1.0">
  <object name="the_object">
    <title>list.schindler</title>
    <meta>
      <authors><author>Steven                              Spielberg</author></authors>
      <description></description>
    </meta>
    <arguments>
      <argument name="ARG1" type="symbol">first arg</argument>
    </arguments>
    <properties>
      <property name="@year" type="int">year of creation</property>
    </properties>
    <inlets>
      <inlet><xinfo on="bang">start movie</xinfo></inlet>
    </inlets>
    <outlets>
      <outlet>description</outlet>
    </outlets>
    <example>
      <pdascii>
<![CDATA[
[play(
|
[list.schindler]
]]>
      </pdascii>
```

[2] There are plans to offer the HTML documentation for Vanilla objects soon.

[3] See https://github.com/uliss/pddoc

[4] Online help available at https://ceammc.github.io/pd-help/help-en/

```
    </example>
  </object>
</pddoc>
```

PdASCII, a special syntax for Pd examples was added. It simplifies the process of adding the usage examples in pddoc files, see below how it is used to connect two object boxes.

EXAMPLE 02 – PdASCII

```
[mtof]
|
[F]
```

Properties

Many core Pd objects have a internal state that can be changed but not retrieved. For example, **[delay]** has a delay value, but you can't know it. You must do additional tricks to do that: create an additional **[f]** object that stores it thus adding a new state and task to sync them together.

The changing internal state is traditionally considered to be the source of many mistakes. So first, all of our objects support the "dump" message that prints the object's state into the Pd console. This serves mainly for debugging purposes, because you can't handle this information in the Pd patch itself.

The concept of properties is to have a standard mechanism to store and retrieve an object's state. If the user can change the object state it also should be able to retrieve it. Properties become especially useful when objects have many tunable options, for example Graphical User Interface objects. But all ceammc library objects support this mechanism, not only GUI objects.

The property syntax is adopted from CICM[5] libraries for Pure Data (that is also used in our build) - the property is also a special kind of argument for an object, it starts with "**@**" sign with property name and the data - until the next property definition. The syntax seems to be inspired

[5] **CicmWrapper** (https://github.com/CICM/CicmWrapper) and **CreamLibrary** (https://github.com/CICM/CreamLibrary) by Pierre Guillot, both are abandoned now.

from MAX's 'attributes', that you can also access and change in the patch via the **[attrui]** object. In a sense, attributes are a bit similar to 'flags' in Pd[6], whose syntax is "-" and are used to set optional arguments in more complex objects (see objects like **[sigmund~]** and **[soundfiler]** as examples). GUI (and other objects) in ceammc can be created with properties like this: **[ui.slider @size 300 10].**

Pure Data Vanilla GUI objects don't have anything similar. Atom boxes and arrays can't even be created as objects with arguments. IEMguis (bang, toggle, sliders, etc) were originally an external library further incorporated into Vanilla, for this reason they can be created as objects with arguments, but the order of arguments is not documented, and they lack the ability to use flags. Moreover, even if they did, you cannot retrieve their state unless you open the properties window. Like other Pd objects and features, IEMguis have an incredibly stable interface that hasn't changed much, so we don't expect this to change. Nevertheless, it's an unnecessary restriction. This is really strange if one treats Pd as a programming language because all known languages allow properties to be *readonly* or *readwrite*, but here is an example of *writeonly* property.

There can be many reasons to retrieve the properties of an object. Maybe you are debugging something, or you want to save them in some way. Maybe you want to set a new value based on current, for example make the color darker or brighter, etc.

Message property access

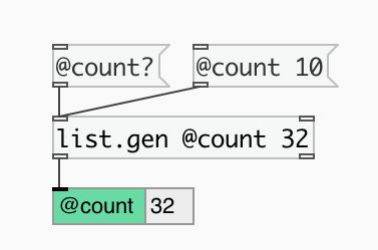
There are two ways of accessing properties. Via Pd messages and with **[prop.get]/[prop.set]** objects that use internal Pd calls. Each property can be retrieved by sending a property get request message to the object and can be set by sending property set request message to the object. For instance, **[list.gen]** is a list generator that has property **@count** that defines the number of generated items.

You can change this value by sending the "@count 10" message to the object. The property just changes, and the object does and outputs nothing. You can get this value by sending the

[6] The key difference between flags and properties is that the first cannot be read, they are write-only, whilst properties can be read and write.

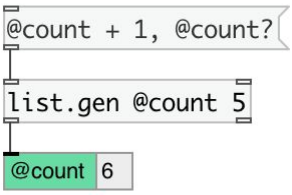
"@count?" message to the object. Upon receiving this request, the object sends the message "@count *CURRENT_VALUE*" to its first outlet, after that it can be routed and handled.

FIGURE 02 – Patch



Numeric properties support some extra capabilities - sending "@count + 1" to **[list.gen]** object will increment the object's property **@count**. Also, numeric properties with bounds can be randomized, for example you can send a "@count random" message to set the object's property randomly in the allowed range (using a uniform distribution).

FIGURE 03 – Patch



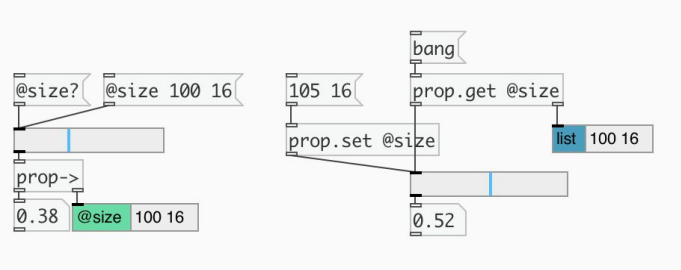
You also can get list of all properties with the "@*?" message if you want to do some introspection and iterate with all available properties.

FIGURE 04 – Patch

Pulled property access

The other way to get/set properties is with the **[prop.get]**/**[prop.set]** objects. You should connect their first outlet to the target object, and they will do their job via Pd internal black magic. Sometimes this type of connection is more convenient.

FIGURE 05 – Property handling with **[prop.get]**/**[prop.set]**



Data types

Popular general programming languages can deal with different data types, can define their own types and also have powerful built in container support. With common containers (like string, stack, dict, set, fifo queue) users have no need to reinvent the wheel and can focus on main tasks of logic.

Understanding that Pd's primary goal is to programming DSP in realtime, dealing with sensors, network requests (including text protocols like HTTP), traversing file systems, doing generative algorithms, we may find that standard Pd types (**float**, **symbol**, **list**, **pointer**, **array**) may be quite limited for the specified complex tasks.

Limitation of standard Pd types

float

The most commonly used Pd distribution uses a single precision (32-bit) float as a main numeric type. Apparently the next Pd Vanilla release (0.52) may provide double precision (64-bit)

builds, which is needed, for instance, when indexing large sound tables. Pd-vanilla is currently available in double precision only if you compile it like that. Besides single precision, we also provide both Pd-ceammc distribution and ceammc library in double precision. The ceammc library in double precision is also available in deken and you can download it if you compiled Pd for double precision.

Symbol

Many people think that the symbol type in Pd is a string, but internally it's not. Each time you create a new symbol it creates a new entry in the internal Pd hashtable. This memory will never return even if you don't use this symbol anymore. Thus, constantly creating new symbols on your running Pd will eventually consume all the available memory on the system. This can be a terrible situation when Pd runs for a long time on embedded devices with low memory, such as a Raspberry Pi used in sound installations, etc.

For this reason, developers offer only a few objects to deal with symbols, especially that create the new ones. In Pd Vanilla there are two ways to create a new symbol. First is **[makefilename],** that by its name signals the user that it should be used only in seldom cases. The second way is dollar args expansion in messages, like "message_$1".

Core developers understand that making C-like objects (sprintf, strcat etc.) dealing with symbols is rather easy, the difficult part is what to do with the constant hash table growth.

So, using the **symbol** type as a string is a real bad thing, you shouldn't do it on long running Pd programs which, for example. get some changing string data for sound installations via HTTP, read strings from files, etc. To solve this problem, we need a missing **string data type**.

list

In general, the list type in Pd is ok. The lists are destroyed after usage, so they do not occupy memory after that. Only one limitation exists, that lists are 1-dimensional. So, you can't have a list of lists, etc. For instance, if we have to store MIDI Note On events as a 3-element list (for channel,

note and velocity) we can't store the whole melody as a list of Note On events. To do so we have to flatten the list to one dimension. If we had a multidimensional list this task could be done much more easily.

pointer

In Pd you can define your own data structure, but the interface is unclean and so misleading that we think it's not worth, in the educational process, to touch these dark corners as the advantages come at a very high cost as shown by our practice.

CEAMMC data types

We created an extended data type system using only the ceammc library of externals, so it works both in Pd-vanilla and Pd-ceammc.

We introduce a special atom type: **dataatom**. Actually, you can treat it as a pointer to concrete data type with a reference counter, which means that after you are creating new data it will be freed when no object will use it anymore. You can create thousands of Strings, but their memory will return to the system when you are finished with them.

The **dataatom** itself is a simple Pd atom that holds the custom data type inside, so you can pass Pd lists that contain **dataatoms** etc.

String

This is the missing String data type, it supports all common string operations: single character access, substring extraction, substring search, comparing, string split by separator, concatenation etc. All operations support UTF-8 encoding so you can use any language.

Set

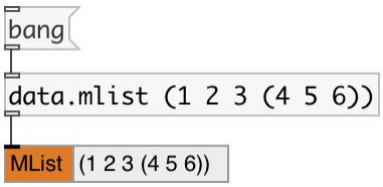
This data type allows storing unique items in a "list"-like container, so you don't need to do these specific common tasks (filtering duplicates etc) as in case you were working with lists. You can simply convert 'list' to 'set' (by sending a list to the **[data.set]** object) to filter duplicate values (i.e. **1 2 2 3** becomes **1 2 3**) and then process this data (i.e. with the **[set.union]**, **[set.intersect]**) still keeping the duplicate values away.

FIGURE 06 – Patch

MList

Multi-level list. This data type provides an option to have nested lists. All the **[list.*]** objects support this data type. The multilevel list syntax has the same syntax as Lisp: `(1 2 3 (1 2 3))`

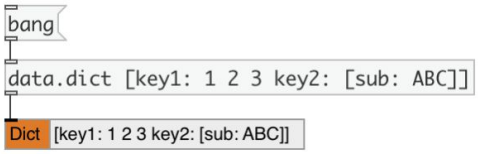
FIGURE 07– Patch



Dict

Dictionary. This typw allows us to get/store elements by key. It can be loaded from or saved as JSON files. The dictionary syntax is a little bit python-like: `[key: 1 2 3 key2: [sub: ABC]]`

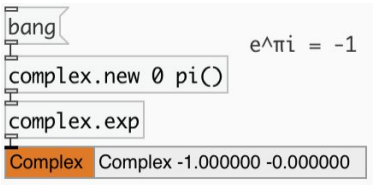
FIGURE 08 – Patch



Complex

This is a set of objects for complex math. It must be explicitly loaded with **[declare -lib ceammc/numeric]**, both in our distribution and in Vanilla.

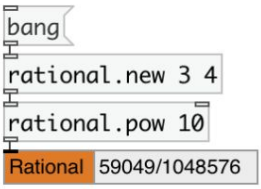
FIGURE 09 – Patch



Rational

This type is available in the **numeric** library just mentioned. It adds support for rational number calculus without precision lost.

FIGURE 10 – Patch

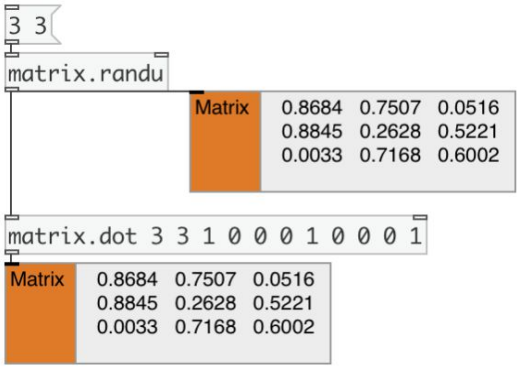


Matrix

We have a rich set of objects for matrix manipulation with the **matrix** sub library. You must explicitly load it with **[declare -lib ceammc/matrix]**. Based on the Armadillo library[7], this has more than 90 objects to perform matrix math.

[7] See http://arma.sourceforge.net/

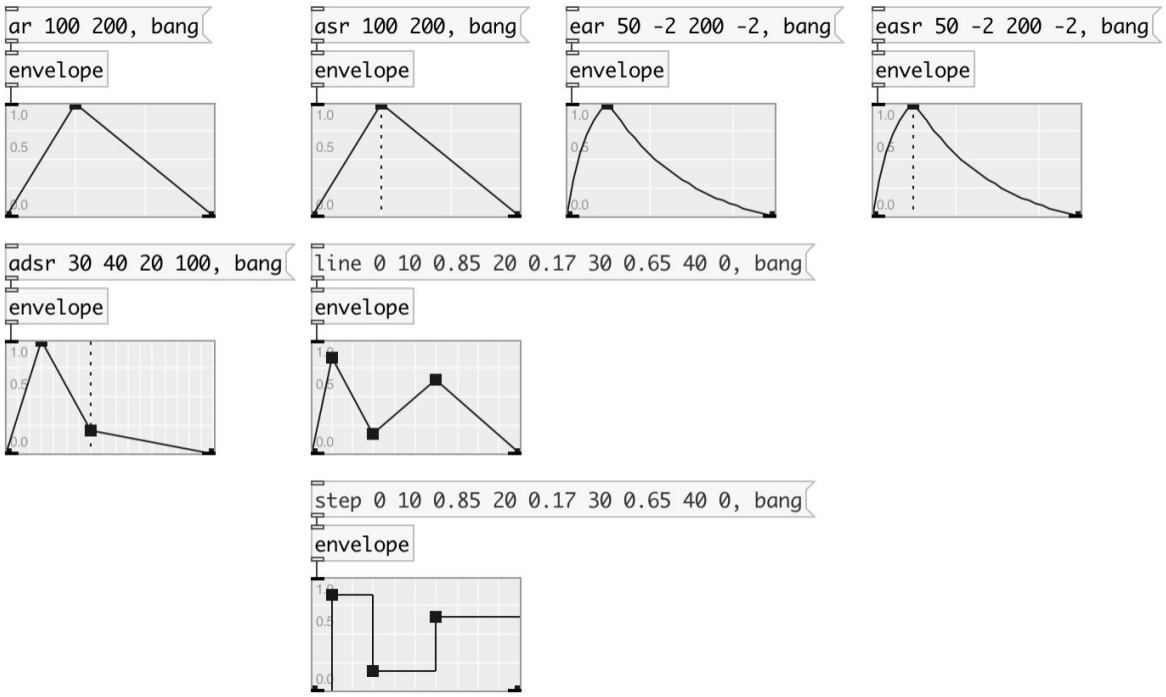
FIGURE 11 – Patch



Envelope

This is a data type for envelopes or breakpoint functions. It consists of points and connecting segments. It is a more generic type than a classic ADSR envelope, it can describe more complex breakpoint functions that contain arbitrary number of points, stop-points and segments.

FIGURE 12 – Patch



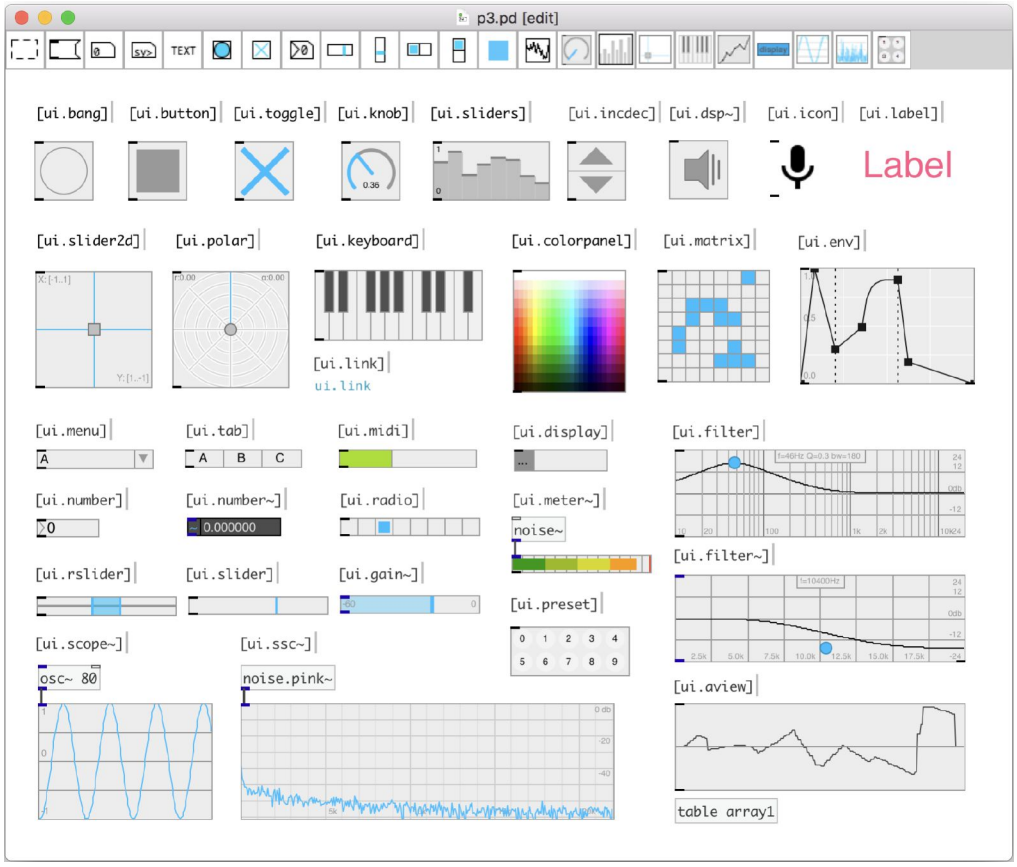
The **[env.*]** objects can perform various manipulations on envelopes, you can: shrink, stretch, edit points and segments, add stop points, etc. The GUI object for interactive Envelope edits is: **[ui.env]**.

Graphical User Interface objects

When we started our project there were no solved issues in Pd-vanilla with IEMgui objects: zoom support was broken on some of them. We tried to understand and fix these issues, but the vanilla GUI codebase has a hard learning curve for newcomers. Also, we found nice modern objects from the Cream Library and decided to include our own GUI objects based on this library into the **ui** category.

At the time of ceammc's v0.9.4 release we have 35 objects in this category. A MIDI learning functionality is available for most the important UI objects, like the **[ui.knob]**. MIDI learning is performed on the level of the external, so it works in Pd-vanilla as well[8].

FIGURE 13 – Patch



[8] In performance mode **Double-click + Shift** on an object (for example **[ui.slider]**), the object border becomes red, the object now is in MIDI learning state. Move your MIDI control, then the red border becomes blue, this means the binding is done and the object is waiting to pick up MIDI value, it will not change until it receives a value close to current one, this prevents sudden control jumps when changing values both from GUI and MIDI.

Presets

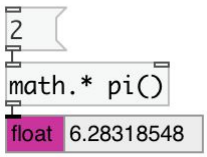
As we are performers, it's a frequent task during rehearsals to save the performance's settings. Existing preset systems for Pd weren't suitable for our needs. We have a simple preset system concept by just putting **[ui.preset]** in the patch and you are ready. You can read/write presets with that GUI object, you can linearly interpolate between them, you can store presets to text files and load them back. All of our GUI objects support this preset system, and no additional objects are required. If you also want to manage presets with Vanilla's Guis, the **[preset.float]** object is available.

Argument's expansion

Pd-vanilla supports dollar argument expansion, which is a very useful feature. Almost all ceammc objects (with some reasonable exceptions) extend this syntax so you can use expressions and function calls when the object is initialized.

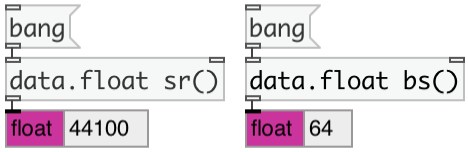
In the simplest example below with **[math.*]**, when the object is created, the **pi()** argument is replaced with π value, literally 3.1415926.

FIGURE 14 – Patch



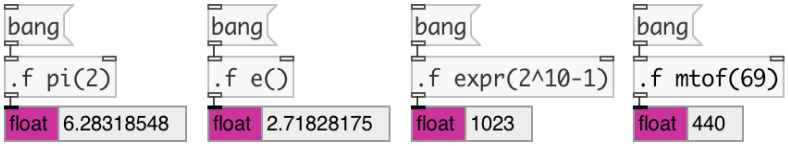
The **sr**, **bs**, **env**, **pi**, **e**, **expr**, **mtof**, **ftom**, **ms2bpm**, **bpm2ms**, **db2amp**, **amp2db**, **seq**, **repeat**, **reverse**, **rtree**, **euclid** functions are equally supported. They can be divided into several groups. The First group is functions for getting Pd's current running information: **sr()** returns the current sample rate, **bs()** evaluates to current block size, **env(%VAR%)** returns system environment variable value.

FIGURE 15 – Patch



The second group is for math calculations: **pi()** returns a value of π, **pi(2.5)** returns 2.5π, **e()** returns *e*, **expr()** calculates math expressions like expr(2^14-1), **mtof(v)**/**ftom(v)** and others are unit converters.

FIGURE 16 – Patch



The last group is for dealing with lists: **seq(from/to/step)** generates a float list sequence with a specified range and step, **repeat(n args...)** creates list repeating args 'n' times, **reverse()** returns the list in backwards order, **rtree()** expands to the Rhythm Tree syntax and **euclid()** generates Euclid beat patterns. Function calls can be nested, by the way.

FIGURE 17 – Patch

Also, integer numbers in Hex and Bin format are recognized. We found this a very helpful feature, but you should note: it is supported **only** by ceammc library's objects, not in the Vanilla's objects in Pd-ceammc.

FIGURE 18 – Patch

OS support

In the teaching process we needed to support all Operating systems used by students. So there are releases of our distribution for macOS, Windows and Linux. MacOS and Windows have 64-bit single and double precision versions. The Linux version is only single precision now and distributed via the Snap package.

The ceammc library of externals is available for Pd-vanilla via deken with MacOS and Windows variants. There's no deken support for Linux possible yet, because our external depends on some extra libraries and we can't distribute the ***.so** needed files via the deken package manager. Even if we compile it statically, we can get into trouble with different toolchains in different distros. We hope Linux distro maintainers will be interested to support this natively via system package managers.

Distribution

The Pd-ceammc was initially considered as a distribution like Python - "all batteries are included". This simplified the deployment process during masterclasses and the educational process. But we understand the importance of choice, so we also made our externals available to Pd-vanilla users via deken. The main feature of our distribution is in the ceammc library of externals itself. So our Pd-ceammc is more of a distro than a real fork. The distribution also contains some

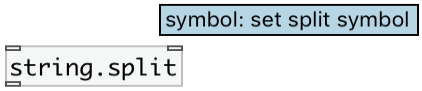
third-party external libraries that we found useful for the learning process and for the ease of the live performance.[9]

The differences to Pd-vanilla are: 1) Different visual design, such as different colors for audio and control in/outlets (which is more descriptive for newbies in our opinion and was the standard in Pd-extended); 2) The Tcl/Tk version in our distro usually is more fresh than in Pd-vanilla's; 3) We added a toolbar plugin that is shown in edit mode thus giving a possibility to distinguish the edit and locked modes; 4) The autocompletion plugin (that is available via deken) is also included in our distribution.

Moreover, Pd-ceammc on macOS has a newer native CoreMidi driver support, while Pd-vanilla uses an almost abandoned cross-platform PortMidi. When you connect a new MIDI controller to your computer while Pd-ceammc is running, you can see this controller available in the MidiSettings dialog right away. This doesn't happen in Pd-vanilla on macOS and you should restart it to see the updated device list. This CoreMidi driver also fixes SysEx message audio-dropouts in Pd on macOS.

Pd-ceammc supports inlet/outlet tooltips for ceammc objects that can provide this information, just move mouse pointer over inlet/outlet and, after a short time period, a popup tooltip will appear[10]. Pd-vanilla has nothing like this yet.

FIGURE 19 – Patch



Another major difference between Pd-vanilla and Pd-ceammc is the distribution's licenses: Pd-vanilla is BSD, while Pd-ceammc is GPL3[11].

[9] The distribution includes the following externals: vasp, zconf, xsample, disis_munger, comport, autotune, FFTease, LyonPotpourri. Almost all of them were not available in deken when the project was started.
[10] Not all of our objects provide this information yet.
[11] Because Ceammc uses code under GPL licences

Development

We try to make our distribution stable, reliable and production ready, for this we are using unit tests that are run on every platform. Almost any new object logic is tested, this allows us to safely do refactoring when needed. We are constantly syncing the Pd code base with the recent Pd-vanilla version, lagging behind usually no more than one minor version. When designing new objects, we are trying to make more dataflow than spaghetti-like. We prefer properties then using multiple inlets/outlets when possible.

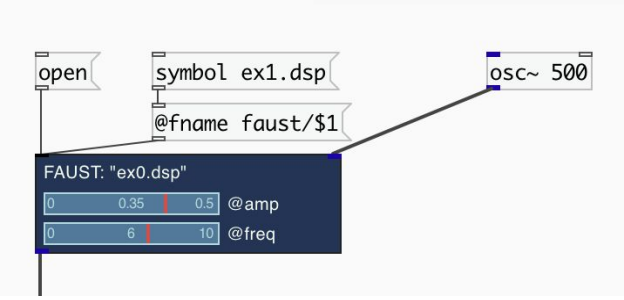
More features

Faust Programming Language

We are greatly inspired by Faust[12] and more than 100 of our objects are generated from Faust libraries, which allows us to easily provide a big bunch of DSP related objects: filters, compressors, reverbs, wave guided synthesizers etc.

Also we have experimental support for compiling files with faust code on the fly. For this we have **[lang.faust~]** and **[ui.faust~]** objects. There are other Pd externals with the same functionality[13], but ours is the only that can generate GUI interfaces on the fly.

FIGURE 20 – Patch



[12] Check https://faust.grame.fr/

[13] See https://github.com/CICM/pd-faustgen and https://agraef.github.io/pure-docs/pd-faust.html

Audio formats support

The Ceammc library contains the **[snd.file]** object that supports loading various sound file formats not supported by vanilla's **[soundfiler]** such as: Flac, Ogg, Opus, MP3 and many others. The **[snd.file]** loader also supports resampling of loaded files to a desired sample rate, if your running Pd sample rate doesn't match the file's one. With **[snd.file]** you can precisely specify which part of the sound file you want to load not only in terms of samples, but also in other time units: milliseconds, SMPTE etc.

Sound fonts support

SF2 and SFZ sound fonts are supported by **[fluid~]** and **[sfizz~]** objects.

TTS

Text to speech real time synthesis engine is included into the ceammc library's, see the **[speech.flite~]** external. It is based on the CMU Flite engine.[14] Only the English language is supported by now.

Future plans

There are many interesting domains to us: Artificial Intelligence (generating music and recognition patterns), better TTS support for other languages, automatic GUI generation, convenient remote control, support for Internet protocols, generative algorithms, support for a wide range of Linux distros etc.

[14] http://www.festvox.org/flite/

ABOUT THE AUTHORS

Serge Poltavsky is a viola player with great experience in performing very different music, composed and improvised, with and without the electronics. ///// Alex Nadzharov is a composer and multimedia artist working much with live electronics and visuals. E-mail: anadjarov@gmail.com