Una estensione di luatex: luatex lunatic

Luigi Scarso

Sommario

`luatex lunatic` è una estensione dell'interprete `Lua` di `luatex`, per consentire l'integrazione di un completo interprete `python`.

Intenzionalmente l'integrazione dell'interprete è limitata alla versione 2.6 di Python e a una versione di `luatex` per sistemi operativi Linux a 32 bit.

Abstract

`luatex lunatic` is an extension of the `Lua` interpreter of `luatex` to permit an embedding of a complete `python` interpreter.

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

1 Introduzione

La parola "TEX" assume ai giorni nostri tre diversi significati distinguibili dal contesto:

1. `TEX`: sistema batch di composizione tipografica. Il sistema è disponibile per diverse architetture hardware e software nella ormai consolidata distribuzione `Tex Collection` liberamente disponibile al `http://www.tug.org/texcollection`;

2. TEX: un linguaggio di programmazione. È un *macro* linguaggio Turing-complete, la cui semantica riguarda la composizione tipografica;

3. `*tex`: un particolare interprete del linguaggio TEX; ve ne sono più d'uno. In generale, l'output di un programma in TEX è un formato binario `dvi` (`DeVice Independent`) oppure `pdf` (`Portable Document Format`). È da notare che lo standard ISO/IEC 32000-1:2008 riguarda proprio la definizione di un formato pdf informalmente noto come pdf vers. 1.7.

Data la sua natura, è ovviamente possibile scrivere ed usare pacchetti di macro in TEX; per agevolarne il caricamento a tempo di esecuzione da parte dell'interprete, grossi pacchetti vengono usualmente precompilati in un *formato* binario, solitamente indicati dal suffisso `fmt`.

Al giorno d'oggi è molto raro che venga usato il linguaggio TEX senza alcun formato: le macro primitive sono troppo a basso livello. Così sono nati pacchetti di macro via via più complessi: `plain`, `eplain`, AMS-TEX, LATEX, ConTEXt… Naturalmente, un formato corrispondente ad un linguaggio complesso tendenzialmente è potente ma lento da caricare ed eseguire.

☞ Per brevità, con ConTEXt (ad esempio) si indica sia il linguaggio sia il formato.

Un secondo programma è praticamente sempre presente nelle installazioni: `mpost`, interprete di METAPOST, un linguaggio di tipo procedurale con sematica orientata al disegno vettoriale. È una modifica di METAFONT, un linguaggio procedurale con semantica orientata alla progettazione di font.

Esistono diversi pacchetti macro che permettono una stretta integrazione da TEX e METAPOST, basata nella maggior parte dei casi sul formato `dvi`.

Per quanto riguarda gli interpreti TEX è d'obbligo notare che il formato `dvi` viene praticamente usato solo internamente al sistema `TEX`, in quanto il `pdf` è lo standard de facto dell'industria della stampa (ed in misura minore il PostScript). Abbiamo, tra gli altri, i seguenti interpreti:

1. `tex`: output in `dvi`, necessita di un convertitore `dvips+ps2pdf`;

2. `etex` (*extended*TEX): output in `dvi`, necessita di un convertitore `dvips+ps2pdf`;

3. `pdftex`: output in `dvi` e `pdf`;

4. `xetex`: output in `xdv` (*extended*`dvi`), necessita di un convertitore `xdvipdfmx` ;

5. `luatex`: output in `dvi` e `pdf`.

A volte è possibile trovare il termine TEX *the program* per riferirsi all'interprete e TEX *the language* per riferirsi al linguaggio-macro.

Ogni interprete viene utilizzato con un pacchetto di macro (solitamente `plain`) e quindi con il relativo formato per garantire un minimo di funzionalità; per i formati LATEX e ConTEXt abbiamo i seguenti interpreti:

- LATEX: `pdftex`, `xetex`, più raramente `etex`
- ConTEXt: `pdftex`, `xetex`, `luatex`: `etex` è praticamente scomparso.

La differenza tra `plain`, `eplain`, LATEX e ConTEXt si può sintetizzare così:

- plain è sostanzialmente il formato per il TEXbook di Knuth: più precisamente plain costituisce l'ossatura, manmac contiene le macro per il layout che si "appoggiano" su plain. eplain è una estensione del formato plain, una prima generalizzazione di plain, non ai livelli di LATEX però.
- LATEX è un pacchetto di macro che nasce per fornire alcuni modelli di documenti (book, article, report, letter) ed è evoluto in modo non coordinato per aggregazione di altri *package*, anche se il nuovo LATEX3 tenta di rimediare a questo approccio
- ConTEXt nasce per affrontare *in generale* il problema della pubblicazione di contenuto, quindi fornisce macro generiche ad alto livello ed una ottima integrazione con METAPOST. Inoltre lo sviluppo è esclusiva di un ristrettissimo team (1-2 persone) ed i pacchetti di terze parti attualmente "vivi" (cioè utilizzati) sono non più di 5-6, in quanto i pacchetti base sono ampiamente sufficienti.

A scanso di equivoci, ConTEXt è un formato più lento di LATEX: per alcuni compiti (ad esempio typesetting di form) addirittura eplain potrebbe essere il formato più adatto, se non fosse in disuso.

Possiamo brevemente tracciare una evoluzione delle estensioni del linguaggio TEX:

1. TEX: nessun supporto per sorgente unicode, font type1, font truetype, opentype, direzionalità di composizione;
2. ε-TEX: nessun supporto per sorgente unicode, font type1, font truetype, opentype; si gioca sulla "e" per extended ma si trova anche ε, cioè "minima" nel gergo della matematica. ε-TEX è una importante estensione del TEX, ad esempio introduce la direzionalità di composizione (LR e RL);
3. pdfTEX: nessun supporto per sorgente unicode, font opentype; nuove primitive per il supporto del formato pdf. Il supporto parziale di Unicode è assicurato dalla codifica utf8x;
4. XƎTEX: supporto per unicode, font opentype;
5. LuaTEX: supporto per unicode, font opentype, scripting in Lua

Possiamo tracciare una catena estensiva in TEX $\subset$ ε-TEX $\subset$ pdfTEX $\subset$ LuaTEX; XƎTEX ha una genesi differente, anche se ora è a buon diritto un motore di impaginazione del TEX .

Ad esempio pdfTEX include ε-TEX che include TEX, ma già con LuaTEX l'inclusione con pdfTEX non è completa — forse in futuro. Allo stesso modo il programma pdftex può essere chiamato in modalità etex (output in dvi), quindi in definitiva esiste un solo programma pdftex che riunisce 3 diversi interpreti pdftex, etex e tex.

Ancora, TEX può essere programmato per il parsing degli *ottetti* dello stream di ingresso, quindi rende teoricamente possibile gestire qualsiasi tipo di codifica, inclusa UTF-8. Quindi il supporto per unicode in codifica UTF-8 di fatto è una realtà almeno a partire da pdfTEX.

Per riferimenti più puntuali è opportuno consultare l'ultima versione della GuidaGuIT (Beccari), (fare riferimento al sito del GuIT, perchè è aggiornata con una certa frequenza).

Gli interpreti tex, etex, pdftex, xetex sono codificati in un mix di pascalWEB, C, C++; luatex ha recentemente (EuroTEX 2009) completato la traduzione del codice da pascalWEB a C (e possibilmente C++) con il target CWEB (l'attuale linguaggio di programmazione di Knuth). Comunque, non si sbaglia di molto se si afferma che oggi la maggior parte degli utenti TEX usa pdfTEX+LATEX seguito XƎTEX+LATEX e pdfTEX+ConTEXt; ε-TEX e TEX hanno un uso via via sempre più ristretto.

LuaTEX merita una menzione particolare:

- nasce come fork dal codice di pdfTEX (a volte si parla di pdftex 2.0);
- *per design* utilizza lua come *glue language* tra le librerie C, permettendo di scrivere macro in lua nel sorgente *tex ed in generale facilitando l'accesso allo stato interno del programma grazie ad una nutrita serie di librerie. Per far ciò LuaTEX ha al suo interno un interprete lua customizzato;
- viene sviluppato in sincronia con ConTEXt: in questo modo si implementano le *feature* più importanti per un immediato utilizzo/test da parte degli utenti esperti;
- è staticamente linkato con mplib, fornendo una completa integrazione con METAPOST. Parallelamente allo sviluppo di LuaTEX il team ha infatti deciso di riprendere il programma mpost, tradurre il codice completamente in CWEB e trasformarlo in una libreria mplib (mpost utilizza ora la libreria mpilib) proprio per integrarla in LuaTEX. Il team ha calcolato che per alcuni job il guadagno in termini di tempo è di circa 1000, cosa ragionevole se si pensa ai costi di setup di un task runtime.

Con LuaTEX, il problema della presentazione del contenuto può essere "aggredito" su tre fronti: con un linguaggio macro come TEX, con un ordinario linguaggio di programmazione come Lua e con un linguaggio grafico ad alto livello come METAPOST, tutti e tre altamente integrati.

Allo stato attuale nessuno degli interpreti TEX visti offre queste caratteristiche.

Il manuale di riferimento per programmare in Lua è Ierusalimschy (2006).

2 LuaTeX

Come detto nel paragrafo precedente, LuaTeX è sviluppato in sincronia con ConTeXt. Tuttavia, ConTeXt può usare come motore di impaginazione pdfTeX e XeTeX. Come è gestita la scelta del motore?

L'evoluzione di ConTeXt passa attraverso 4 `mark`-fasi con una certa continuità, cfr. Hagen *et al.* (2009):

- `markI` : è il formato che usa `(e)tex` e `pdftex`, con comandi in olandese; praticamente scomparso, rimane qualche traccia in qualche pacchetto;
- `markII (mkii)`: è il formato attuale che usa `(e)tex` (uso non supportato), `pdftex` e `xetex` e che è *frozen*;
- `markIII`: è un termine riservato. Non corrisponde ad un formato e talvolta viene usato per indicare una possibile futura customizzazione di `xetex`, per separarlo da `markII`, oppure per indicare una evoluzione di `mkii` che usa esclusivamente `etex`. Un'altra ragione del non-uso è un possibile fraintendimento con LaTeX3, che è in intenso sviluppo;
- `markIV (mkiv)`: è il formato che usa esclusivamente `luatex` come motore. Condivide poco col formato `mkii`: anzi, il codice TeX per ConTeXt-`mkii` è indicato col suffisso `*mkii`, quello per ConTeXt-`mkiv` è indicato col suffisso `*mkiv` e `*lua`.

Infine a runtime:

- `texexec nomefile` seleziona ConTeXt-`mkii` con `pdftex`;
- `texexec -xtx nomefile` seleziona ConTeXt-`mkii` con `xetex`;
- `context nomefile` seleziona ConTeXt-`mkiv` con `luatex` .

Quindi ConTeXt-`mkiv` è in sostanza il modo migliore per usare LuaTeX, ed è questo che verrà usato in questo articolo; LuaTeX con `plain` è sicuramente vantaggioso in termini di tempo per testare singole parti.

L'altra caratteristica è la possibilità, almeno sulla carta, di sfruttare l'interprete `lua` per estendere `luatex` a runtime. Lua mette a disposizione la funzione `package.loadlib()` che permette il caricamento dinamico di librerie `*.so` o `*.dll`: una volta caricata, una libreria appare come una normale `table` di Lua, quindi utilizzabile in modo trasparente — posto che si conosca l'API (Application Program Interface) della relativa libreria.

Ad esempio, sarebbe possibile caricare runtime la libreria `libgs.so` o `libgs.dll` ed interagire con l'interprete postscript di Ghostscript — ancora una volta, bisogna conoscere l'API relativa, deducibile dalla documentazione e soprattutto dal codice `C` di qualche programma.

Il più delle volte è necessario del codice Lua di interfacciamento: questo codice costituisce il Lua *binding* della libreria.

Così, proseguendo su questa scia, risulta comprensibile il lavoro di Gustavo Niemeyer nel suo "laboratorio personale" (Niemeyer, 2009): **`Lunatic python`**.

Semplicemente, è un programma in `C` che permette di caricare la libreria `libpython.so` e di usare l'interprete Python tramite Lua, traducendo oggetti Python in oggetti Lua. È, per così dire, un "ponte" (*bridge*) tra Lua e Python; a volte si usa l'espressione "embedding" o "hosting".

☞ `Lunatic python` è in realtà qualcosa di più: permette a Python di usare l'interprete Lua, quindi è un *two-way bridge*

`luatex lunatic` è quindi il bridge *runtime* tra LuaTeX e Python.

Ma è possibile costruire questo ponte?

E se sì, a quale prezzo?

Prima di rispondere a queste domande è bene sottolineare quanto segue: **è evidente che non è possibile parlare di `luatex lunatic` nel modo di cui sopra per gli altri interpreti TeX; detto in altro modo `luatex lunatic` è possibile solo grazie a luatex.**

3 Motivazioni e obiettivi

TeX è un sinonimo di "portabilità" (è facile implementare/adattare TeX *the program*) e "stabilità" (TeX *the language* viene modificato solo per correggere errori, anche se ormai è congelato e gli eventuali bug sono considerati caratteristiche).

Possiamo riassumere dicendo che "la composizione con TeX aspira ad essere ovunque e per sempre".

Queste caratteristiche sono un po' inusuali nell'odierno scenario di sviluppo di applicazioni: nessuno si stupisce se un programma esiste per un solo sistema operativo (e a volte perfino per sistemi operativi non più sviluppati, date le varie tecniche di virtualizzazione) e soprattutto nessuno si stupisce dell'uscita di una nuova versione di un programma, il che significa, in effetti, correzione di errori e implementazione di nuove funzioni (da notare che l'inverso è di solito un fatto negativo: nessuna nuova uscita implica che lo sviluppo del programma è stato interrotto).

Naturalmente, se prendiamo in considerazione il sistema-LaTeX, ovvero LaTeX e i suoi pacchetti più usati, lo sviluppo di questo non è affatto congelato: basta controllare ogni giorno `http://www.mail-archive.com/ctan-ann@dante.de`. Inoltre pdfTeX viene costantemente modificato perché tenga il passo con le varie versioni del PDF.

Per `luatex lunatic` ho deciso di adottare questo punto di vista: ConTeXt-`mkiv` come strumen-

to per la pubblicazione di contenuti, con alcune funzioni di *content management* integrate.

In quanto strumento, non c'è da stupirsi se ci sono "frequenti" (per il mondo di TeX) nuove uscite, che possono essere dovute a un aggiornamento di LuaTeX, di ConTeXt-`mkiv`, di Python, o di una libreria di cui Python fornisce i relativi *binding*. E, naturalmente, non c'è da sorprendersi che tutto ciò rischi di diventare rapidamente ingestibile, se non viene trattato *cum grano salis*.

Il prezzo da pagare è una potenziale **perdita di stabilità**: lo stesso documento (con gli stessi font e le stesse immagini) compilato con una versione più recente può produrre un risultato differente (ecco perché sarei contrario a farne una funzionalità fondamentale di LuaTeX).

Per quanto riguarda la portabilità, per motivi dovuti a mancanza di risorse non rientra tra i miei progetti immediati sviluppare per sistemi operativi diversi da Linux; quindi è riscontrabile anche una potenziale **perdita di portabilità**.

Possiamo riassumere dicendo che "comporre con LuaTeX-lunatic è qui e ora", dove "qui" significa "per uno specifico sistema operativo" e "ora" significa "con questa versione". In questo momento "qui" significa "Ubuntu Linux 32 bit" e "ora" significa `luatex-snapshot-0.42.0.tar.bz2` con ConTeXt-`mkiv current 2008.07.17`; entrambi, probabilmente, saranno già sorpassati nel momento in cui quest'articolo andrà in stampa.

Naturalmente queste penalizzazioni devono essere inquadrate nell'ottica del sistema `TEX`: ovvero *stabilità e continuità*. L'autore ha realizzato la prima versione di `luatex lunatic` poco più di due anni fa, e sostanzialmente le differenze con quello attuale sono minime e dovute al notevole cambiamento (questo si) del codice di Luatex e del processo di building.

Durante lo sviluppo di `luatex lunatic` è emersa un'ulteriore motivazione: la possibilità di usare ConTeXt-`mkiv` come una specie di strumento per il *literate programming* adattabile a uno specifico contesto, una sorta di *meta-literate programming*.

È noto che CWEB è un sistema per "intrecciare" insieme un programma scritto in uno *specifico* linguaggio (`C`, in questo caso) e la sua documentazione, redatta in un linguaggio di markup *semplice*, TeX; ConTeXt-`mkiv` può essere usato per "intrecciare" insieme un programma scritto in un *qualsiasi* linguaggio e la sua documentazione, redatta con un linguaggio di markup *altamente estensibile*, ConTeXt-`mkiv`.

Detto in maniera diversa: oggi un applicazione "chiama" TeX o LaTeX (cioè crea un processo) per ottenere un risultato a partire da un frammento di codice `tex` (per esempio una formula matematica); invece ConTeXt-`mkiv` si estende runtime "incorporando" l'applicazione tramite caricamento dinamico (senza creare cioè un nuovo processo) per

FIGURA 1: Etichetta con codice a barre.

ottenere il risultato da inserire nel flusso di token analizzato da `luatex`.

Per esempio, durante la redazione di un testo di matematica, l'autore può usare ConTeXt-`mkiv` per comporre *e valutare* una formula matematica, tramite un *python binding* con una opportuna libreria di calcolo simbolico (ne esistono diverse, alcune sono in puro python come `sympy`).

Infine, risulta interessante costruire dei formati dedicati per compiti specifici, che magari richiedono l'integrazione di una/due particolari applicazioni esterne, senza però il bisogno di caricare funzionalità di alto livello sul lato TeX del sistema (METATeX in ConTeXt-`mkiv`[1]).

Si consideri ad esempio il problema di stampare etichette (dette moduli di produzione) come quelle di figura 1.

Si tratta essenzialmente di una tabella con un codice a barre e due o tre font (meglio se a spaziatura fissa), generalmente in bianco e nero. ConTeXt-`mkiv` già contiene il necessario per le tabelle e perfino un meccanismo di *layer* per posizionare oggetti in coordinate (x,y); `luatex lunatic` può fornire quanto serve per il codice a barre con il *binding* per Ghostscript ed il programma `barcode.ps`. Non c'è nessun bisogno di colori, interazioni, indici o divisioni del testo, eventuali esigenze grafiche anche complesse possono essere affrontate con il binding con l'interprete Postscript e/o con METAPOST.

Un'altra applicazione interessante per questo tipo di formati può essere quella che concerne programmi come Sage o R. Questi, attualmente, aggiungono ai rispettivi costrutti di calcolo matematico o statistico alcune istruzioni per ottenere report formattati con LaTeX.

1. ConTeXt permette di creare dei formati minimali di questo tipo a partire da `metatex.tex`. Citando dalla documentazione del file:
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`
In fondo assomiglia molto all'idea del formato `eplain`...

Con `luatex lunatic` ha senso ribaltare l'ottica: creare dei formati MetaTEXSage e MetaTEXR ad-hoc per elaborare a runtime codice `tex` che contiene parti di codice R o Sage da interpretare. Questa prospettiva è interessante se abbinata ad un sistema CMS come `Plone` al fine di creare un "laboratorio didattico" online di matematica o statistica di alto livello (ricordiamo che R e Sage sono framework usati per ricerca in ambito accademico).

4 Cenni alla fase di building

Questa sezione, di natura tecnica, è dedicata ad una traccia di installazione e descrizione dei problemi incontrati. Presuppone una certa familiarità con lo sviluppo e compilazione di codice; come già accennato, la piattaforma utilizzata è Linux Ubuntu 8.04 LTS 32 bit su i386. Un buon punto di riferimento è il testo di Levine (2000) ed il sito di Drepper.

La strategia è quella di installare il più possibile un sistema isolato, in modo analogo a `Sage Math`.

Fissata per convenienza la directory base `HOMEDIR=/opt/luatex/luatex-lunatic`

1. installiamo Python 2.6.1 in

   `$HOMEDIR/Python-2.6.1`

   ```
   $> ./configure \
   --prefix=/opt/luatex/luatex-lunatic\
   --enable-unicode=ucs4 \
   --enable-shared \
   $> export PATH=\
   /opt/luatex/lunatic-python/bin:$PATH
   ```

2. installiamo `luatex-snapshot-0.42.0` in `$HOMEDIR/luatex-snapshot-0.42.0` e creiamo il link simbolico

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

   Possiamo eventualmente compilare con `build.sh`, giusto per vedere se non ci sono intoppi.

   Installiamo ConTEXt-`mkiv` tramite

   `minimals` in `$HOMEDIR/minimals` ed attiviamola con

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

3. installiamo `lunatic-python`

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

   modifichiamo `setup.py` per adeguarlo alla nostra installazione:

   ```
   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
   >
   ```

   ed infine prepariamo il wrapper `python.lua`

   ```
   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
   ```

Ora, prima di installare, dobbiamo apportare le necessarie modifiche al codice per abilitare il dynamic loading.

Innanzitutto, dobbiamo permettere *anche* al nostro interprete python il dynamic loading, cioè di poter caricare librerie `*.so` a runtime; modifichiamo `loadlib.c`:

```
luatex/source/texk/web2c/
luatexdir/lua51/loadlib.c
69c69
< void *lib=dlopen(path, RTLD_NOW);
---
> void *lib=dlopen(path, RTLD_NOW|RTLD_GLOBAL);
```

Questo non è sufficiente:

il manuale `luatexref-t.pdf` (LuaTEX Development team, 2009) dice espressamente a pag 23:

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

"Abilitare" il dynamic loading non comporta rilevanti modifiche al codice: solo al file `liblua51.am`

```
luatex/source/texk/web2c/luatexdir/am/liblua51.am
12c12
< liblua51_a_CPPFLAGS += -DLUA_USE_POSIX
---
> liblua51_a_CPPFLAGS += -DLUA_USE_LINUX
```

ed al file `Makefile.in`

```
luatex/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
-fvisibility-inlines-hidden -ldl
```

In particolare, `-uluaL_openlibs` dice al linker di includere il simbolo `luaL_openlibs` anche se questo non è utilizzato da `luatex` standard, ed è proprio questo che permette in ultima analisi di poter utilizzare `loadlib`; l'altro switch `-ldl` permette il link con `libdl.so` per poter risolvere il simbolo `dlopen`. È evidente che gli sviluppatori hanno definito delle funzioni analoghe a `loadlib()` per poter permettere un binding selettivo statico, solo a tempo di compilazione, quindi senza la necessità di usare `dlopen`: si veda ad esempio `luaopen_mplib(L)` in `luastuff.c`:

```
luatex/source/texk/web2c/
 luatexdir/lua/luastuff.c
void luainterpreter(void)
    lua_State *L;
    L = lua_newstate(my_luaalloc, NULL);
    :
    /* our own libraries */
    luaopen_ff(L);
    luaopen_pdf(L);
    :
    luaopen_tex(L);
    luaopen_mplib(L);
```

L'ultima istruzione dice chiaramente che viene caricata `luaopen_mplib(L)` e linkata staticamente nell'eseguibile `luatex`.

Gli switch `-fvisibility=hidden` e `-fvisibility-inlines-hidden` ci introducono in un serio problema: la collisione tra simboli. Possiamo spiegarla con un esempio: `luatex` è linkato staticamente con `libpng.so vers 1.2.38`, la quale espone il simbolo `png_memcpy_check`, una funzione `C`. Supponiamo che a runtime l'interprete `python` esegua un binding con una diversa libreria `libpng.so 1.3` e che venga richiamato proprio il simbolo `png_memcpy_check`. Questo verrà risolto verso `libpng.so vers 1.2.38` ed il più delle volte questo causa segmentation fault e quindi abort del programma. Questo tipo di eccezione non si può gestire, ed è a runtime. Una soluzione consiste quindi nel nascondere i simboli di `luatex` e questo è piuttosto semplice: basta aggiungere al file `build.sh` di `luatex`

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

Il problema è che, cosi facendo, anche `loadlib()` viene resa inefficace; quindi bisogna "svelare" i simboli relativi a `Lua`.

La soluzione si può tranquillamente classificare come un *dirty trick* di basso livello: salviamo l'output della compilazione in `out`:

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

Localizzate le linee sensibili, togliamo il flag `-fvisibility=hidden`: ovvero

```
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/lua51
-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/lua51/lapi.c'
|| echo
'../../../source/texk/web2c/'`
 luatexdir/lua51/lapi.c
mv -f .deps/liblua51_a-lapi.Tpo
        .deps/liblua51_a-lapi.Po
```

diventa

```
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/lua51
-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/lua51/lapi.c'
|| echo
'../../../source/texk/web2c/'
`luatexdir/lua51/lapi.c
mv -f .deps/liblua51_a-lapi.Tpo
.deps/liblua51_a-lapi.Po
```

Ci sono circa una trentina di righe come questa, ed è un lavoro manuale di circa 20 minuti.

Dopo ricompiliamo `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 libff.a
libluamisc.a
libzzip.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
```

Siamo quindi pronti per `luatex lunatic`: copiamo l'eseguibile nella directory standard

```
$> cd $HOMEDIR/luatex/build/texk/web2c/luatex
$HOMEDIR/minimals/tex/texmf-linux/bin
```

e compiliamo

```
$> context --make
$> cp $HOMEDIR/lunatic-python
$> python setup.py build \
&& python setup.py install
```

Un test semplice: salviamo

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

in `test.tex` in una directory in cui è presente anche il wrapper `python.lua` ed eseguiamo

```
$> luatex --fmt=plain --output-format=pdf test.tex
```

5 Esempi

Assumeremo una installazione di `luatex lunatic` in `HOMEDIR=/opt/luatex/luatex-lunatic`. Il codice TEX non è inteso come didattico per ConTEXt-`mkiv`.

5.1 R

R "*is a free software environment for statistical computing and graphics*" (Gentleman e Ihaka, 2009). Si sta rapidamente diffondendo negli ambienti accademici (Cohen e Cohen, 2008); esiste un binding in python, `rpy2` (Gautier, 2009) che possiamo installare nella nostra `$HOMEDIR`.

Può essere necessario adeguare l'ambiente con

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

È conveniente distinguere la parte Python dalla parte TEX in questo modo: si salva il codice seguente in `test-R.py`

```
#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()
```

Da notare che si tratta di codice Python e si può testare a riga di comando — non è necessario ConTEXt-`mkiv`: questo ci preserva dal "fastidio" degli spazi, in quanto in Python il livello di indentazione è nella grammatica del linguaggio. La parte di presentazione ha una interfaccia in Lua, una in TEX, ed infine la parte `Main`; il codice Python `test-R.py` è visto come una libreria in Lua e caricato con `pyR  = python.import("test-R")`:

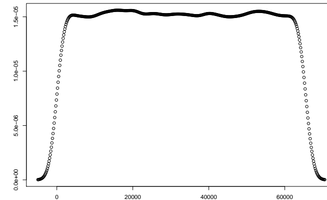
```
%% test-R.tex
%%
%% Lua interface
\startluacode
function testR(samples,outpdf,w,h,kernel)
  require("python")
  pyR  = python.import("test-R")
  dens = pyR.density(samples,outpdf,w,h,kernel)
  dens.run()
end
\stopluacode
%%
%% TeX interface

\def\plotdenstiy[#1]{%
\getparameters[R][#1]%
\expanded{\ctxlua{testR("u-random-int",
          "\Routpdf",10,7,"\Rkernel")}}%
%%
%% Main TeX
\setupbodyfont[sans,14pt]
\starttext
\startTEXpage
\plotdensity[outpdf={test-g.pdf},kernel={g}]
\plotdensity[outpdf={test-r.pdf},kernel={r}]
\plotdensity[outpdf={test-t.pdf},kernel={t}]
\plotdensity[outpdf={test-e.pdf},kernel={e}]
\plotdensity[outpdf={test-b.pdf},kernel={b}]
\plotdensity[outpdf={test-c.pdf},kernel={c}]
\plotdensity[outpdf={test-o.pdf},kernel={o}]
```

```
\hsize=400bp
This is a density plot of around {\tt 100 000}
random numbers between
$0$ and $2^{16}-1$ generated
from \type{/dev/urandom}
%
\externalfigure[test-g.pdf][width={400bp}]\par
{\ss gaussian}\par
\externalfigure[test-r.pdf][width={400bp}]\par
{\ss rectangular}\par
\externalfigure[test-t.pdf][width={400bp}]\par
{\ss triangular}\par
\externalfigure[test-e.pdf][width={400bp}]\par
{\ss epanechnikov}\par
\externalfigure[test-b.pdf][width={400bp}]\par
{\ss biweight}\par
\externalfigure[test-c.pdf][width={400bp}]\par
{\ss cosine}\par
\externalfigure[test-o.pdf][width={400bp}]\par
{\ss optcosine}
\stopTEXpage
\stoptext
```

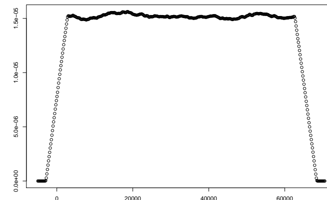
Il risultato si può vedere qui a fianco: sette grafici di densità di probabilità relativi alla stesso set di dati, ma con i sette diversi kernel (gaussian, rectangular, triangular, epanechnikov, biweight, cosine, optcosine) messi a disposizione da R.

Da notare che il codice di cui sopra fornisce il pdf delle dimensioni *esatte* per contenere grafici e testo: una semplice conversione in png o jpg rende il pdf immediatamente fruibile per altri contesti, come ad esempio una pagina html. Appare allora ragionevole pensare ad un processo che, ricevendo in ingresso un set di dati ed un set di kernel, "pubblichi" tramite un server web il risultato (ad esempio il pdf; in realtà una sua conversione bitmap) *con qualità tipografica tipica del TEX e affidabilità del risultato data da* R.

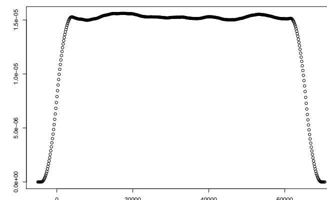
This is a density plot of around 100 000 random numbers between 0 and $2^{16}-1$ generated from /dev/urandom



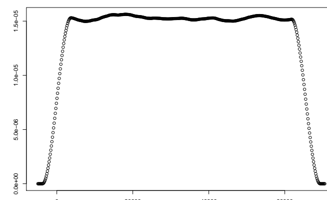
gaussian



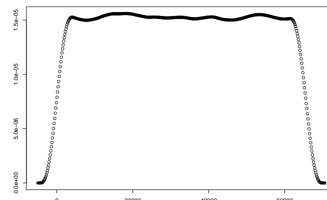
rectangular



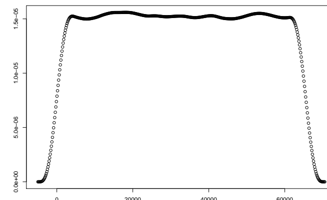
triangular



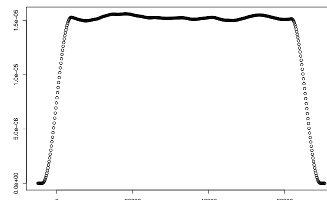
epanechnikov



biweight



cosine



optcosine

5.2 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.” (SAGE). Sage è l’acronimo di Software for Algebra and Geometry Experimentation.

In questo caso il setup di `lunatic python` deve far riferimento all’installazione python di SAGE, per cui è necessario modificare in modo adeguato il file di setup e ricompilare `lunatic python`. Anche la variabile d’ambiente `$PATH` deve essere modificata per includere `python` di SAGE.

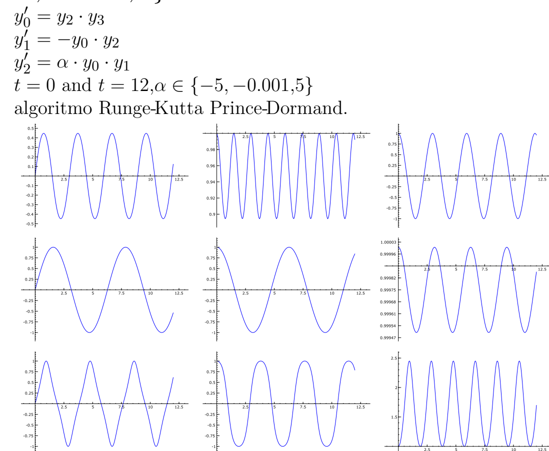
Ecco il codice python:

```
from sage.all_cmdline import *
def plot_eqn(diff_eqn,f0,f1,f2):
    T=ode_solver()
    T.algorithm="rk8pd"
    g_1 = eval("lambda t,y:%s" % diff_eqn)
    T.function=g_1
    T.y_0=[0,1,1]
    T.scale_abs=[1e-4,1e-4,1e-5]
    T.error_rel=1e-4
    T.ode_solve(t_span=[0,12],num_points=100)
    f = T.interpolate_solution()
    plot(f,0,12,).show(filename=f0)
    f = T.interpolate_solution(i=1)
    plot(f,0,12).show(filename=f1)
    f = T.interpolate_solution(i=2)
    plot(f,0,12).show(filename=f2)
```

E questo è il codice TEX:

```
\startluacode
function testODE(eqn,f0,f1,f2)
  require("python")
  ode = python.import("test-ode")
  ode.plot_eqn(eqn,f0,f1,f2)
end
\stopluacode
%
\starttext\startTEXpage
\hsize=12cm
\ctxlua{testODE(
        "[y[1]*y[2],-y[0]*y[2],-5*y[0]*y[1]]",
        "y0.pdf","y1.pdf","y2.pdf")}%
\ctxlua{testODE(
        "[y[1]*y[2],-y[0]*y[2],-0.001*y[0]*y[1]]",
        "ya0.pdf","ya1.pdf","ya2.pdf")}%
\ctxlua{testODE(
         "[y[1]*y[2],-y[0]*y[2],5*y[0]*y[1]]",
         "yb0.pdf","yb1.pdf","yb2.pdf")}%
$ y_0'=y_2 \cdot y_3 $
$ y_1'=-y_0 \cdot y_2 $
$ y_2'=\alpha\cdot y_0\cdot y_1$
$t=0$ and $t=12$,$\alpha\in\{-5,-0.001,5\}$ \\
algoritmo \hbox{Runge-Kutta} \hbox{Prince-Dormand}.
\hbox{\externalfigure[y0.pdf][width=4cm]%
\externalfigure[y1.pdf][width=4cm]%
\externalfigure[y2.pdf][width=4cm]}
\par
\hbox{\externalfigure[ya0.pdf][width=4cm]%
\externalfigure[ya1.pdf][width=4cm]%
\externalfigure[ya2.pdf][width=4cm]}
\par
\hbox{\externalfigure[yb0.pdf][width=4cm]%
\externalfigure[yb1.pdf][width=4cm]%
\externalfigure[yb2.pdf][width=4cm]}
\stopTEXpage\stoptext
```

Il risultato è la figura qui sotto: il plot delle 3 diverse soluzioni $(y_o, y_1, y_2)(\alpha)$ per $\alpha \in \{-5, -0.001, 5\}$:



Anche in questo caso è evidente la possibità di “esplorare” i risultati al variare di α (ad esempio), utilizzando un framework come Sage apprezzato dalla comunità scientifica.

5.3 Ghostscript

Con Ghostscript (Deutsch, 2009), noto interprete `PostScript`, introduciamo una interessante *feature*: l’utilizzo di `ctypes` di Python per la creazione di binding senza dover ricorrere a particolari framework come `SWIG` (Beazley, 2009b). Nello specifico, l’autore ha abbastanza facilmente creato il wrapper `testgs.py` dal programma `gslib.c` incluso nella distribuzione di Ghostscript, esponendo solo le funzioni per convertire eps/ps in pdf.

```
\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
```

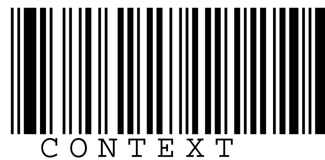
Il risultato è la classica figura `golfer.eps` che si vede alla pagina successiva: abbiamo *esteso* con un altro formato vettoriale il set delle figure gestite normalmente da LuaTEX (png, jpg, pdf).

golfer.eps

A prescindere dall'ottimo testo Casselman, una interessante applicazione è la libreria `barcode.ps` (Burton, 2009):

```
\startluacode
function barcode(text,type,width,
              height,options,savefile)
  require("python")
  gsmodule = python.import("testgs")
  w = width / 25.4
  h = height / 25.4
  H = h / w
  prologue = string.format("%%%%!\n%s\n",'gsave')
  scale = string.format("%f %f scale\n",w,w)
  barcode_string = \
string.format("1 12 moveto (%s) \
(%s height=%f) %s barcode showpage",
text,options,H,type)
  epilogue = string.format("\n%s\n", 'grestore')
  psfile = string.format("%s.ps",savefile)
  pdffile = string.format("%s.pdf",savefile)
  temp = io.open(psfile,'w')
  temp:write(prologue .. scale ..
tostring(barcode_string) .. epilogue,"\n")
  temp:flush()
  io.close(temp)
  ghost = gsmodule.gs()
  ghost.rawappendargs('-q')
  ghost.rawappendargs(string.format(
'-dDEVICEWIDTHPOINTS=%f', w*72*2.3+4))
  ghost.rawappendargs(string.format(
'-dDEVICEHEIGHTPOINTS=%f',h*72+4+24))
  ghost.rawappendargs('-dFIXMEDIA')
  ghost.rawappendargs('-dNOPAUSE')
  ghost.rawappendargs('-sDEVICE=pdfwrite')
  ghost.rawappendargs(string.format(
'-sOutputFile=%s',pdffile))
  ghost.rawappendargs('barcode.ps')
  ghost.InFile= psfile
  ghost.run()
end
\stopluacode

\def\PutBarcode[#1]{%
\getparameters[bc][#1]%
\ctxlua{barcode("\bctext","\bctype",\bcwidth,
\bcheight, "\bcoptions","\bcsavefile" )}%
\expanded{\externalfigure[\bcsavefile]}%
}
\starttext
\startTEXpage
{\PutBarcode[text={CODE 39},type={code39},
width=37,height=20,
options={includecheck includetext},
savefile={TEMP1}]}\\
{\ss code39}
\blank
{\PutBarcode[text={CONTEXT},type={code93},
width=37,height=20,
options={includecheck includetext},
savefile={TEMP2}]}\\
{\ss code93}
\blank
{\PutBarcode[text={977147396801},type={ean13},
width=37,height=20,
options={includetext},savefile={TEMP3}]}\\
{\ss ean13}
\stopTEXpage
\stoptext
```

code39



code93

ean13

Vi sono ovviamente (e per fortuna!) numerosi altri siti da esplorare; ad esempio `scipy` (SciPy), oppure `networkx` (Hagberg, 2009) per calcolo su network, `ROOT` (Wikipedia, 2009) e (The ROOT team, 2009), il framework per l'analisi dei dati del `CERN`, il `Natural Language Toolkit`, "*Open source Python modules, linguistic data and documentation for research and development in natural language processing*" (NLTK) e (Bird *et al.*, 2009). L'autore in Scarso (2009) ha per esempio una demo di piccolo sistema basato su `Berkeley DB XML` (Brian, 2006), `XQuery` (Walmsley, 2007) Python e LuaTeX per estrarre e stampare articoli da un dump di `wikiversity` (wikimedia downloads) e (Wikiversity) — un interessante approfondimento di diversi argomenti.

Numerosi i libri su Python e *scientific computing* come Langtangen (2009b) e Langtangen (2009a), mentre per l'aspetto generale della pro-

grammazione certamente il prossimo Lutz (2009) e l'ottimo reference Beazley (2009a).

6 Conclusioni

Possiamo ritenere soddisfacente il set di patch al codice base, in quanto sono esigue e ben individuabili. La fase di building però è pesantemente influenzata dal problema del symbol collision, e certamente può essere migliorata. Tuttavia la fase attuale di intenso sviluppo non consiglia di proseguire ad ulteriori ottimizzazioni, in quanto potrebbero rivelarsi inutili: non è escluso infatti che `loadlib()` venga resa accessibile dal team di sviluppo, ed in questo caso dovranno essere risolti i problemi relativi alla portabilità (ad esempio quelli su piattaforma Windows/i386), alla stabilità (eccezioni non intercettabili tramite i meccanismi usuali di Lua) ed alla sicurezza (abilitazione di `loadlib` runtime, come `\write18`), decisioni che evidentemente l'autore non può prendere.

Non è sostenibile la posizione di usare Python per sostituire o completare in qualche maniera Lua nella sua funzione di interfaccia con TEX: Lua e TEX sono un perfect match. Lua è stabile, leggero, veloce, sviluppato in ambiente accademico, saldamente ancorato al `C`, con una comunità consistente. Non tutte queste caratteristiche sono condivise da Python: ad esempio, Python è piu lento, più ingombrante e meno stabile — attualmente circolano le vers. 2.4, 2.5, 2.6, 2.7alpha e 3.1.1, e Google sta studiando una release intermedia tra 2.5 e 2.6 .

Un binding di Python ad una libreria è importante solo se non esiste un analogo binding di Lua e non si ritiene conveniente costruirne uno. In definitiva un binding di Python è da privilegiare se è completo, ricco, stabile e mantenuto, oppure perché si utilizza massivamente Python anche in contesti completamente differenti (come ad esempio `SAGE`); altrimenti è opportuno pensare ad un Lua binding, magari ottenuto con `SWIG` (e potrebbe essere il caso di `R`).

È certamente vero che Python "offre" più di Lua, ma questa offerta deve essere calibrata bene, altrimenti si rischia di trasformare ConTEXt-`mkiv` in un tool molto potente che fa tutto — ma di cui non si ha un completo controllo.

Così ad esempio non deve essere sottovalutata la presenza di METAPOST: esistono numerosissime librerie di METAPOST che coprono praticamente ogni esigenza. Ad esempio, i grafi possono essere disegnati con Metagraph in METAPOST; se si vuole un binding in Lua per `graphviz` (Graphviz) si può usare Leuwer (2009). `NetworkX` o `igraph` (Csárdi e Nepusz, 2009) sono interessanti se bisogna eseguire *calcoli* sui grafi (per esempio il minimun spanning tree), non meramente per disegnare.

Ancora: nella libreria standard di Python viene fornito `sqlite` (Hipp, 2009), un embedded relational database molto apprezzato dalla comunità. Quindi se si ha una sorgente dati in `sqlite`, `luatex lunatic` appare un buon candidato. Ma non è certamente una buona strategia utilizzare `sqlite` per gestire il multipassing, ad esempio; esistono già tecniche consolidate in LATEX e ConTEXt , e comunque la `table` di Lua è ottima anche per queste problematiche, senza contare la libreria `lpeg` di ConTEXt-`mkiv` — un parser per expression grammar, paragonabile alle Context-free Grammar.

ConTEXt-`mkiv` con `luatex lunatic` è uno strumento estremamente potente — *ma la potenza è nulla senza il controllo.*

7 Ringraziamenti

Desidero ringraziare Taco & Hans per questi primi 3 anni di LuaTEX— è questo il "qualcosa di completamente differente".

Inoltre sono estremamente riconoscente a Massimiliano Max Dominici per aver tradotto in italiano una parte del mio articolo originale in inglese — un pessimo inglese, un'ottima traduzione. Quest'articolo è anche merito suo.

Infine un grazie al professore Claudio Beccari, per il suo lavoro passato e presente — e per quello futuro — ed a Gianluca Pignalberi per il suo paziente lavoro sulla mia bibliografia.

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

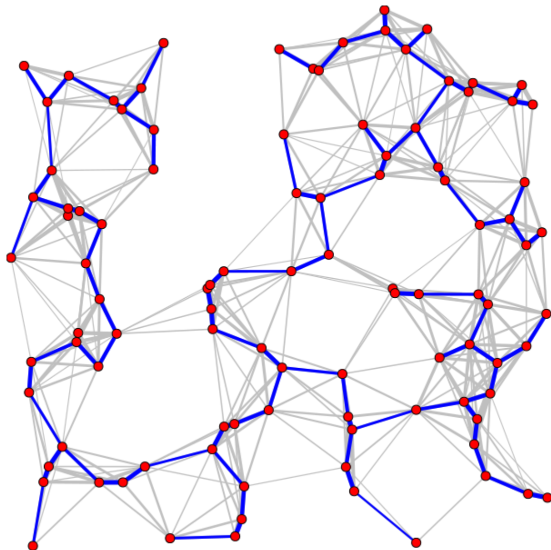
Cool. Thanks for letting me know about this. I've cc'd this to the Sage-devel mailing list.

William

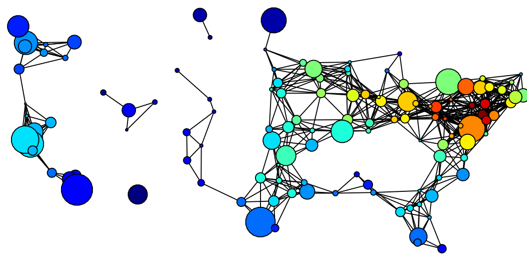
— William Stein Associate Professor of Mathematics University of Washington

http://groups.google.com/group/sage-devel/browse_thread/thread/b4e9c3f6080e887d#

Just for fun (giusto per i fan...)



igrah — Minimum Spanning Tree



NetworkX — miles_dat.txt

fontforge — outlines

Riferimenti bibliografici

Beazley, D. M. (2009a). *Python Essential Reference*. Addison-Wesley Professional, 4ª edizione.

— (2009b). «SWIG». URL `http://www.swig.org`.

Beccari, C. *Introduzione all'arte della composizione tipografica con LATEX*. URL `www.guit.sssup.it/downloads/GuidaGuIT.pdf`.

Bird, S., Klein, E. e Loper, E. (2009). *Natural Language Processing with Python — Analyzing Text with the Natural Language Toolkit*. O'Reilly Media.

Brian, D. (2006). *The definitive Guide to Berkeley DB XML*. Apress.

Burton, T. (2009). «Barcode Writer in Pure PostScript». URL `http://www.terryburton.co.uk/barcodewriter/`.

Casselman, B. *Mathematical Illustrations A Manual of Geometry and PostScript*. URL `http://www.math.ubc.ca/~cass/graphics/text/www/index.html`.

Cohen, Y. e Cohen, J. (2008). *Statistic and Data with R*. Wiley.

Csárdi, G. e Nepusz, T. (2009). «The igraph library». URL `http://igraph.sourceforge.net`.

Deutsch, L. P. (2009). «Ghostscript». URL `http://ghostscript.com/`.

Drepper, U. «How to Write Shared Libraries». URL `http://people.redhat.com/drepper/dsohowto.pdf`.

Gautier, L. (2009). «A simple and efficient access to R from Python». URL `http://rpy.sourceforge.net/`.

Gentleman, R. e Ihaka, R. (2009). «The R Project for Statistical Computing». URL `http://www.r-project.org/`.

Graphviz (2009). «Graphviz - Graph Visualization Software». URL `http://www.graphviz.org`.

Hagberg, A. (2009). «NetworkX». URL `http://networkx.lanl.gov`.

Hagen, H., Henkel, H. e Hoekwater, T. (2009). «LuaTEX documentation». URL `http://www.luatex.org/documentation.html`.

Hipp, D. R. (2009). «SQLite». URL `http://sqlite.org/`.

Ierusalimschy, R. (2006). *Programming-in-lua*. 2ª edizione. URL `Lua.org`.

Langtangen, H. P. (2009a). *A Primer on Scientific Programming with Python*. Springer.

— (2009b). *Python Scripting for Computational Science*. Springer.

Leuwer, H. (2009). «LuaGRAPH - Reference Graph Programming with Lua». URL `http://luagraph.luaforge.net/graph.html`.

Levine, J. (2000). *Linkers & Loaders*. Morgan Kaufmann Publisher.

LuaTEX Development team (2009). *LuaTEX Reference*. `luatexref-t.pdf`. Available in `manual` folder of `luatex-snapshot-0.42.0.tar.bz2`.

Lutz, M. (2009). *Learning Python*. O'Reilly, 4ª edizione.

Metagraph (2009). «METAGRAPH: (un)directed graphs with METAPOST». URL `http://vigna.dsi.unimi.it/metagraph`.

Niemeyer, G. (2009). «Lunatic Python». URL `http://labix.org/lunatic-python`.

NLTK (2009). «Natural Language Toolkit». URL `http://www.nltk.org/`.

SAGE (2009). «SAGE». URL `http://www.sagemath.org/`.

Scarso, L. (2009). «User:Luigi.scarso/luatex lunatic». URL `http://wiki.contextgarden.net/User:Luigi.scarso/luatex_lunatic`.

SciPy (2009). «SciPy». URL `http://www.scipy.org/`.

The ROOT team (2009). «ROOT: A Data Analysis Framework». URL `http://root.cern.ch`.

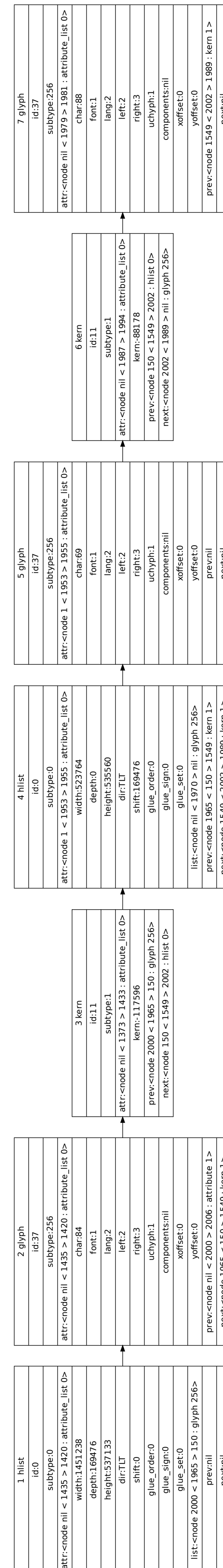
Walmsley, P. (2007). *XQuery*. O'Reilly.

wikimedia downloads (2009). «Wikimedia Downloads». URL `http://download.wikimedia.org`.

Wikipedia (2009). «ROOT». URL `http://en.wikipedia.org/wiki/ROOT`.

Wikiversity (2008). «Getting stats out of Wikiversity XML dumps». URL `http://en.wikiversity.org/wiki/Getting_stats_out_of_Wikiversity_XML_dumps`.

▷ Luigi Scarso
Logo S.r.l.



graphviz — nodelist