

LuaTeX lunatic

And Now for Something Completely Different
– Monty Python, 1972

Abstract

luatex lunatic is an extension of the Lua language of luatex to permit embedding of a Python interpreter.

A Python interpreter hosted in luatex allows macro programmers to use all modules from the Python standard library, allows importing of third modules, and permits the use of existing bindings of shared libraries or the creation of new bindings to shared libraries with the Python standard module ctypes.

Some examples of such bindings, particularly in the area of scientific graphics, are presented and discussed.

Intentionally the embedding of interpreter is limited to the python-2.6 release and to a luatex release for the Linux operating system (32 bit).

Keywords

Lua, Python, dynamic loading, ffi.

History

I met luatex sometime around November 2006, and I started to play with it to explore the possibility of typesetting xml documents in an alternative way than the traditional approach based on xsl stylesheet plus xslt processor.

My first intention was to typeset a wikipedia xml dump [4] which is compressed with bzip2; given that I mainly use Python for my programming language, I quickly found python-bz2 and then Gustavo Niemeyer's "personal laboratory" [15] where I have discovered Lunatic Python [14].

To avoid confusion, here Python means CPython, the C implementation of the Python language [49]. There are other implementations of the Python language: for example Jython [41] and IronPython [37]. According to [6] "the origin of the name (is) based on the television series Monty Python's Flying Circus."

In March 2007 I started to investigate the possibility of integrating Lunatic Python with luatex [57] and in August 2007 I made the first release of luatex-lunatic [20], just around the birth of my second daughter Martina (09/08/07, [33]).

During the 2nd ConTeXt meeting [21] I found that luatex was stable enough to finalize the project, so I remade all steps and some new examples too (ConTeXt meetings are good places for these kinds of things).

Examples are now hosted at contextgarden [35] while [20] remains for historical purposes.

Motivations & goals

TeX is synonymous with portability (it's easy to implement/adapt TeX *the program*) and stability (TeX *the language* changes only to fix errors).

We can summarize by saying that "*typesetting in TeX tends to be everywhere everytime.*"

These characteristics are a bit unusual in today's scenario of software development: no one is surprised if programs exist only for one single OS (and even for a discontinued OS, given the virtualization technology) and especially no one is surprised at a new release of a program, which actually means bugs fixed and new features implemented (note that the converse is in some sense negative: no release means program discontinued).

Of course, if we consider the *L^ATeX-system*, i.e. L^ATeX and its most used packages, this is not frozen at all: just see the near-daily announcements from CTAN. pdfTeX also changes to follow pdf releases.

With luatex-lunatic I adopt this point of view: LuaTeX or more specifically LuaTeX & ConTeXt-mkiv as a **tool** for publishing content, with some extent to content management. As a tool, it is no surprise if there are "often" (for a TeX user) new releases, given that we can have a LuaTeX update, or a ConTeXt-mkiv update, or a Lua update, or a Python update, or a library update for which the Python binding exists; and, of course, if made with no "cum grano salis", no surprise if this can become quickly unmanageable.

The price to pay is the potential **loss of stability**: the same document (with the same fonts and images) processed with a new release can produce a different output.

With regard to portability, the LuaTeX team uses libtool: *GNU Libtool simplifies the developer's job by encapsulating both the platform-specific dependencies, and the user interface, in a single script. GNU Libtool is designed so that the complete functionality of each host type is available via a generic interface, but nasty quirks are hidden from the programmer* [39]), while in Lua and Python support for dynamic loading is a feature of the languages, i.e. there is a (Lua/Python) layer that hides the details of binding.

Thus stated, due to the lack of resources, I have no plan in the immediate future to investigate any OS other than Linux, so this article will cover this OS only; or, stated in another way, there is a potential **loss of portability**.

We can summarize saying that “*typesetting in luatex-lunatic is here and now*”, where *here* stands for “a specific OS” and *now* for “with this release”. Actually *here* stands for “Linux 32 bit”, and *now* stands for `luatex-snapshot-0.42.0.tar.bz2` with ConTeXt-mkiv current 2008.07.17; probably both will already be old by the time this is printed.

Another motivation has emerged during the development of `luatex-lunatic`: the possibility to use ConTeXt-mkiv as a sort of literate programming tool for a specific context.

It is well known that CWEB is a way to tangle together a program written in a *specific* programming language (C) with its documentation written with a macro markup language, TeX; `luatex-lunatic` and ConTeXt-mkiv can be used to tangle together a program written in an (almost) *arbitrary* programming language with its documentation written with a *high level* macro markup language, ConTeXt-mkiv.

Put in another way: currently an application calls TeX or LaTeX (i.e. it creates a process) to obtain a result from a tex snippet (for example to show a math formula); instead `luatex-lunatic` with ConTeXt-mkiv calls the application by dynamic loading (i.e. it does not create a process) to obtain the result to insert into tex source.

For example one can use `luatex-lunatic` ConTeXt-mkiv to typeset a math formula, and the binding for the evaluation of the same formula (there are several symbolic-math Python modules already available).

We will see more about this later, when we will talk of Sage.

We want to find the smallest set of patches of the `luatex` codebase, or, better, we want to avoid:

1. constraints of any sort to the development team;
2. massive modifications of the code base;
3. radical modification of the building process.

Lunatic Python

There is no better site than [14] to explain what is Lunatic Python:

Lunatic Python is a two-way bridge between Python and Lua, allowing these languages to intercommunicate. Being two-way means that it allows Lua inside Python, Python inside Lua, Lua inside Python inside Lua, Python inside Lua inside Python, and so on.

...

The bridging mechanism consists of creating the missing interpreter state inside the host interpreter. That is, when you run the bridging system inside Python, a Lua interpreter is created; when you run the system inside Lua, a Python interpreter is created.

Once both interpreter states are available, these interpreters are provided with the necessary tools to interact freely with each other. The given tools offer not only the ability of executing statements inside the alien interpreter, but also to acquire individual objects and interact with them inside the native state. This magic is done by two special object types, which act by bridging native object access to the alien interpreter state.

Almost every object which is passed between Python and Lua is encapsulated in the language specific bridging object type. The only types which are not encapsulated are strings and numbers, which are converted to the native equivalent objects.

Besides that, the Lua side also has special treatment for encapsulated Python functions and methods. The most obvious way to implement calling of Python objects inside the Lua interpreter is to implement a `__call` function in the bridging object metatable. Unfortunately this mechanism is not supported in certain situations, since some places test if the object type is a function, which is not the case of the bridging object. To overwhelm these problems, Python functions and methods are automatically converted to native Lua function closures, becoming accessible in every Lua context. Callable object instances which are not functions nor methods, on the other hand, will still use the metatable mechanism. Luckily, they may also be converted in a native function closure using the `asfunc()` function, if necessary.



According to [68], page 47, a *closure* is “a function plus all it needs to access non-local variables correctly”; a non-local variable “is neither a global variable nor a local variable”. For example consider `newCounter`:

```
function newCounter()
  local i = 0
  return function()
    i = i+1
    return i
  end
end
c1 = newCounter()
print(c1()) --> 1
print(c1()) --> 2
c2 = newCounter()
print(c2()) --> 1
print(c1()) --> 3
print(c2()) --> 2
```

`i` is a non-local variable; we see that there is no interference between `c1` and `c2`—they are two different closures over the same function.

It's better to track a layout of installation of `luatex-lunatic` on a Linux box.

Let's set up a home directory:

```
HOMEDIR=/opt/luatex/luatex-lunatic
```

Next:

1. download and install `python-2.6.1` (at least) from [49]. Assuming `$HOMEDIR/Python-2.6.1` as build directory, let's configure `python-2.6.1` with

```
./configure
  --prefix=/opt/luatex/luatex-lunatic
  --enable-unicode=ucs4
  --enable-shared
```

and install it. After installation we should end in a "Filesystem Hierarchy Standard"-like Filesystem (cf. [46], except for `Python-2.6.1`), i.e. something like this:

```
$> cd $HOMEDIR && ls -lX
bin
include
lib
man
share
Python-2.6.1
```

It's also convenient to extend the system path:

```
$> export PATH=
/opt/luatex/lunatic-python/bin:$PATH
```

so we will use the `python` interpreter in `$HOMEDIR`.

2. download `luatex` source code from [43]; we will use `luatex-snapshot-0.42.0`, so let's unpack it in `$HOMEDIR/luatex-snapshot-0.42.0`. For uniformity, make a symbolic link

```
$> cd $HOMEDIR
$> ln -s luatex-snapshot-0.42.0 luatex
```

It's convenient to have a stable `ConTeXt` `minimals` distribution installed (cf. [23]) under `$HOMEDIR`, i.e. `$HOMEDIR/minimals`, so that we will replace its `luatex` with our `luatex-lunatic`. Remember to set up the environment with

```
$> . $HOMEDIR/minimals/tex/setuptex
```

We don't build it now, because `build.sh` needs to be patched.

3. download `luatex-lunatic` from [3], revision 7, and put it in `lunatic-python`, i.e.

```
$> cd $HOMEDIR
$> bzip branch lp:lunatic-python
```

We must modify `setup.py` to match `luatex` installation (here "<" stands for the original `setup.py`, ">" stands for the modified one; it's a diff file):

```
1c1
< #!/usr/bin/python
---
> #!/opt/luatex/luatex-lunatic/bin/python
14,16c14,16
< LUALIBS = ["lua5.1"]
< LUALIBDIR = []
< LUAINCDIR = glob.glob("/usr/include/lua*")
---
> LUALIBS = ["lua51"]
> LUALIBDIR = ['/opt/luatex/
  luatex-lunatic/
  luatex/build/texk/web2c']
> LUAINCDIR = glob.glob("../
  luatex/source/texk/web2c/luatexdir/lua51*")
48a49
>
```

When we build `lunatic-python`, we will end with a `python.so` shared object that will be installed in the `$HOMEDIR/lib/python2.6/site-packages` directory, so it's convenient to prepare a `python.lua` wrapper like this one:

```
loaded = false
func = package.loadlib(
"/opt/luatex/luatex-lunatic/lib/python2.6/
site-packages/python.so", "luaopen_python")
if func then
  func()
  return
end
if not loaded then
  error("unable to find python module")
end
```

Before building, we must resolve the dynamic loading problem; again from [14]

... Unlike Python, Lua has no default path to its modules. Thus, the default path of the real Lua module of `Lunatic Python` is together with the Python module, and a `python.lua` stub is provided. This stub must be placed in a path accessible by the `Lua require()` mechanism, and once imported it will locate the real module and load it.

Unfortunately, there's a minor inconvenience for our purposes regarding the Lua system which imports external shared objects. The hardcoded behavior of the

loadlib() function is to load shared objects without exporting their symbols. This is usually not a problem in the Lua world, but we're going a little beyond their usual requirements here. We're loading the Python interpreter as a shared object, and the Python interpreter may load its own external modules which are compiled as shared objects as well, and these will want to link back to the symbols in the Python interpreter. Luckily, fixing this problem is easier than explaining the problem. It's just a matter of replacing the flag RTLD_NOW in the loadlib.c file of the Lua distribution by the or'ed version RTLD_NOW|RTLD_GLOBAL. This will avoid "undefined symbol" errors which could eventually happen.

Modifying luatex/source/tekk/web2c/
 luatexdir/lu51/loadlib.c
 is not difficult:

```
69c69
< void *lib = dlopen(path, RTLD_NOW);
---
> void *lib = dlopen(path, RTLD_NOW|RTLD_GLOBAL);
```

(again "<" means original and ">" means modified).



According to dlopen(3) - Linux man page (see for example [18]),

The function dlopen() loads the dynamic library file named by the null-terminated string filename and returns an opaque "handle" for the dynamic library. If filename is NULL, then the returned handle is for the main program. If filename contains a slash ("/"), then it is interpreted as a (relative or absolute) pathname. Otherwise, the dynamic linker searches for the library as follows (see ld.so(8) for further details):

- (ELF only) If the executable file for the calling program contains a DT_RPATH tag, and does not contain a DT_RUNPATH tag, then the directories listed in the DT_RPATH tag are searched.
- If the environment variable LD_LIBRARY_PATH is defined to contain a colon-separated list of directories, then these are searched. (As a security measure this variable is ignored for set-user-ID and set-group-ID programs.)
- (ELF only) If the executable file for the calling program contains a DT_RUNPATH tag, then the directories listed in that tag are searched.
- The cache file /etc/ld.so.cache (maintained by ldconfig(8)) is checked to see whether it contains an entry for filename.
- The directories /lib and /usr/lib are searched (in that order).

If the library has dependencies on other shared libraries, then these are also automatically loaded by the dynamic linker using the same rules. (This process may occur recursively, if those libraries in turn have dependencies, and so on.)

One of the following two values must be included in flag:

- RTLD_LAZY
 Perform lazy binding. Only resolve symbols as the code that refer-

ences them is executed. If the symbol is never referenced, then it is never resolved. (Lazy binding is only performed for function references; references to variables are always immediately bound when the library is loaded.)

- RTLD_NOW
 If this value is specified, or the environment variable LD_BIND_NOW is set to a non-empty string, all undefined symbols in the library are resolved before dlopen() returns. If this cannot be done, an error is returned.

Zero or more of the following values may also be ORed in flag:

- RTLD_GLOBAL
 The symbols defined by this library will be made available for symbol resolution of subsequently loaded libraries.
- RTLD_LOCAL
 This is the converse of RTLD_GLOBAL, and the default if neither flag is specified. Symbols defined in this library are not made available to resolve references in subsequently loaded libraries.
- RTLD_NODELETE (since glibc 2.2)
 Do not unload the library during dlclose(). Consequently, the library's static variables are not reinitialized if the library is reloaded with dlopen() at a later time. This flag is not specified in POSIX.1-2001.
- RTLD_NOLOAD (since glibc 2.2)
 Don't load the library. This can be used to test if the library is already resident (dlopen() returns NULL if it is not, or the library's handle if it is resident). This flag can also be used to promote the flags on a library that is already loaded. For example, a library that was previously loaded with RTLD_LOCAL can be re-opened with RTLD_NOLOAD | RTLD_GLOBAL. This flag is not specified in POSIX.1-2001.
- RTLD_DEEPBIND (since glibc 2.3.4)
 Place the lookup scope of the symbols in this library ahead of the global scope. This means that a self-contained library will use its own symbols in preference to global symbols with the same name contained in libraries that have already been loaded. This flag is not specified in POSIX.1-2001.

If filename is a NULL pointer, then the returned handle is for the main program. When given to dlsym(), this handle causes a search for a symbol in the main program, followed by all shared libraries loaded at program startup, and then all shared libraries loaded by dlopen() with the flag RTLD_GLOBAL.

External references in the library are resolved using the libraries in that library's dependency list and any other libraries previously opened with the RTLD_GLOBAL flag. If the executable was linked with the flag "-rdynamic" (or, synonymously, "-export-dynamic"), then the global symbols in the executable will also be used to resolve references in a dynamically loaded library.

If the same library is loaded again with dlopen(), the same file handle is returned. The dl library maintains reference counts for library handles, so a dynamic library is not deallocated until dlclose() has been called on it as many times as dlopen() has succeeded on it. The _init routine, if present, is only called once. But a subsequent call with RTLD_NOW may force symbol resolution for a library earlier loaded with RTLD_LAZY.

If dlopen() fails for any reason, it returns NULL.

Nevertheless this is not enough: reference manual [66] says (page 23):

Dynamic loading of .so and .dll files is disabled on all platforms.

So we must “enable” it and we must ensure that the `luatex` executable is linked against `libdl.so` because this contains the `dlopen()` symbol; also we must ensure that all the Lua functions involved in a `dlopen()` call must be resolved in the `luatex-lunatic` executable.

Assuming that we are always in `$HOMEDIR`, we must modify

```
source/teXk/web2c/luatexdir/am/liblua51.am
and source/teXk/web2c/Makefile.in.
```

For

```
source/teXk/web2c/luatexdir/am/liblua51.am:
```

```
12c12
```

```
< liblua51_a_CPPFLAGS += -DLUA_USE_POSIX
```

```
---
```

```
> liblua51_a_CPPFLAGS += -DLUA_USE_LINUX
```

```
while for source/teXk/web2c/Makefile.in:
```

```
98c98
```

```
< @MINGW32_FALSE@am__append_14 = -DLUA_USE_POSIX
```

```
---
```

```
> @MINGW32_FALSE@am__append_14 = -DLUA_USE_LINUX
1674c1674
```

```
< $(CXXLINK) $(luatex_OBJECTS) $(luatex_LDADD)
$(LIBS)
```

```
---
```

```
> $(CXXLINK) $(luatex_OBJECTS) $(luatex_LDADD)
$(LIBS) -Wl,-E -uluaL_openlibs -fvisibility=hidden
en -fvisibility-inlines-hidden -ldl
```

The last patch is the most important, so let’s examine it more closely. Essentially, we are modifying the linking phase of building process of `luatex` (switch `-Wl,-E`) by adding `libdl` (switch `-ldl`) because `libdl` contains the symbol `dlopen` as stated before.

The switch `-uluaL_openlibs` tells the linker to consider the symbol `luaL_openlibs` even if it’s not necessary for building `luatex-lunatic`. In fact `luaL_openlibs` is coded in `lunatic-python/src/luainpython.c` and it needs to be resolved at runtime only when `luatex-lunatic` wants to load the Python interpreter.

So, even if `luaL_openlibs` is a function coded in `$HOMEDIR/luatex/source/teXk/web2c/luatexdir/lu51/limit.c`, it’s not used by `luatex`, so the linker discards this symbol because it’s useless.



According to `ld(1)`:

□ `-u` symbol

Force symbol to be entered in the output file as an undefined symbol. Doing this may, for example, trigger linking of additional modules from standard libraries. `-u` may be repeated with different option arguments to enter additional undefined symbols. This option is equivalent to the “`EXTERN`” linker script command.



It’s possible to examine how symbols are resolved runtime by setting `LD_DEBUG=all`; for example

```
$> export LD_DEBUG=all;
$> luatex python.lua &>python.LD_DEBUG;
$> export LD_DEBUG=
```

Here we are assuming a correct final `luatex lunatic luatex` and the `python.lua` wrapper seen before.

The file `python.LD_DEBUG` will show something like this:

```
3736: symbol=luaL_openlibs;
      lookup in file=./luatex-lunatic [0]
3736: binding file /opt/luatex/luatex-lunatic/
      lib/python2.6/site-packages/python.so [0]
      to ./luatex-lunatic [0]:
      normal symbol ‘luaL_openlibs’
```


Without the `-uluaL_openlibs` linker flag, we will see something like this:

```
4033: symbol=luaL_openlibs;
      lookup in file=./luatex-lunatic-0.42.0.-test [0]
4033: symbol=luaL_openlibs;
      lookup in file=/lib/tls/i686/cmox/libm.so.6 [0]
4033: symbol=luaL_openlibs;
      lookup in file=/lib/tls/i686/cmox/libdl.so.2 [0]
4033: symbol=luaL_openlibs;
      lookup in file=/lib/libreadline.so.5 [0]
4033: symbol=luaL_openlibs;
      lookup in file=/lib/libhistory.so.5 [0]
4033: symbol=luaL_openlibs;
      lookup in file=/lib/libncurses.so.5 [0]
4033: symbol=luaL_openlibs;
      lookup in file=/lib/tls/i686/cmox/libc.so.6 [0]
4033: symbol=luaL_openlibs;
      lookup in file=/lib/ld-linux.so.2 [0]
4033: symbol=luaL_openlibs;
      lookup in file=/opt/luatex/luatex-lunatic/lib/
      python2.6/site-packages/python.so [0]
4033: symbol=luaL_openlibs;
      lookup in file=/lib/tls/i686/cmox/libpthread.so.0 [0]
4033: symbol=luaL_openlibs;
      lookup in file=/lib/tls/i686/cmox/libutil.so.1 [0]
4033: symbol=luaL_openlibs;
      lookup in file=/lib/tls/i686/cmox/libc.so.6 [0]
4033: symbol=luaL_openlibs;
      lookup in file=/lib/ld-linux.so.2 [0]
4033: /opt/luatex/luatex-lunatic/lib/python2.6/
      site-packages/python.so:
      error: symbol lookup error:
      undefined symbol: luaL_openlibs (fatal)
4033:
4033: file=/opt/luatex/luatex-lunatic/lib/python2.6/
      site-packages/python.so [0]; destroying link map
```

And near the bottom we can see this error: `symbol lookup error: undefined symbol: luaL_openlibs (fatal)`.

The last two switches, namely `-fvisibility=hidden` and `-fvisibility-inlines-hidden`, are `gcc` switches (not linker switches) and again they are related with

symbols, more precisely with symbols collisions. Consider this: in \$HOMEDIR/luatex/source/libs/libpng there is a libpng library (currently vers. 1.2.38). This library, once compiled, will be merged by the linker into the luatex executable, and hence into the luatex-lunatic executable too. Now, we can build a Python binding to another libpng library or, better, we can import a Python module (e.g. PythonMagickWand, an interface to ImageMagick[®], see [40]) that has a binding to its own libpng library. In this situation, at runtime the dynamic loader will resolve for the Python module the symbols of libpng from luatex libpng, instead of those from its own libpng. Now, we cannot guarantee that these two libraries are the same, because we cannot replace the libpng of luatex (see near the end of the preceding section “Motivation & goals”) and, of course, we cannot replace the libpng library from the Python module with the one from luatex, because the last one can be patched for luatex only. So, we have symbols collisions (see [9]): almost for sure, a symbol collision will cause a segmentation fault, and the program abort.

 More information about this can be found starting from the already cited [9], especially [69]. A good text is also [63].

A solution can be this: “hide” to the “outside” all symbols that aren’t necessary for dynamic loading of shared objects. For standard luatex, this means “hide all”: for luatex-lunatic, this means “hide all but not symbols from lua”, otherwise we will not be able to use loadlib. It’s not so difficult to “hide all”: just patch the build.sh script of luatex sources by adding

```
28a29,36
> CFLAGS="-g -O2 -Wno-write-strings
      -fvisibility=hidden"
> CXXFLAGS="$CFLAGS
      -fvisibility-inlines-hidden"
> export CFLAGS
> export CXXFLAGS
```

The hardest part is to “unhide” the Lua part. We can proceed in this manner: collect the result of the patched build.sh in an out file:

```
$> cd $HOMEDIR/luatex; ./build.sh &> out
```

Then locate in out *all* the lines about Lua and remove the -fvisibility=hidden flag: for example

```
gcc -DHAVE_CONFIG_H -I.
-I../../../../source/teXk/web2c -I../
-I/opt/luatex/luatex-lunatic/
  luatex-snapshot-0.42.0/build/teXk
-I/opt/luatex/luatex-lunatic/
```

```
  luatex-snapshot-0.42.0/source/teXk
-I../../../../source/teXk/web2c/luatexdir/lu51
-DLUA_USE_LINUX -g -O2
-Wno-write-strings
-fvisibility=hidden
-Wdeclaration-after-statement
-MT liblua51_a-lapi.o
-MD -MP -MF .deps/liblua51_a-lapi.Tpo
-c -o liblua51_a-lapi.o
'test -f
'luatexdir/lu51/lapi.c'
|| echo
'../../../../source/teXk/web2c/'
  luatexdir/lu51/lapi.c
mv -f .deps/liblua51_a-lapi.Tpo
  .deps/liblua51_a-lapi.Po
```

will become

```
gcc -DHAVE_CONFIG_H -I.
-I../../../../source/teXk/web2c -I../
-I/opt/luatex/luatex-lunatic/
  luatex-snapshot-0.42.0/build/teXk
-I/opt/luatex/luatex-lunatic/
  luatex-snapshot-0.42.0/source/teXk
-I../../../../source/teXk/web2c/luatexdir/lu51
-DLUA_USE_LINUX
-g -O2 -Wno-write-strings
-Wdeclaration-after-statement
-MT liblua51_a-lapi.o
-MD -MP -MF .deps/liblua51_a-lapi.Tpo
-c -o liblua51_a-lapi.o
'test -f
'luatexdir/lu51/lapi.c'
|| echo
'../../../../source/teXk/web2c/
'luatexdir/lu51/lapi.c
mv -f .deps/liblua51_a-lapi.Tpo
  .deps/liblua51_a-lapi.Po
```

After that, recompile luatex

```
/bin/bash ./libtool
--tag=CXX
--mode=link
./CXXLD.sh -g -O2
-Wno-write-strings
-fvisibility=hidden
-fvisibility-inlines-hidden
-o luatex
luatex-luatex.o
libluatex.a libbff.a
libluamisc.a libzip.a
libluasocket.a liblua51.a
/opt/luatex/luatex-lunatic/
```

```

    luatex-snapshot-0.42.0/build/libs/
    libpng/libpng.a
/opt/luatex/luatex-lunatic/
    luatex-snapshot-0.42.0/build/libs/
    zlib/libz.a
/opt/luatex/luatex-lunatic/
    luatex-snapshot-0.42.0/build/libs/
    xpdf/libxpdf.a
/opt/luatex/luatex-lunatic/
    luatex-snapshot-0.42.0/build/libs/
    obsdcompat/libopenbsd-compat.a
libmd5.a libmplib.a
lib/lib.a
/opt/luatex/luatex-lunatic/
    luatex-snapshot-0.42.0/build/texk/
    kpathsea/libkpathsea.la
-lm -Wl,-E
-uluaL_openlibs
-fvisibility=hidden
-fvisibility-inlines=hidden
-ldl

```

Of course it's better to edit a `trick.sh` from out (see [34]) that will do all the work, paying the price of ~20 minutes of cut and paste for every new luatex release for preparing this trick file.

After executing `$HOMEDIR/luatex/trick.sh` we will have an *unstripped* luatex binary in `$HOMEDIR/luatex/build/texk/web2c` so we are ready for the final step. It's better not to strip it, because we can track problems more easily.

- we copy luatex into the bin directory of ConTeXt minimals and remade formats:

```

$> cp $HOMEDIR/luatex/build/texk/web2c/luatex
    $HOMEDIR/minimals/tex/texmf-linux/bin
$> context --make

```

And in the end we must build the `lunatic-python` shared object:

```

$> cp $HOMEDIR/lunatic-python
$> python setup.py build && python setup.py
    install

```



We can now make a simple test; let's save this in `test.tex`:

```

\directlua{require "python";
sys = python.import("sys");
tex.print(tostring(sys.version_info))}
\bye

```

Next let's run `callgrind`, a tool of `valgrind` (see [30]), to generate a *call graph* [5]:

```

$> valgrind --tool=callgrind
    --callgrind-out-file=test-%p.callgrind
    --dump-instr=yes
    luatex --fmt=plain --output-format=pdf test.tex

```

To see and analyze this call graph we can use `kcachegrind` [13]: see appendix at page 53 for the graph centered at `main` function, with `Min. node cost=1%`, `Min. call cost=1%`.

Examples

Image processing

ImageMagick. *ImageMagick* is “a software suite to create, edit, and compose bitmap images. It can read, convert and write images in a variety of formats (over 100) including DPX, EXR, GIF, JPEG, JPEG-2000, PDF, PhotoCD, PNG, PostScript, SVG, and TIFF. Use *ImageMagick* to translate, flip, mirror, rotate, scale, shear and transform images, adjust image colors, apply various special effects, or draw text, lines, polygons, ellipses and Bézier curves.” (See [40].) There are two bindings in Python, and we choose the `PythonMagickWand` [48], a ctypes-based wrapper for *ImageMagick*.



According to [50] ctypes is a foreign function library for Python. It provides C compatible data types, and allows calling functions in DLLs or shared libraries. It can be used to wrap these libraries in pure Python. ctypes is included in Python.

This simple script create a 200×200 pixel image at 300dpi with a shadow:

```

import PythonMagickWand as pmw
pmw.MagickWandGenesis()
wand = pmw.NewMagickWand()
background = pmw.NewPixelWand(0)
pmw.MagickNewImage(wand,200,200,background)
pmw.MagickSetImageResolution(wand,118.110,118.110)
pmw.MagickSetImageUnits(wand,
    pmw.PixelsPerCentimeterResolution)
pmw.MagickShadowImage(wand,90,3,2,2)
pmw.MagickWriteImage(wand,"out.png")

```

i.e., something like this:



Suppose we want to use it to generate a background for text, i.e.

```
\startShadowtext%
\input tufte
\stopShadowtext%
```

Let's now look at luatex lunatic and ConTeXt-mkiv in action for the first time:

```
\usetyescriptfile[type-gentium]
\usetyescript[gentium]
\setupbodyfont[gentium,10pt]
\setuppapersize[A6][A6]
\setuplayout[height=middle,topspace=1cm,
  header={2\lineheight},footer=0pt,backspace=1cm,
  margin=1cm,width=middle]
%%
%% lua layer
%%
\startluacode
function testimagemagick(box,t)
  local w
  local h
  local d
  local f
  local res = 118.11023622047244094488 -- 300 dpi
  local opacity = 25
  local sigma = 15
  local x = 10
  local y = 10
  w = math.floor((tex.wd[box]/65536 )
    /72.27*2.54*res)
  h = math.floor(((tex.ht[box]/65536)+
    (tex.dp[box]/65536))
    /72.27*2.54*res)
  f = string.format("%s.png", t)
  --
  -- Call the python interpreter
  --
  require("python")
  pmw = python.import("PythonMagickWand")
  wand = pmw.NewMagickWand()
  background = pmw.NewPixelWand(0)
  pmw.MagickNewImage(wand,w,h,background)
  pmw.MagickSetImageResolution(wand,res,res)
  pmw.MagickSetImageUnits(wand,
    pmw.PixelsPerCentimeterResolution)
  pmw.MagickShadowImage(wand,opacity,sigma,x,y)
  pmw.MagickWriteImage(wand ,f)
end
\stopluacode
%%
%% TeX layer
%%
```

```
\def\testimagemagick[#1]{%
\getparameters[Imgk][#1]%
\ctxlua{%
  testimagemagick(\csname Imgkbox\endcsname,
    "\csname Imgkfilename\endcsname")}%
}
%%
%% ConTeXt layer
%%
\newcount\shdw
\long\def\startShadowtext#1\stopShadowtext{%
\bgroupp%
\setbox0=\vbox{#1}%
\testimagemagick[box=0,
  filename={shd-\the\shdw}]%
\defineoverlay[backg]%
  [{\externalfigure[shd-\the\shdw.png]}]%
\framed[background=backg,
  frame=off,offset=4pt]{\box0}%
\global\advance\shdw by 1%
\egroupp%
}
\starttext
\startTEXpage%
\startShadowtext%
\input tufte
\stopShadowtext%
\stopTEXpage
\stoptext
```

As we can see, there is an almost one-to-one mapping between Python code and Lua code, a good thing for a small script.

And here is the result:

We thrive in information—thick worlds because of our marvelous and everyday capacity to select, edit, single out, structure, highlight, group, pair, merge, harmonize, synthesize, focus, organize, condense, reduce, boil down, choose, categorize, catalog, classify, list, abstract, scan, look into, idealize, isolate, discriminate, distinguish, screen, pigeon-hole, pick over, sort, integrate, blend, inspect, filter, lump, skip, smooth, chunk, average, approximate, cluster, aggregate, outline, summarize, itemize, review, dip into, flip through, browse, glance into, leaf through, skim, refine, enumerate, glean, synopsisize, winnow the wheat from the chaff and separate the sheep from the goats.



What about symbols collisions?

```
$> eu-readelf --all luatex &> luatex.dump
$> export LD_DEBUG=all;context test-imagemagick.tex &> test-
  imagemagick.tex.LD_DEBUG; export LD_DEBUG=
```

If we search png_memcpy_check which is coded in \$HOMEDIR/source/libs/libpng/libpng-1.2.38/pngmem.c of luatex, we will find that

it's bound to system libpng:

```
25749: symbol=png_memcpy_check;
  lookup in file=luatex [0]
25749: symbol=png_memcpy_check;
  lookup in file=/lib/tls/i686/cmov/libm.so.6 [0]
25749: symbol=png_memcpy_check;
  lookup in file=/lib/tls/i686/cmov/libdl.so.2 [0]
:
: (62 lines after)
:
25749: symbol=png_memcpy_check;
  lookup in file=/usr/lib/libpng12.so.0 [0]
25749: binding file /usr/lib/libpng12.so.0 [0]
to /usr/lib/libpng12.so.0 [0]:
normal symbol 'png_memcpy_check' [PNG12_0]
```

In fact if we search for `png_memcpy_check` in `luatex.dump` we see that it's hidden now:

```
Symbol table [40] '.symtab' contains 10087 entries:
 9592 local symbols String table: [41] '.strtab'
:
Num:   Value   Size Type
4837: 082022b0  27 FUNC

Bind  Vis    Ndx Name
LOCAL  HIDDEN  13  png_memcpy_check
:
```

As a counterexample, suppose that we don't use hidden flags, so now `png_memcpy_check` is visible:

```
Num:   Value   Size Type
2273: 08243050  27 FUNC

Bind  Vis    Ndx Name
GLOBAL  DEFAULT  13  png_memcpy_check
```

Now we have a fatal error:

```
$> export LD_DEBUG=all;context test-imagemagick.tex &> test-
imagemagick.tex.LD_DEBUG; export LD_DEBUG=
:
MTXrun | fatal error, no return code, message: luatex: execu-
tion interrupted
:
```

and we see that `png_memcpy_check` is resolved in `luatex`:

```
24213: symbol=png_memcpy_check;
  lookup in file=luatex [0]
24213: binding file /usr/lib/libpng12.so.0 [0]
to luatex [0]:
normal symbol 'png_memcpy_check' [PNG12_0]
```

so we have symbols collisions. In this case it can be hard to track the guilty symbol; even in this case the fatal error can be given by another symbols collision, not necessarily `png_memcpy_check`. Also note that this code

```
\starttext
```

```
\externalfigure[out.png]
\stoptext
```

compiles right—of course, because there is no `PythonImageMagickWand` involved and so no symbols collisions. So this kind of error can become a nightmare.

Let's continue with our gallery.

PIL – PythonImageLibrary. PIL (see [51]) is similar to `ImageMagick`, but at least for `png` doesn't require `libpng`, so we are safe from symbol collisions.

```
\startluacode
function testPIL(imageorig,imagesepia)
  require("python")
  PIL_Image = python.import("PIL.Image")
  PIL_ImageOps = python.import("PIL.ImageOps")
  python.execute([[
def make_linear_ramp(white):
  ramp = []
  r, g, b = white
  for i in range(255):
    ramp.extend((r*i/255, g*i/255, b*i/255))
  return ramp
]])
  -- make sepia ramp
  -- (tweak color as necessary)
  sepia = python.eval
    ("make_linear_ramp((255, 240, 192))")
  im = PIL_Image.open(imageorig)
  -- convert to grayscale
  if not(im.mode == "L")
  then
    im = im.convert("L")
  end
  -- optional: apply contrast
  -- enhancement here, e.g.
  im = PIL_ImageOps.autocontrast(im)
  -- apply sepia palette
  im.putpalette(sepia)
  -- convert back to RGB
  -- so we can save it as JPEG
  -- (alternatively, save it in PNG or similar)
  im = im.convert("RGB")
  im.save(imagesepia)
end
\stoptluacode

\def\SepiaImage#1#2{%
\ctxlua{testPIL("#1", "#2")}%
\startcombination[1*2]
{\externalfigure[#1][width=512pt]}\ss Orig.}
{\externalfigure[#2][width=512pt]}\ss Sepia}
\stopcombination
}
```

```
\starttext
\startTEXpage
%\SepiaImage{lena.jpg}{lena-sepia.jpg}
\SepiaImage{lena.png}{lena-sepia.png}
\stopTEXpage
\stoptext
```

Here is the result (sorry, Lena is too nice to show her only in black and white):



The code shows how to define a Python function inside a Lua function and how to call it. Note that we must respect the Python indentation rules, so we can use the multiline string of Lua [[. .]].

Language adapter

Suppose we have a C library for a format of a file (i.e. TIFF, PostScript) that we want to manage in the same way as png, pdf, jpeg and jbig. One solution is to build a quick binding with ctypes of Python, and then import it

in luatex-lunatic as a traditional Lua module. As an example, let's consider ghostscript [10], here in vers. 8.64. It can be compiled as a shared library, and building a testgs.py (see [35]#Ghostscript) binding is not difficult (see file base/gslib.c in source distribution). The key here is to build a binding that fits our needs, not a general one.

```
\startluacode
function testgs(epsin,pdfout)
  require("python")
  gsmodule = python.import("testgs")
  ghost = gsmodule.gs()
  ghost.appendargs('-q')
  ghost.appendargs('-dNOPAUSE')
  ghost.appendargs('-dEPSCrop')
  ghost.appendargs('-sDEVICE=pdfwrite')
  ghost.InFile = epsin
  ghost.OutFile = pdfout
  ghost.run()
end
\stopluacode

\def\epstopdf#1#2{\ctxlua{testgs("#1","#2")}}
\def\EPSfigure[#1]{%lazy way to load eps
\epstopdf{#1.eps}{#1.pdf}%
\externalfigure[#1.pdf]}
```

```
\starttext
\startTEXpage
{\EPSfigure[golfer]}
{\ss golfer.eps}
\stopTEXpage
\stoptext
```

Here is the result:



We can also use PostScript libraries: for example barcode.ps [56], a PostScript barcode library:

```
\startluacode
function epstopdf(epsin,pdfout)
  require("python")
  gsmodule = python.import("testgs")
  ghost = gsmodule.gs()
  ghost.appendargs('-q')
```

```

ghost.appendargs('-dNOPAUSE')
ghost.appendargs('-dEPSCrop')
ghost.appendargs('-sDEVICE=pdfwrite')
ghost.InFile = epsin
ghost.OutFile = pdfout
ghost.run()
end
function barcode(text,type,options,savefile)
require("python")
gsmodule = python.import("testgs")
barcode_string =
string.format("%! \n100 100 moveto (%s) (%s)
%s barcode showpage",
text,options,type)
psfile = string.format("%s.ps",savefile)
epsfile = string.format("%s.eps",savefile)
pdffile = string.format("%s.pdf",savefile)
temp = io.open(psfile,'w')
print(psfile)
temp.write(tostring(barcode_string),"\n")
temp:flush()
io.close(temp)
ghost = gsmodule.gs()
ghost.rawappendargs('-q')
ghost.rawappendargs('-dNOPAUSE')
ghost.rawappendargs('-sDEVICE=epswrite')
ghost.rawappendargs(
string.format('-sOutputFile=%s',epsfile))
ghost.rawappendargs('barcode.ps')
ghost.InFile= psfile
ghost.run()
end
\stopluacode

\def\epstopdf#1#2{\ctxlua{epstopdf("#1","#2")}}
\def\EPSfigure[#1]{%lazy way to load eps
\epstopdf{#1.eps}{#1.pdf}%
\externalfigure[#1.pdf]%
}

\def\PutBarcode[#1]{%
\getparameters[bc][#1]%
\ctxlua{barcode("\csname bctext\endcsname",
"\csname bctype\endcsname",
"\csname bcoptions\endcsname",
"\csname bcsavefile\endcsname" )}%
\expanded{\EPSfigure
[\csname bcsavefile\endcsname]}%
}

\starttext
\startTEXpage
{\PutBarcode[text={CODE 39},type={code39},
options={includecheck includetext},
savefile={TEMP1}]\}

```

```

{\ss code39}
\blank
{\PutBarcode[text={CONTEXT},type={code93},
options={includecheck includetext},
savefile={TEMP2}]\}
{\ss code93}
\blank
{\PutBarcode[text={977147396801},type={ean13},
options={includetext},
savefile={TEMP3}]\}
{\ss ean13}
\stopTEXpage
\stoptext

```

Of course one can implement a direct conversion into ps->pdf, instead of ps->eps->pdf.

Here is the result:

For a beautiful book on PostScript see [58] (and its site [42]) and also [2].

Scientific & math extensions

Sage. “*Sage is a free open-source mathematics software system licensed under the GPL. It combines the power of many existing open-source packages into a common Python-based interface. Mission: Creating a viable free open source alternative to Magma, Maple, Mathematica and Matlab.*” [53]

Given that Sage comes with its own Python interpreter, we must rebuild lunatic-python and adapt python.lua accordingly; also sage is a command line program, so we need a stub `sagestub.py`:

```
from sage.all_cmdline import *
```

Here is the ConTeXt-mkiv code:

```

\startluacode
function test_ode(graphout)
require("python")
pg = python.globals()
SAGESTUB = python.import("sagestub")
sage = SAGESTUB.sage
python.execute([[
def f_1(t,y,params):
return[y[1],
-y[0]-params[0]*y[1]*(y[0]**2-1)]
]])
python.execute([[
def j_1(t,y,params):
return [ [0,1.0],
[-2.0*params[0]*y[0]*y[1]-1.0,
-params[0]*(y[0]*y[0]-1.0)], [0,0]
]])

```

```

T=sage.gsl.ode.ode_solver()
T.algorithm="rk8pd"
f_1 = pg.f_1
j_1 = pg.j_1
pg.T=T
python.execute("T.function=f_1")
T.jacobian=j_1
python.execute("T.ode_solve(y_0=[1,0],
                    t_span=[0,100],
                    params=[10],num_points=1000)")
python.execute(string.format(
    "T.plot_solution(filename='%s')",
    graphout ))
end
\stopluacode

\def\TestODE#1{%
\ctxlua{test_ode("#1")}%
\startcombination[1*2]
{%
\ vbox{\hsize=8cm
Consider solving the Van der Pol oscillator
 $x'(t) + ux'(t)(x(t)^2-1) + x(t) = 0$ 
between  $t=0$  and  $t= 100$ .
As a first order system it is
 $x'=y$ 
 $y'=-x+uy(1-x^2)$ 
Let us take  $u=10$  and use
initial conditions  $(x,y)=(1,0)$  and use the
\emphsl{\hbox{Runge-Kutta} \hbox{Prince-Dormand}}
algorithm.
}%
}{\ss \ }
{\externalfigure[#1][width=9cm]}{\ss Result
for 1000 points}}

\starttext
\startTEXpage
\TestODE{ode1.pdf}
\stopTEXpage
\stoptext

```

As we can see, here we use the `python.globals()` Lua function to communicate between the Python interpreter and Lua, and this can generate a bit of useless redundancy.

R. R “is a free software environment for statistical computing and graphics” [52]. It has its own language, but there is also a Python binding, `rpy2` [27], that we install in our `$HOMEDIR`.



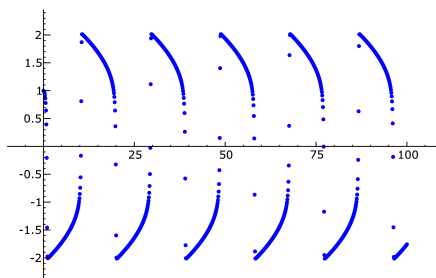
It can be necessary to add these env. variables

```

$>export R_HOME=/opt/luatex/luatex-lunatic/lib/R
$>export LD_PRELOAD=/opt/luatex/
          luatex-lunatic/lib/R/lib/libR.so

```

Consider solving the Van der Pol oscillator $x''(t) + ux'(t)(x(t)^2 - 1) + x(t) = 0$ between $t = 0$ and $t = 100$. As a first order system it is $x' = y$ $y' = -x + uy(1 - x^2)$ Let us take $u = 10$ and use initial conditions $(x,y) = (1,0)$ and use the *Runge-Kutta Prince-Dormand* algorithm.



Result for 1000 points

Figure 1. Result of the Sage code, with sage-3.2.3-pentiumM-ubuntu32bit-i686-Linux

For R we split the Python side in Lua in a pure Python script `test-R.py`:

```

import rpy2.robjects as robjects
import rpy2.rinterface as rinterface
class density(object):
    def __init__(self,samples,outpdf,w,h,kernel):
        self.samples = samples
        self.outpdf= outpdf
        self.kernel = kernel
        self.width=w
        self.height=h
    def run(self):
        r = robjects.r
        data = [int(k.strip())
                for k in
                file(self.samples,'r').readlines()
                ]
        x = robjects.IntVector(data)
        r.pdf(file=self.outpdf,
             width=self.width,
             height=self.height)
        z = r.density(x,kernel=self.kernel)
        r.plot(z[0],z[1],xlab='',ylab='')
        r['dev.off']()
if __name__ == '__main__' :
    dens =
    density('u-random-int', 'test-001.pdf',10,7,'o')
    dens.run()

```

and import this into Lua:

```
\startluacode
function testR(samples,outpdf,w,h,kernel)
  require("python")
  pyR = python.import("test-R")
  dens =
  pyR.density(samples,outpdf,w,h,kernel)
  dens.run()
end
\stopluacode

\def\plotdensiy[#1]{%
\getparameters[R][#1]%
\expanded{\ctxlua{testR("\Rsamples",
                        "\Routpdf",
                        \Rwidth,
                        \Rheight,"\Rkernel")}}}}

\setupbodyfont[sans,14pt]
\starttext
\startTEXpage
\plotdensiy[samples={u-random-int},
            outpdf={test-001.pdf},
            width={10},height={7},
            kernel={o}]

\setupcombinations[location=top]
\startcombination[1*2]
{\vbox{\hsize=400bp
This is a density plot of around {\tt 100 000}
random numbers between
$0$ and $2^{16}-1$ generated
from {\tt \hbox{/dev/urandom}}}}{
{\externalfigure[test-001.pdf][width={400bp}]}{
\stopcombination
\stopTEXpage
\stoptext
```

Note the conditional statement
`if __name__ == '__main__':`
 that permits to test the script with an ordinary Python
 interpreter.



It's worth noting that rpy2 is included in Sage too.

For more information about scientific computation with Python see [61] and [62] (also with site [31]) and [54].

The example of Sage shows that in this case we can think of `luatex lunatic` as an *extension* of Sage but also that `luatex lunatic` is *extended* with Sage. This is somehow similar to CWEB: code and description are tangled together, but now there's not a specific language like C in CWEB (in fact we can do the same with R). Eventually "untangled" is a matter of separation of

This is a density plot of around 100 000 random numbers between 0 and $2^{16} - 1$ generated from `/dev/urandom`

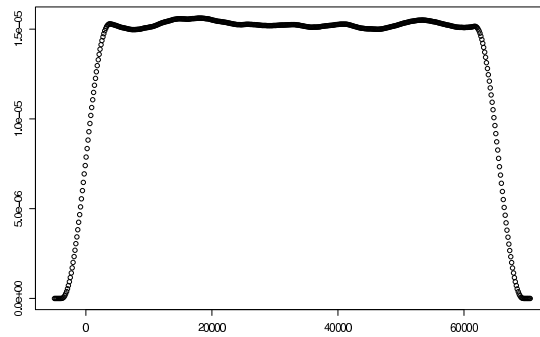


Figure 2. Result of the R code

Python code in a different file from the source tex file.



By the way, it's important to underline that CWEB is more advanced than other (C) code documentation systems because it embeds source code inside descriptive text rather than the reverse (as is common practice in most programming languages). Documentation can be not "linear", a bit unusual for ordinary programmers, but it's a more efficient and effective description of complex systems. Here we are talking about "linear" documentation, much like this article: a linear sequence of text-and-code printed as they appear.

Of course some computations may require much more time to be completed than the time of generation of the respective document (and ConTeXt is also a multipass system), so this approach is pretty inefficient—we need a set of macros that take care of intermediate results, i.e. *caching macros* or *multipass macros*. For example, in ConTeXt-mkiv we can say

```
\doifmode{*first}{%
  % this code will be executed only at first pass
  \Mymacro
}
```

so `\Mymacro` will be executed only at the first pass; there is also a Two Pass Data module `core-two.mkiv` that can be used for this, but don't forget that we also have Lua and Python for our needs.

Graphs

In `LuaTeX-ConTeXt-mkiv` we already have a very powerful tool for technical drawing: MetaPost. Simple searching reveals for example METAGRAPH [32] or the more generic LuaGRAPH [19], a Lua binding to graphviz [38] with output also in MetaPost; both are capable of drawing (un)directed graphs and/or networks. The next two modules are more oriented to graph calculations.

☕ MetaPost is an example of “embedding” an interpreter in LuaTeX at compilation time (see `luaopen_mplib(L)` in `void luainterpreter(void)` in `$HOMEDIR/source/teX/web2c/luatexdir/luastuff.c`). So hosting Python is not a new idea: the difference is that the “embedding” is done at run time.

igraph. *igraph* “is a free software package for creating and manipulating undirected and directed graphs. It includes implementations for classic graph theory problems like minimum spanning trees and network flow, and also implements algorithms for some recent network analysis methods, like community structure search. The efficient implementation of *igraph* allows it to handle graphs with millions of vertices and edges. The rule of thumb is that if your graph fits into the physical memory then *igraph* can handle it. [11]

☕ To install *igraph* we must first install *pycairo*, a Python binding to *cairo* [1], a well known 2D graphics library: so we gain another tool for generic drawing.

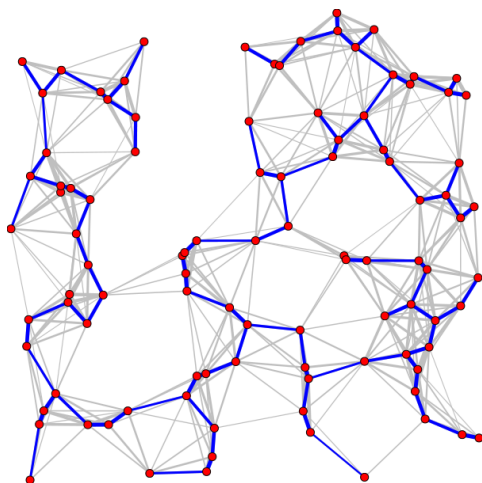


Figure 3. The result of the *igraph* code.

This time we coded the Python layer as a class:

```
import igraph
class spanningtree(object) :
    def __init__(self, ofn):
        self.ofn = ofn
    def distance(self, p1, p2):
        return ((p1[0]-p2[0]) ** 2
                + (p1[1]-p2[1]) ** 2) ** 0.5
    def plotimage(self):
        res = igraph.Graph.GRG(100,
                               0.2, return_coordinates=True)
        g = res[0]
        xs = res[1]
        ys = res[2]
        layout = igraph.Layout(zip(xs, ys))
```

```
weights = [self.distance(layout[edge.source],
                          layout[edge.target]) for edge in g.es]
max_weight = max(weights)
g.es["width"] = \
[6 - 5*weight/max_weight for weight in weights]
mst = g.spanning_tree(weights)

fig = igraph.Plot(target=self.ofn)
fig.add(g, layout=layout,
        opacity=0.25,
        vertex_label=None)
fig.add(mst,
        layout=layout,
        edge_color="blue",
        vertex_label=None)
fig.save()
if __name__ == '__main__':
    sp = spanningtree('test-igraph.png')
    sp.plotimage()
```

In this case we calculate a minimum spanning tree of a graph, and save the result in `test-igraph.png`. The Lua layer is so simple that it is encapsulated in a TeX macro:

```
\def\PlotSpanTree#1{%
\startluacode
require("python")
local spantree_module
local sp
spantree_module =
python.import("test-igraph")
sp = spantree_module.spanningtree("#1")
sp.plotimage()
\stopluacode
\externalfigure[#1]
\starttext
\startTEXpage
\PlotSpanTree{test-igraph.png}
\stopTEXpage
\stoptext
```

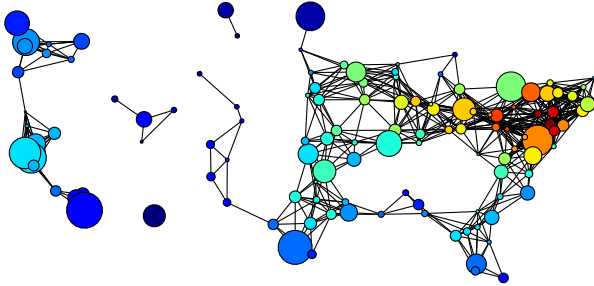
NetworkX. *NetworkX* is a Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks. [22]

The code is simpler: we have only two layers: the Python layer, and the TeX layer. The Python layer is a trivial modification of `knuth_miles.py` (see [24], [36], [60]), and is left to the reader (hint: rename `..__init__..` in `def run()`).

```
\starttext
\startTEXpage
\ctxlua{require("python");
knuth=python.import("test-networkx");
knuth.run();}
```

```
\externalfigure[knuth_miles]
\stopTEXpage
\stoptext
```

Here is the result (with a bit of imagination, one can see the USA):



ROOT. ROOT is an object-oriented program and library developed by CERN. It was originally designed for particle physics data analysis and contains several features specific to this field. [7], [26]

In this example we will draw 110 lines from file data (each line being 24 float values separated by spaces); each line will be a curve to fit with a polynomial of degree 6. We isolate all relevant parts in a Python script test-ROOT1.py:

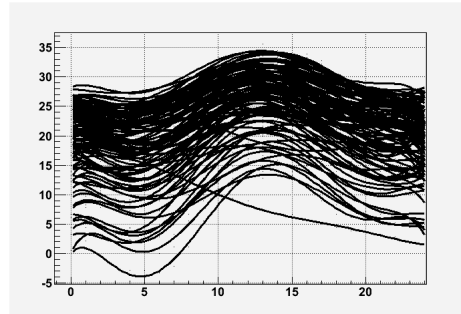
```
from ROOT import TCanvas,
    TGraph,TGraphErrors,TMultiGraph
from ROOT import gROOT
from math import sin
from array import array
def run(filename):
    c1 = TCanvas("c1","multigraph",200,10,700,500)
    c1.SetGrid()
    mg = TMultiGraph()
    n = 24; x = array('d',range(24))
    data = file('data').readlines()
    for line in data:
        line = line.strip()
        y = array('d',
            [float(d) for d in line.split()])
        gr = TGraph(n,x,y)
        gr.Fit("pol6","q")
        mg.Add(gr)
    mg.Draw("ap")
    c1.Update(); c1.Print(filename)
```

This is the Con \TeX t side:

```
\startluacode
function test_ROOT(filename)
    require("python")
    test = python.import('test-ROOT1')
    test.run(filename)
```

```
end
\stopluacode
\starttext \startTEXpage
\ctxlua{test_ROOT("data.pdf")}
\rotate[rotation=90]{\externalfigure[data.pdf]}
\stopTEXpage \stoptext
```

Here is the result:



Database

Oracle Berkeley DB XML. Oracle Berkeley DB XML “is an open source, embeddable xml database with XQuery-based access to documents stored in containers and indexed based on their content. Oracle Berkeley DB XML is built on top of Oracle Berkeley DB and inherits its rich features and attributes” [45];

We take as our data source a Wikiversity XML dump [8], more specifically enwikiversity-20090627-pages-articles.xml, a ~95MByte uncompressed xml file (in some sense, we end were we started).

Building a database is not trivial, so one can see [35] under Build_the_container for details. The most important things are indexes; here we use

```
container.addIndex("", "title",
    "edge-element-substring-string", uc)
container.addIndex("", "username",
    "edge-element-substring-string", uc)
container.addIndex("", "text",
    "edge-element-substring-string", uc)
```

These indexes will be used for substring queries, but not for regular expressions, for which it will be used the much slower standard way. Again it’s better to isolate the Python code in a specific module, wikidbxml_queryTxn.py (see [35] under Make pdf for details). This module does the most important work: translate from a ‘MediaWiki-format’ to Con \TeX t-mkiv. A ‘MediaWiki-format’ is basically made by <page> like this:

```
<page>
<title>Wikiversity:What is Wikiversity?</title>
```



```
<id>6</id>
<revision>
<id>445629</id>
<timestamp>2009-06-08T06:30:15Z</timestamp>
<contributor>
<username>Jr.duboc</username>
<id>138341</id>
</contributor>
<comment>/* Wikiversity for teaching */</comment>
<text xml:space="preserve">{{policy|[[WV:IS]]}}
{{about wikiversity}}
[[Image:Plato i sin akademi,
av Carl Johan Wahlbom
(ur Svenska Familj-Journalen).png
|thumb|left|300px|Collaboration between students
and teachers.]]
__TOC__
==Wikiversity is a learning community==
[[Wikiversity]] is a community effort to learn
and facilitate others'
learning. You can use Wikiversity to find
information or ask questions about a subject you
need to find out more about. You can also use it
to share your knowledge about a subject,
and to build learning
materials around that knowledge.
:
&lt;!-- That's all, folks! --&gt;
</text>
</revision>
</page>
```

So, a `<page>` is an xml document with non-xml markup in `<text>` node (which is an unfortunate tag name for an xml document); even if `<page>` is simple, parsing `<text>` content, or, more exactly, the text node of `<text>` node, is not trivial, and we can:

- implement a custom parser using the lpeg module of ConTeXt-mkiv (e.g. \$HOMEDIR/minimals/tex/texmf-context/tex/context/base/lxml-tab.lua); this can be a good choice, because we can translate 'MediaWiki-format' directly into ConTeXt markup, but of course we must start from scratch;
- use an external tool, like the Python module mwlib: MediaWiki parser and utility library [25].

We choose mwlib (here in vers. 0.11.2) and implement the translation in two steps:

1. from 'MediaWiki-format' to XML-DocBook (more exactly DocBook RELAX NG grammar 4.4; see [44])
2. from XML-DocBook to ConTeXt-mkiv (this is done by the `getConTeXt(title, res)` function)



Actually, the `wikidbxml_queryTxn.writers()` function writes the result of the query by calling `wikidbxml_queryTxn.getArticleByTitle()` which in turn calls `wikidbxml_queryTxn.getConTeXt()` function.

The ConTeXt-mkiv side is (for the moment forget about the functions `listtitles(title)` and `simplereports(title)`):

```
\usetyescriptfile[type-gentium]
\usetyescript[gentium]
\setupbodyfont[gentium,10pt]
\setuppapersize[A5][A5]
\setuplayout[height=middle,
topspace=1cm,header={2\lineheight},
footer=0pt,backspace=1cm,margin=1cm,
width=middle]
%%
%% DB XML
%%
\startluacode
function testdbxml(title,preamble,
                    postamble,filename)
    require("python")
    pg = python.globals()
    wikiversity =
        python.import("wikidbxml_queryTxn")
    wikiversity.writers(title,preamble,
                        postamble,filename)
end
\stopluacode
%%
%% sqlite
%%
\startluacode
function listtitles(title)
    require("python")
    pg = python.globals()
    wikiversity =
        python.import("wikidbxml_queryTxn")
    r = wikiversity.querycategory(title)
    local j = 0
    local res = r[j] or {}
    while res do
        local d =
            string.format("%s\par",
                string.gsub(tostring(res),'_',' '))
        tex.sprint(tex.ctxcatcodes,d)
        j = j+1
        res = r[j]
    end
end
\stopluacode
%%
%% sqlite
```



```

%%
\startluacode
function simplereports(title)
  require("python")
  pg = python.globals()
  wikiversity =
    python.import("wikidbxml_queryTxn")
  r = wikiversity.simplereports(title)
  local j = tonumber(r)
  for v = 0, j-1 do
    local d =
      string.format("\\input reps\\%04d ", v)
    tex.sprint(tex.ctxcatcodes, d)
  end
  print( j )
end
\stopluacode
%% ConTeXt
\def\testdbxml[#1]{%
\getparameters[dbxml][#1]%
\ctxlua{%
testdbxml("\csname dbxmltitle\endcsname",
"\csname dbxmlpreamble\endcsname",
"\csname dbxmlpostamble\endcsname",
"\csname dbxmlfilename\endcsname")}%
\input \csname dbxmlfilename\endcsname %
}
\starttext
\testdbxml[title={Primary mathematics/Numbers},
preamble={},
postamble={},
filename={testres.tex}]
\stoptext

```

Here we query for the exact title Primary mathematics/Numbers: for the result, see page 55.

sqlite. Python offers adapters for practically all well known databases like PostgreSQL, MySQL, ZODB, etc. (“ZODB is a persistence system for Python objects” written in Python, see [16]. ZODB is the heart of Plone [47], a popular content management system also written in Python), but here we conclude with *sqlite*, a “software library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine. *SQLite is the most widely deployed SQL database engine in the world. The source code for SQLite is in the public domain*” (see [29]).

sqlite is a module of the standard Python library, and we can use it to query a `category.db` for titles. (`category.db` is a db made from `enwikiversity-20090627-category.sql`, which is a MySQL dump. Conversion is not difficult and is not shown here.)

The code uses the two functions seen before, `listtitles` and `simplereports`:

```

\starttext
{\bfb Query for 'geometr':}
\ctxlua{listtitles("geometr")}%
\ctxlua{simplereports("geometr")}%
\stoptext

```

See p. 57 for a short result (actually the first page of second hit, Geometry. The complete document is 12 pages).

MetaTeX

What is MetaTeX?

Quoting from `$HOMEDIR/tex/texmf-context/tex/context/base/metatex.tex`:

This format is just a minimal layer on top of the LuaTeX engine and will not provide high level functionality. It can be used as basis for dedicated (specialized) macro packages.

A format is generated with the command:
`luatools --make --compile metatex`

It should be clear from previous examples that a system with *all* these “bindings” becomes quickly unmanageable: one can spend almost all available time upgrading to latest releases. Just as an example: already at time of preprinting `ghostscript` was at rel. 8.70 (vs. rel. 8.64 of this article) Sage was at rel. 4.1 (vs. rel. 3.2.2), Python itself was at rel. 2.6.2 (vs. rel. 2.6.1) and there even exists rel. 3.1

Also not all Python packages are “robust”: for example, in the file `docbookwriter.py` of `mwlib` we can see

Generate DocBook from the DOM tree generated by the parser.

Currently this is just a proof of concept which is very incomplete

(and of course, `mwlib` was at rel. 0.12.2 (vs. rel. 0.11.2) so this message may have disappeared as well).

So, in the end, it’s better to have more distinct “tools” than one big tool that does anything and badly. We can see now why MetaTeX fits well in this line: it permits to create the exact “tool” needed and `luatex lunatic` can be used to complete this tool. For example, consider the problem of typesetting labels like the one on top if the next page.

Basically, it’s a table with barcode and two or three fonts (preferably monospaced fonts), most of the time black and white. `ConTeXt-mkiv` already comes with natural tables, or even a layer mechanism (see [70]); `luatex-lunatic` with `barcode.ps` provides the barcode. We don’t need colors, interaction, indexes, sectioning.



Financial reports are similar: here we can benefit from the decimal Python module that is included in the standard library (decimal is an implementation of Decimal fixed point and floating point arithmetic; see [28]).

MetaTeX can be used to produce very specific formats for educational purposes: think for example of a MetaTeX Sage, or a MetaTeX R from the CWEB point of view, i.e. embedded source code inside descriptive text rather than the reverse.

Also, Python can be used as a query language for Plone (mentioned previously), a powerful CMS written in Python, so it can be possible to print specific content type without translating it into an intermediate form like xml (and maybe in the future the reverse too, i.e. push a content type made by a MetaTeX Plone).

Conclusion

LuaTeX with ConTeXt-mkiv is a powerful tool for publishing content, and with an embedded Python interpreter we unquestionably gain more power, especially when MetaTeX becomes stabilized. If one wants, one can also experiment with JPype “an effort to allow Python programs full access to Java class libraries. This is achieved not through re-implementing Python, as Jython/JPYthon has done, but rather through interfacing at the native level in both virtual machines” [12] (currently unstable under Linux).

So it’s better here to emphasize “the dark side of the moon”.

First, it should be clear that currently we cannot assure **stability** and **portability** in the TeX sense.

Moreover, under Linux there is immediately a price to pay: symbol collisions. Even if the solution presented here should ensure that there are no symbol collisions between luatex and an external library, it doesn’t resolve problems of collision between symbols of two external libraries; installing all packages under a folder /opt/luatex/luatex-lunatic can help to track this problem, but it’s not a definitive solution. Of course, we avoid this problem if we use pure Python libraries, but these tend to be slower than C libraries.

ctypes looks fascinating, but a binding in ctypes is usually not easy to build; we must not forget that Lua offers its loadlib that can always be used as an alternative to ctypes or to any other Python alternative like SWIG [55] which can, anyway, build wrapper code for Lua too, at least from development release 1.3. In the end, an existing Python binding is a good choice if it is stable, rich, complete and mature with respect to an existing Lua binding, or if there is not a Lua binding.

For a small script, coding in Lua is not much different from coding in Python; but if we have complex objects, things can be more complicated: for example this Python code

```
z = x*np.exp(-x**2-y**2)
```

is translated in this not-so-beautiful Lua code

```
z=x.__mul__(np.exp((x.__pow__(2).
    __add__(y.__pow__(2))).__neg__()))
```

(see [35]#Scipy). It is better to separate the Python layer into an external file, so we can eventually end in a *py,*lua,*tex for the same job, adding complexity to manage.

In the end, note that a Python interpreter does not “complete” in any sense luatex, because Lua is a perfect choice: it’s small, stable, and OS-aware. Conversely, Python is bigger, and today we are seeing Python versions 2.4, 2.5, 2.6.2, 2.7 alpha, 3.1 ... not exactly a stable language from a TeX user point of view.

Acknowledgements

The author would like to thank Taco Hoekwater and Hans Hagen for their suggestions, help and encouragement during the development and the writing process of this article.

The author is also immensely grateful to Massimiliano “Max” Dominici for his strong support, help and encouragement.

References

All links were verified between 2009.08.17 and 2009.08.21.

- [1] <http://cairographics.org>
- [2] <http://cg.scs.carleton.ca/~luc/PSgeneral.html>
- [3] <https://code.launchpad.net/~niemeyer/lunatic-python/trunk>
- [4] <http://download.wikimedia.org>
- [5] http://en.wikipedia.org/wiki/Call_graphs
- [6] [http://en.wikipedia.org/wiki/Python_\(programming_language\)](http://en.wikipedia.org/wiki/Python_(programming_language))
- [7] <http://en.wikipedia.org/wiki/ROOT>

- [8] http://en.wikiversity.org/wiki/Getting_stats_out_of_Wikiversity_XML_dumps
- [9] <http://gcc.gnu.org/wiki/Visibility>
- [10] <http://ghostscript.com/>
- [11] <http://igraph.sourceforge.net>
- [12] <http://jpyype.sourceforge.net/>
- [13] <http://kcachegrind.sourceforge.net/cgi-bin/show.cgi>
- [14] <http://labix.org/lunatic-python>
- [15] <http://labix.org/python-bz2>
- [16] <https://launchpad.net/zodb>
- [17] <http://linux.die.net/man/1/ld>
- [18] <http://linux.die.net/man/3/dlopen>
- [19] <http://luagraph.luaforge.net/graph.html>
- [20] <http://luatex.bluwiki.com/go/User:Luigi.scarso>
- [21] <http://meeting.contextgarden.net/2008>
- [22] <http://networkx.lanl.gov>
- [23] <http://minimals.contextgarden.net/>
- [24] http://networkx.lanl.gov/examples/drawing/knuth_miles.html
- [25] <http://pypi.python.org/pypi/mwlib>
- [26] <http://root.cern.ch>
- [27] <http://rpy.sourceforge.net/>
- [28] <http://speleotrove.com/decimal>
- [29] <http://sqlite.org/>
- [30] <http://valgrind.org/>
- [31] <http://vefur.simula.no/intro-programming/>
- [32] <http://vigna.dsi.unimi.it/metagraph>
- [33] http://wiki.contextgarden.net/Future_ConTeXt_Users
- [34] <http://wiki.contextgarden.net/Image:Trick.zip>
- [35] http://wiki.contextgarden.net/User:Luigi.scarso/luatex_lunatic
- [36] <http://www-cs-faculty.stanford.edu/~knuth/sgb.html>
- [37] <http://www.codeplex.com/IronPython>
- [38] <http://www.graphviz.org>
- [39] <http://www.gnu.org/software/libtool>
- [40] <http://www.imagemagick.org/script/index.php>
- [41] <http://www.jython.org>
- [42] <http://www.math.ubc.ca/~cass/graphics/text/www/index.html>
- [43] <http://www.luatex.org>
- [44] <http://www.oasis-open.org/docbook>
- [45] <http://www.oracle.com/database/berkeley-db/xml/index.html>
- [46] <http://www.pathname.com/fhs/>
- [47] <http://www.plone.org>
- [48] <http://www.procoders.net/?p=39>
- [49] <http://www.python.org>
- [50] <http://www.python.org/doc/2.6.1/library/ctypes.html>
- [51] <http://www.pythonware.com/products/pil/>
- [52] <http://www.r-project.org/>
- [53] <http://www.sagemath.org/>
- [54] <http://www.scipy.org/>
- [55] <http://www.swig.org>
- [56] <http://www.terryburton.co.uk/barcodewriter/>
- [57] private email with Taco Hoekwater
- [58] Bill Casselman, *Mathematical Illustrations: A Manual of Geometry and PostScript*. ISBN-10: 0521547881, ISBN-13: 9780521547888 Available at site <http://www.math.ubc.ca/~cass/graphics/text/www/index.html>
- [59] Danny Brian, *The Definitive Guide to Berkeley DB XML*. Apress, 2006. ISBN-13: 978-1-59059-666-1
- [60] Donald E. Knuth, *The Stanford GraphBase: A Platform for Combinatorial Computing*. ACM Press, New York, 1993. ISBN 978-0-470-75805-2
- [61] Hans Petter Langtangen, *A Primer on Scientific Programming with Python*. Springer, 2009. ISBN: 978-3-642-02474-0
- [62] Hans Petter Langtangen, *Python Scripting for Computational Science*. Springer, 2009. ISBN: 978-3-540-73915-9
- [63] John Levine, *Linkers & Loaders*. Morgan Kaufmann Publisher, 2000. ISBN-13: 978-1-55860-496-4
- [64] Mark Lutz, *Learning Python, Fourth Edition*. O'Reilly, September 2009 (est.) ISBN-10: 0-596-15806-8, ISBN 13: 978-0-596-15806-4
- [65] Mark Lutz, *Programming Python, Third Edition*. O'Reilly, August 2006. ISBN-10: 0-596-00925-9, ISBN 13: 978-596-00925-0
- [66] [luatexref-t.pdf](#). Available in manual folder of `luatex-snapshot-0.42.0.tar.bz2`
- [67] Priscilla Walmsley, *XQuery*. O'Reilly, April 2007. ISBN-13: 978-0-596-00634-1
- [68] Roberto Ierusalimschy, *Programming in Lua (second edition)*. Lua.org, March 2006. ISBN 85-903798-2-5
- [69] Ulrich Drepper, *How to Write Shared Libraries*. <http://people.redhat.com/drepper/dsohowto.pdf>
- [70] Willi Egger, ConTeXt: Positioning design elements at specific places on a page (tutorial). EuroT_EX 2009 & 3rd ConT_EXt Meeting
- [71] Yosef Cohen and Jeremiah Cohen, *Statistic and Data with R*. Wiley 2008. ISBN 978-0-470-75805-2

I currently use Ubuntu Linux, on a standalone laptop—it has no Internet connection. I occasionally carry flash memory drives between this machine and the Macs that I use for network surfing and graphics; but I trust my family jewels only to Linux.

— Donald Knuth

Interview with Donald Knuth

By Donald E. Knuth and Andrew Binstock

Apr. 25, 2008

<http://www.informit.com/articles/article.aspx?p=1193856>

The lunatic is on the grass

The lunatic is on the grass

Remembering games and daisy chains and laughs

Got to keep the loonies on the path

— Brain Damage,

The Dark Side of the Moon,

Pink Floyd 1970

Mr. LuaT_EX hosts a Python,

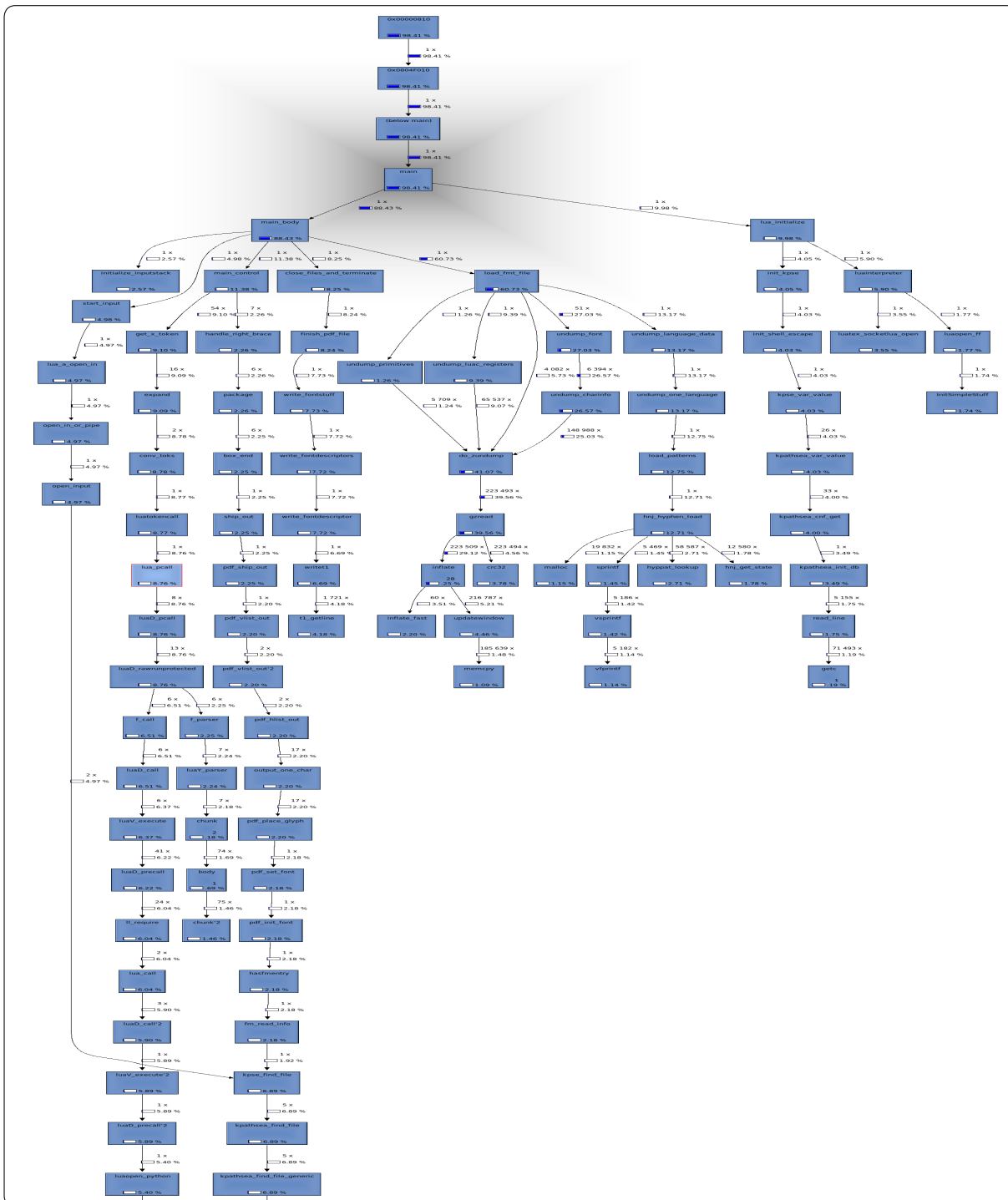
and become a bit lunatic

— Anonymous

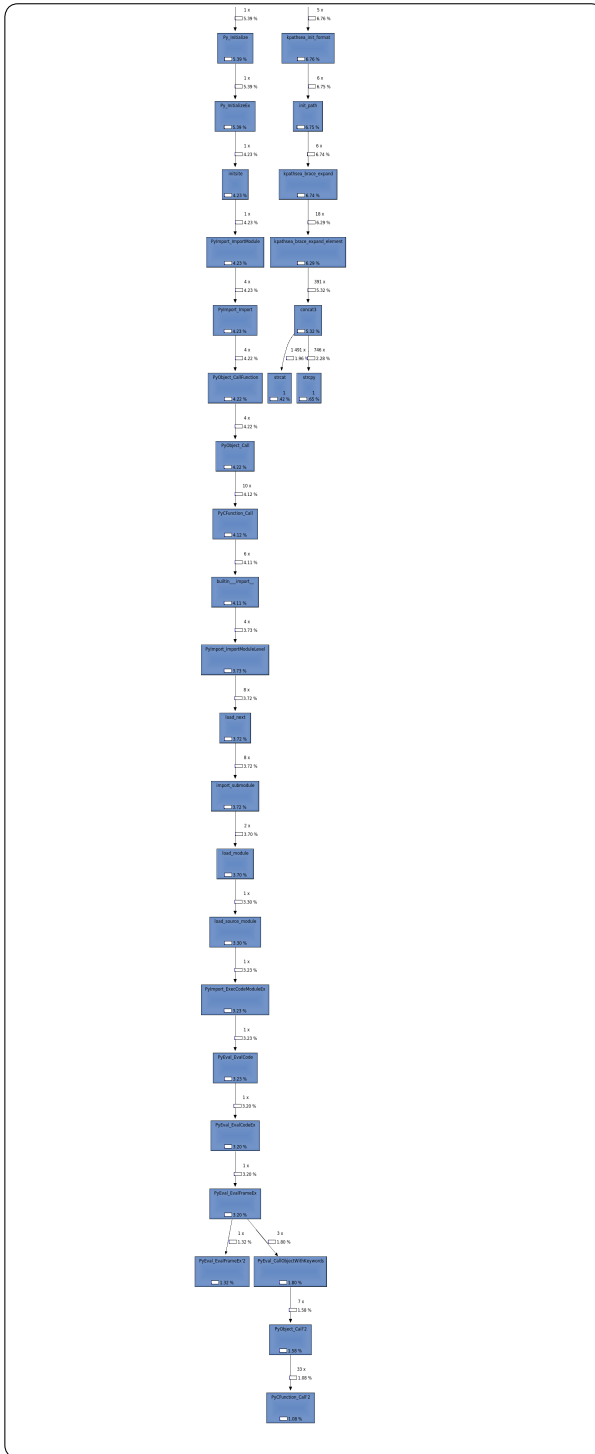
Luigi Scarso

Appendix

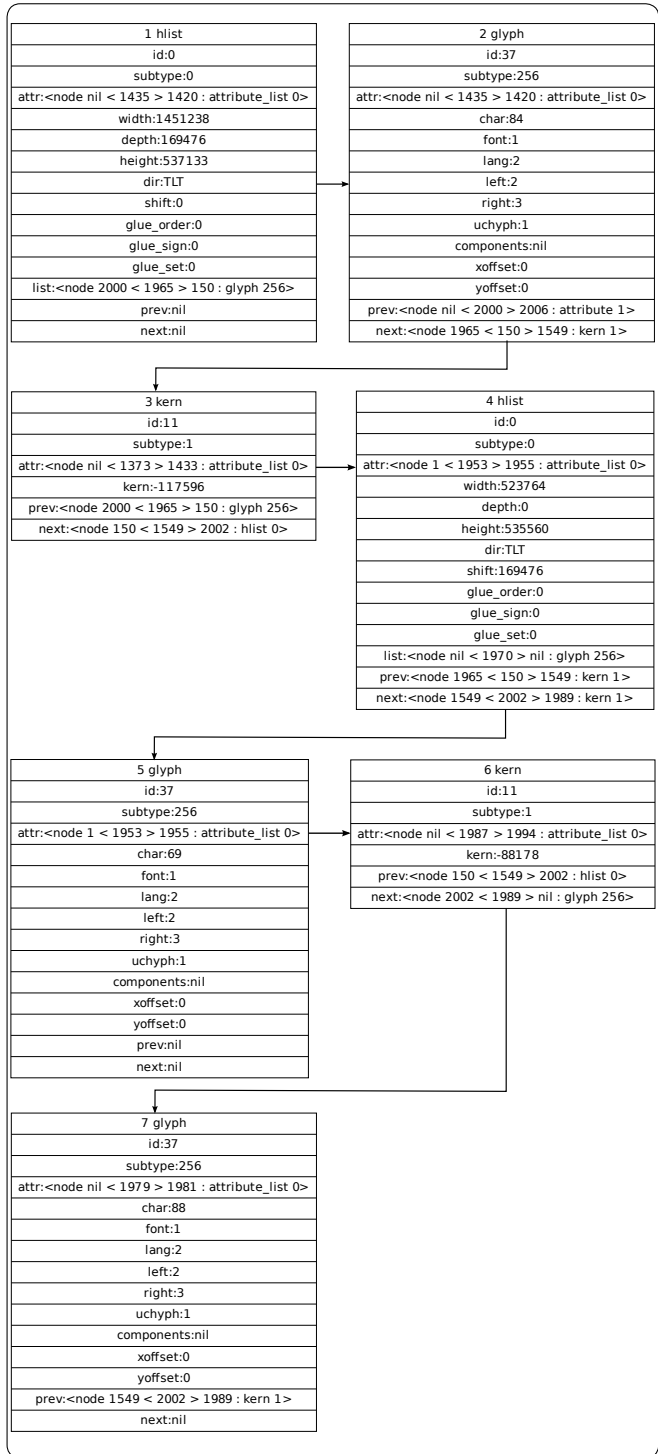
Call graph of a simple run



Call graph of a simple run, cont.



T_EX, forever



T_EX nodelist made with lunatic binding for graphviz

DB XML example

1 Primary mathematics/Numbers

1.1 Primary mathematics/Numbers

1.1.1 Teaching Number

This page is for teachers or home-schoolers. It is about teaching the basic concepts and conventions of simple number.

1.1.1.1 Developing a sound concept of number

Children typically learn about numbers at a very young age by learning the sequence of words, "one, two, three, four, five" etc. Usually, in chanting this in conjunction with pointing at a set of toys, or mounting a flight of steps for example. Typically, 'mistakes' are made. Toys or steps are missed or counted twice, or a mistake is made in the chanted sequence. Very often, from these sorts of activities, and from informal matching activities, a child's concept of number and counting emerges as their mistakes are corrected. However, here, at the very foundation of numerical concepts, children are often left to 'put it all together' themselves, and some start off on a shaky foundation. Number concepts can be deliberately developed by suitable activities. The first one of these is object matching.

1.1.2 Matching Activities

As opposed to the typical counting activity children are first exposed to, matching sets of objects gives a firm foundation for the concept of number and numerical relationships. It is very important that matching should be a physical activity that children can relate to and build on.

Typical activities would be a toy's tea-party. With a set of (say) four toy characters, each toy has a place to sit. Each toy has a cup, maybe a saucer, a plate etc. Without even mentioning 'four', we can talk with the child about 'the right number' of cups, of plates etc. We can talk about 'too many' or 'not enough'. Here, we are talking about number and important number relations without even mentioning which number we are talking about! Only after a lot of activities of this type should we talk about specific numbers and the idea of number in the abstract.

1.1.3 Number and Numerals

Teachers should print these numbers or show the children these numbers. Ideally, the numbers should be handled by the student. There are a number of ways to achieve this: cut out numerals from heavy cardstock, shape them with clay together, purchase wooden numerals or give them sandpaper numerals to trace. Simultaneously, show the definitions of these numbers as containers or discrete quantities (using boxes and small balls, eg. 1 ball, 2 balls, etc. Note that 0 means "no balls"). This should take some time to learn thoroughly (depending on the student).

0 1 2 3 4 5 6 7 8 9

1.1.4 Place Value

The Next step is to learn the place value of numbers.

It is probably true that if you are reading this page you know that after 9 comes 10 (and you usually call it ten) but this would not be true if you would belong to another culture.

Take for example the Maya Culture where there are not the ten symbols above but twenty symbols.

cfr <http://www.michielb.nl/maya/math.html>

Imagine that instead of using 10 symbols one uses only 2 symbols. For example 0 and 1

Here is how the system will be created:

Binary	0	1	10	11	100	101	110	111	1000	...
Decimal	0	1	2	3	4	5	6	7	8	...

Or if one uses the symbols A and B one gets:

Binary	A	B	BA	BB	BAA	BAB	BBA	BBB	BAAA	...
Decimal	0	1	2	3	4	5	6	7	8	...

This may give you enough information to figure the place value idea of any number system.

For example what if you used 3 symbols instead of 2 (say 0,1,2).

Trinary	0	1	2	10	11	12	20	21	22	100	...
Decimal	0	1	2	3	4	5	6	7	8	9	...

If you're into computers, the HEXADECIMAL (Base 16) or Hex for short, number system will be of interest to you. This system uses 4 binary digits at a time to represent numbers from 0 to 15 (decimal). This allows for a more convenient way to express numbers the way computers think - that we can understand. So now we need 16 symbols instead of 2, 3, or 10. So we use 0123456789ABCDEF.

Binary	0	1	10	11	100	101	110	111	1000	1001	1010	1011	1100	1101	1110	1111	10000	...
Decimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	...
Hex	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	...

1.1.5 Resources for Early Math

15 Fun Ways and Easy Games for Young Learners Math: Reproducible, Easy-to-Play Learning Games That Help Kids Build Essential Math Skills, by Susan Julio, Scholastic Professional, 2001.

Enie Meenie Miney Math!: Math Play for You and Your Preschooler, by Linda Allison, Little Brown & Co., 1993.

Marshmallow Math: Early Math for Toddlers, Preschoolers and Primary School Children, by Trevor Schindeler, Trafford, 2002.

Number Wonder: Teaching Basic Math Concepts to Preschoolers, by Deborah Saathoff and Jane Jarrell, Holman Bible, 1999.

cfr Category:School of Mathematics

cfr Category:Pages moved from Wikibooks

cfr Category:Primary education

Next in Primary School Mathematics:

cfr http://en.wikiversity.org/wiki/Primary_mathematics:Adding_numbers

sqlite example

1

Query for 'geometr': Geometric algebra

Geometry

Introductory Algebra and Geometry

Orbital geometry

Coordinate Geometry

2 Geometry

2.1 Geometry

This subdivision is dedicated to bridging the gap between the mathematical layperson and the student who is ready to learn calculus and higher mathematics or to take on any other endeavour that requires an understanding of basic algebra and (at least Euclidean) geometry.

2.1.1 Subdivision news

2.1.2 Departments

2.1.3 Active participants

The histories of Wikiversity pages indicate who the active participants are. If you are an active participant in this subdivision, you can list your name here (this can help small subdivisions grow and the participants communicate better; for large subdivisions a list of active participants is not needed). Please remember: if you have an

cfr <http://en.wikiversity.org/w/index.php?title=Special:Userlogin&type=signup>

cfr Category:Geometry

cfr Category:Introductions

cfr \#

cfr \#

In Cartesian or Analytic Geometry we will learn how to represent points, lines, and planes using the Cartesian Coordinate System, also called Rectangular Coordinate System. This can be applied to solve a broad range of problems from geometry to algebra and it will be very useful later on Calculus.

2.1.4 Cartesian Coordinate System

The foundations of Analytic Geometry lie in the search for describing geometric shapes by using algebraic equations. One of the most important mathematicians that helped to accomplish this task was René Descartes for whom the name is given to this exciting subject of mathematics.

2.1.4.1 The Coordinate System

For a coordinate system to be useful we want to give to each point an attribute that help to distinguish and relate different points. In the Cartesian system we do that by describing a point using the intersection of two(2D Coordinates) or more(Higher Dimensional Coordinates) lines. Therefore a point is represented as $P(x_1, x_2, x_3, \dots, x_n)$ in "n" dimensions.

2.1.5 Licensing:

"Geometry is the only science that it hath pleased God hitherto to bestow on mankind."—Thomas Hobbes

This department of the Olympiad Mathematics course focuses on problem-solving based on circles and vectors, thus generalizing to Coordinate Geometry. Our major focus is on Rectangular (Cartesian) Coordinates, although the course does touch upon Polar coordinates.

The first section is based on the geometric study of circles. Although not based on pure analytical geometry, it uses Apollonius-style reference lines in addition to Theorems on Tangents, Areas, etc.

The second section is devoted to Vector Analysis, covering problem-solving from Lattices and Affine Geometry to Linear Algebra of Vectors

Third section, focusing on locus problems, is all about conic sections and other curves in the Cartesian plane.

2.1.6 Textbooks

2.1.7 Practice Questions