Lessons from Teaching Music-Informatics to Musicologists

Albert Gräf <aggraef@gmail.com>
IKM, Music-Informatics at JGU Mainz

November 2015

JOHANNES GUTENBERG UNIVERSITÄT MAINZ

Synopsis

- Music-Informatics at JGU in Mainz
- Some Thoughts on Teaching Music-Informatics
- Programming Languages and Tools
- Conclusions

JOHANNES GUTENBERG UNIVERSITÄT MAINZ

Bridging the Gap…

JOHANNES GUTENBERG UNIVERSITÄT MAINZ

Bridging the Gap…

- From *teacher* to *students* (didactics)
- From *science* to *art* (interdisciplinarity)
- From *theory* to *application* (practice)
- From *basics* to *expert knowledge* (specialization)
- From *non-programmer* to *programmer* (**languages and tools**)

Music-Informatics at JGU

- **Students:** mostly Musicology and Music, but also Computer Science, Mathematics, Media Science; “Digital Humanities” Master in the making
- Various **courses of studies:** B.A., B.Ed., Master; up until this year also traditional Magister and Diploma studies (currently **282** students in total, **221** among them BA/BEd, **134** in Musicology)
- Most courses either take the form of **seminars** or **tutorials**, number of participants ranging from 5 to 40 students, typically some 20-30 students per semester
- Self-built **computer music lab** (up to 10 students) with MIDI/OSC equipment and PCs, mostly running Linux and other open-source software

JG|U
JOHANNES GUTENBERG UNIVERSITÄT MAINZ

Some Questions

- **Who?** Who should learn Music-Informatics? Who will (or should) care about it?
- **Why?** Why teach Music-Informatics at all? What are the benefits?
- **What?** What are the important topics? What are the prerequisites (math, computer science, music theory, etc.)?
- **How?** How to best convey these topics? How to bridge the gap between theory and practice? How to equip students with the necessary prerequisites?

Our Situation at JGU

- **Who?** Students from widely different backgrounds in the same course pose a considerable challenge ⇒ working in groups and teams where the kind of course permits it
- **Why?** This is all about motivation; students have their own agendas, driven by courses of studies but also by what they find interesting, worthwhile *and (last but not least) fun*
- **What?** Wide interdisciplinary range of subjects, "classical" (e.g., mathematical theory of music, signal processing) and "hot" topics (e.g., interfaces for musical expression, mobile devices)
- **How?** Requires a kind of Faustian mindset ("Two souls alas! are dwelling in my breast.")

JG|U
JOHANNES GUTENBERG UNIVERSITÄT MAINZ

What?

algorithmic composition	remixing
sound synthesis	**digital sound processing**
mathematical theory of music	sound design
tunings and temperaments	**musical codes**
scales and modes	sound and video technologies
musical acoustics	plug-in technology
psycho-acoustics	controller technology
music notation	**computer programming**
musical analysis	database and web technologies

JG|U

JOHANNES GUTENBERG
UNIVERSITÄT MAINZ

How?

- **Scientific method:** enable students to do their *own research* (lectures, seminars)
- **Practical training:** enable students to actually *apply* theoretical knowledge in their research and artistic work (tutorials)
- Trying to integrate as much **group work** and **mini projects** as possible
- Still, **programming** seems to be the biggest hurdle for most students (**mathematics** comes second place)
- In my experience, there's **no silver bullet**, it's just practice, practice, practice, …, so we should make that as easy as possible!

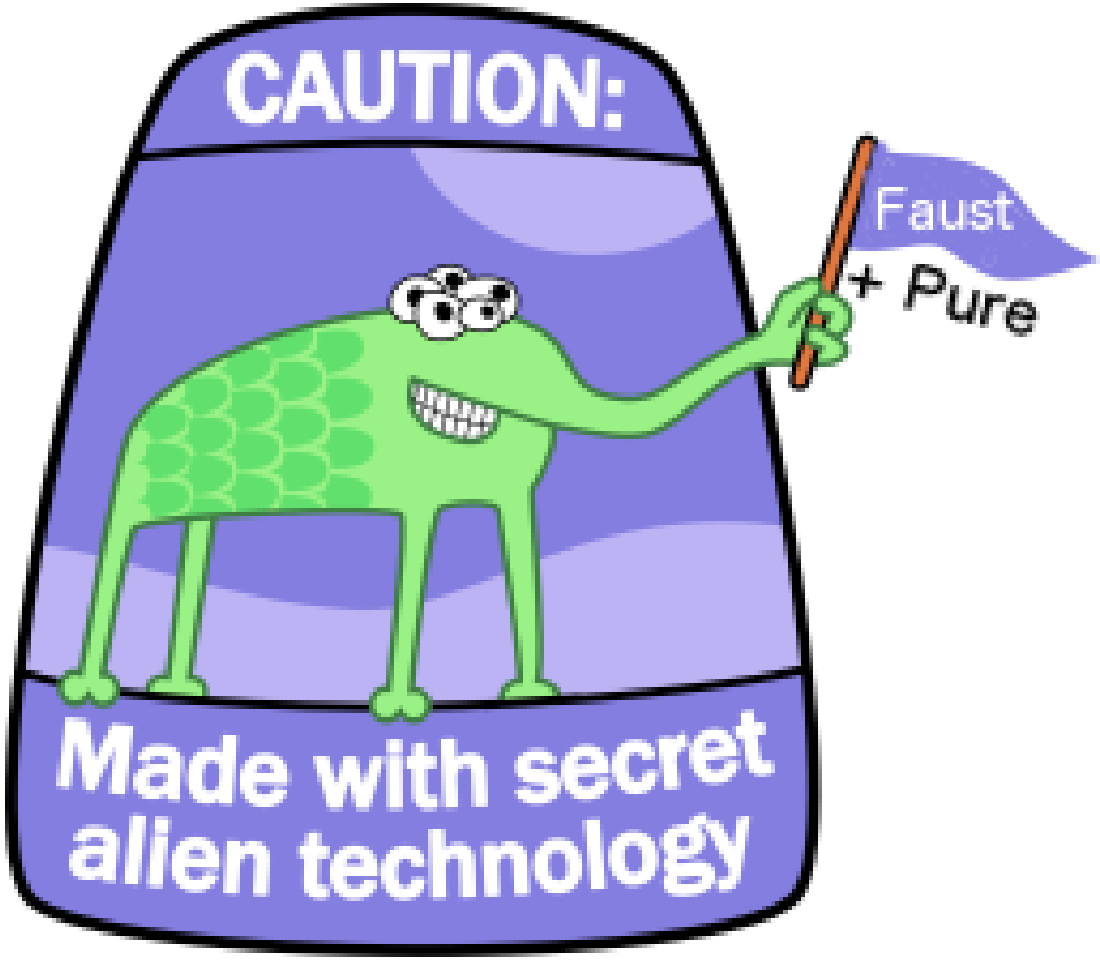
JG|U
JOHANNES GUTENBERG UNIVERSITÄT MAINZ

Tools

Trying to use *portable open-source software* as much as possible (we have Linux in the lab, but students usually have Mac or Windows systems).

- audio and video editing software (Audacity, Kdenlive, Openshot)
- DAW software (Ardour, *Reaper*, *Tracktion*)
- samplers and sequencers, software synthesizers, etc. (QSynth)
- notation software (Lilypond, Frescobaldi, MuseScore)
- **visual dataflow programming** (Pd)
- **programming languages** (Faust, Pure)
  - imperative, object-oriented, functional?
  - general-purpose versus DSL?
  - experimental or mainstream?

Tools



Pd, Faust and Pure

- Like Max/MSP, **Pd** (a.k.a. Pure Data) is a kind of *visual* programming environment (also known as a *dataflow* programming language) to build complex sound processing algorithms from simple building blocks.
- Pd is **easy to work with** but lacks the flexibility of a full-blown programming language. Its built-in objects basically determine what you can do with it.
- If you need to go beyond that then you have to create your own objects, called **externals**. Normally this is done in C, but using **Pd-Pure** and **Pd-Faust** you can easily do this in the Pure and Faust programming languages.

Pd-Faust

- As you all know, **Faust** is a functional programming language for creating signal processing plug-ins.

```
import("music.lib");
gate = button("gate");
gain = hslider("gain", 0.3, 0, 3, 0.01);
freq = hslider("freq", 440, 20, 2000, 1);
process = gate*gain*osc(freq);
```

- The **Faust compiler** can produce efficient, native code for various environments, including Pd and Pure.
- **Pd-Faust** is a library of Pd externals written entirely in Pure which lets you load Faust dsps **dynamically** in Pd.

JG|U
JOHANNES GUTENBERG UNIVERSITÄT MAINZ

Pd-Pure

- **Faust** is tailored for processing of synchronous numeric data, great for doing instruments and audio effects.
- **Pure** comes in when symbolic processing of complex control data is needed.
- Pure is based on **term rewriting** (symbolic evaluation of expressions), but it compiles to **efficient native code** on the fly, using **JIT** (just in time) compilation via **LLVM**.
- **Pd-Pure** is a Pd plug-in loader which lets you program Pd objects in Pure. It also supports **dynamic reloading** of Pure programs to ease **debugging** and **live-coding**.

JG|U
JOHANNES GUTENBERG
UNIVERSITÄT MAINZ

Pure: A (Very) Quick Overview

- Pure is a functional programming language based on *term rewriting*, i.e., the *symbolic evaluation* of expressions.
- Thus, all data in Pure takes the form of *expressions* (also called *terms*), like `12345`, `3.1415`, `"abc"`, `bang`
- Compound expressions are formed using function *applications*, like `fib 21`, `note 60 127`, `1:2:3:4:5:[] = [1,2,3,4,5]`
- Functions are defined using *equations* and *pattern matching*, e.g.:

```
fib 0 = 0; fib 1 = 1;
fib n = fib(n-1) + fib(n-2) if n>1;
```

- Equations are used as *term rewriting rules* in order to reduce expressions to their simplest form (*normal form*):
  `fib 2 => fib 1 + fib 0 => 1 + 0 => 1`
- Cue short **live demo** here

JG|U
JOHANNES GUTENBERG UNIVERSITÄT MAINZ

Polymorphism and Dynamic Typing

- All data structures and functions in Pure are **polymorphic**, i.e., they can take arguments of as many different kinds of data as you like.
- This is also known as **dynamic typing**, as opposed to **static typing** where the types of arguments are restricted. Think (e.g.) Python versus Java, or Lisp versus Haskell.
- Makes things **much easier** when plugging into a dynamic environment like Pd.
- Pd messages (numbers, symbols, lists) can be **mapped 1-1** to corresponding Pure data and vice versa in a straightforward manner.

Pd Objects the Imperative Way: The Actor Model

- Pd basically uses an actor model of computation
- An object takes input data from its *inlets*, …
- … performs some computation, …
  (possibly modifying its internal *state*)
- … and sends output data to its *outlets*.

```
counter = process with
  process _ = n when
    n = get counter;
    put counter (n+1);
  end;
end when
  counter = ref 0;
end;
```

Pd Objects the Functional Way: The Stream Model

- The actor model is inelegant because it requires internal state, i.e., **side-effects**.
- But there's a way to describe the operation of an actor in a **purely functional** manner, without any side effects: **stream processing**.
- Example: the Fibonacci numbers as a stream:

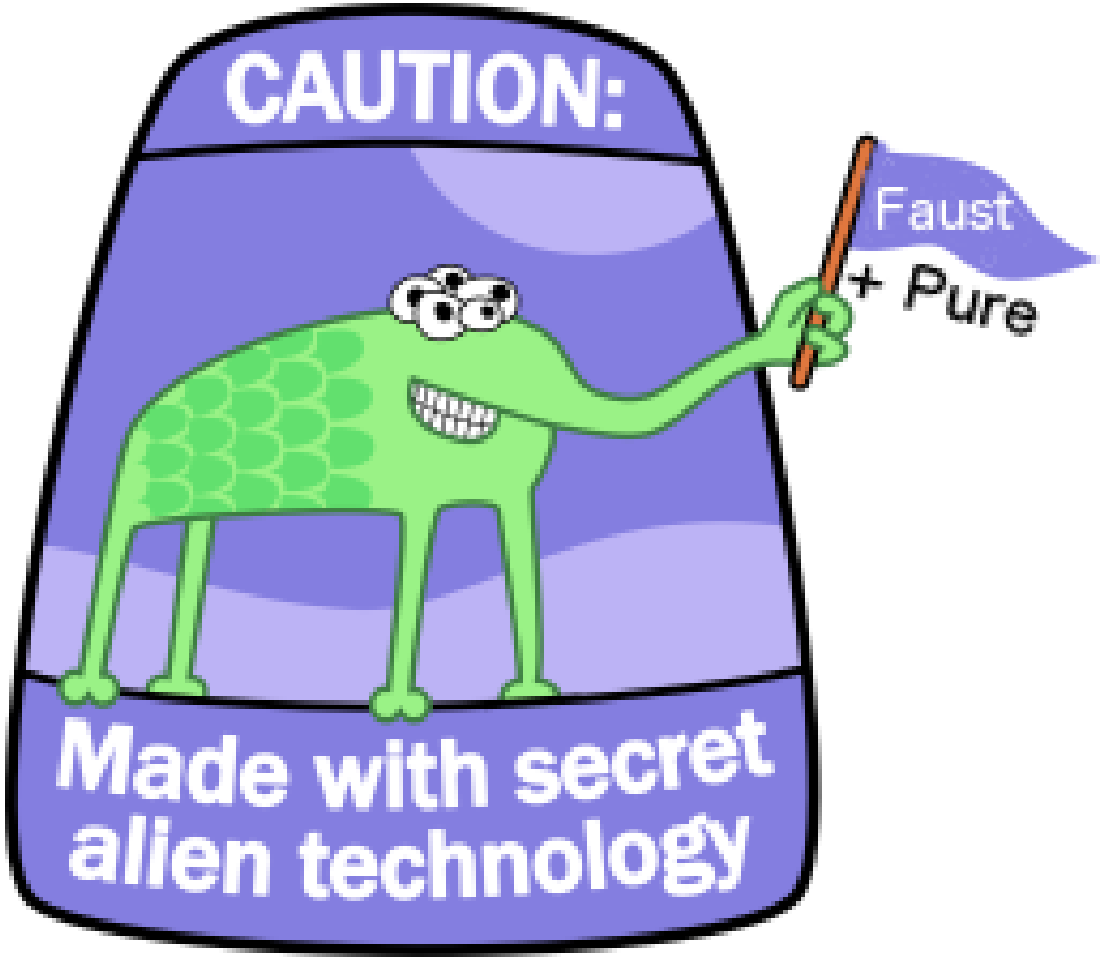
```
fibs = fibs 0 1 with fibs a b = a : fibs b (a+b) & end;
```

- Turn the above stream into a stream processing function for use with Pd:

```
using actor;
fibs = actor (f (fibs 0 1)) with
  f (x:xs) (y:ys) = x : f xs ys &;
  fibs a b = a : fibs b (a+b) &;
end;
```

JOHANNES GUTENBERG UNIVERSITÄT MAINZ

Demo: Pd-Pure and Pd-Faust



JOHANNES GUTENBERG
UNIVERSITÄT MAINZ

Future Work

- **Also available:** Pure and Faust LV2 and VST plug-ins which can be run in *DAW programs*; the latter provide a standardized interface to *polyphonic/multitimbral instruments* with MIDI *controller support* and MTS *tuning capabilities*
- **Long-term goal:** *make all these come together* under a **universal plug-in interface** ("PlugR") which will let you run the same plug-ins in AU/LV2/VST hosts or Csound/Max/Pd/SC3 etc. *without any (source) changes*
- **Short-term:** Unification of the LV2/VST interfaces, improved support for the LV2/VST/Pd-related architectures in the *Faust online compiler*
- We need a **book on Faust** (maybe join forces?)

Download Links

My stuff on **Bitbucket** (Pure, pd-faust, faust-vst, etc.):
`https://bitbucket.org/agraef/agraef.bitbucket.org`

Pure: `http://purelang.bitbucket.org/`

Faust: `http://faust.grame.fr/`, SourceForge:
`http://sf.net/projects/faudiostream/`

JOHANNES GUTENBERG UNIVERSITÄT MAINZ