Extending ConTEXt MKiV with PARI/GP

Abstract
This paper shows how to build a binding to PARI GP, the well known computer algebra system, for ConTEXt MKiV, showing also some examples on how to solve some common basic algebraic problems.

Keywords
LuaTeX, ConTeXt MARKIV, binding, PARI/GP

Introduction

PARI/GP[1] is a relatively small computer algebra system that comes as C library (`libpari`) and an interpreter (gp) for its own language (GP) built on upon the same library. Although it has discrete capabilities on symbolic manipulations, it has an extensive algebraic number theory module and hence it can do, due to the highly optimised C library, complex numeric calculations very quickly and accurately. In this paper we will show a way to 'extend' ConTEXt MKiV with PARI/GP and some examples on how to use this powerful library. PARI stands for 'Pascal ARIthmétique' (the very first choice was the Pascal language, dropped soon for C), while GP originally was GPC for 'Great Programmable Calculator', but the C was dropped for unknown reason. The current stable version is `2.3.4`.

Build the Lua binding

It's well known that ConTEXt MKiV uses LuaTEX as a typesetting engine, but maybe it's little known to the tex-user that Lua itself is used either as *embedded language* to enhance an application with a simple but powerful high level language (e.g. to build plug-ins) or as *glue language* to 'connect' several libraries, mostly written in C\C++ — exactly the same as in Sage[2], where the glue language is Python. In the latter case Lua is *extended* with the new libraries that become practically Lua modules (i.e. modules written in native Lua language) and they can be built in into the `lua` interpreter at compile time (as in the GSL Shell[3] program) or loaded at runtime, which is the case of the extension of this paper.

Most often it's necessary to write some C code that acts as an interface between the library and Lua: this process is called 'build the Lua binding for the library' and it's where the developer decides which symbols of the library (i.e. functions, classes, variables and constants) export to Lua and how they are seen from the Lua side (under which name, for example). This is a delicate phase, because one must know the conventions of the `lua` library on which the Lua language relies (the '`lua` Application Program Interface' or API), the API of the target library and write the appropriate C code for each symbol to export: for the C language these APIs are usually organized in so-called *header files* (with suffix `".h"`) that contain the declarations of each function, variable or constant that the library exposes — but not always all of them can be exported: the developer must consult the documentation to know which set of admissible symbols to export).

Luckily the `lua` API are completely listed in the Lua book[4] and they describe a simple and robust mechanism: basically every C function that wants to interact with the Lua interpreter uses a stack (a LIFO queue) to exchange data. The stack is modified by a relatively small set of functions that act on the *Lua state*, a global data structure that also keeps track of unused objects and calls the garbage collector when necessary. Hence every C function of the binding must only take care of calling the right function of the target library with the right arguments and to use the stack to exchange the in (input to the function) and/or out (output to the Lua interpreter) values. If the target library has many functions this is a long and tedious work, because most of the time the functions follow few common patterns and most of the binding code can be cut-and-pasted with few modifications, but on average the headers of the target library are difficult to read.

This is where SWIG enters the play. SWIG (*Simplified Wrapper and Interface Generator*, see[5]) is a program to help the developer to build bindings and, for some libraries, it can almost automatically build a binding by merely reading all headers files. SWIG reads a driver file, the so-called *interface file* `".i"`, and it executes its instructions. For `libpari` the instructions in the interface file `pari.i` are quite simple: basically 'read the headers and produce the binding'. This is for example the role of the `%include "pari/paritype.h";` instruction, that just says 'read the header paritype.h which is in the pari folder and write the binding code'; but we can also map some `libpari` functions into something else, as in

```
GEN uti_mael2(GEN m,long x1,long x2)
{return mael2(m,x1,x2);}
```

where the `libpari` macro `mael2` is wrapped into the C function `uti_meal2` for sake of simplicity.

The binding is then built with

```
swig  -lua pari.i
```

This is the complete interface file `pari.i` used under Linux 32 bit: the header files are in the sub-folder `pari` of the folder that contains the build script.

```
%module pari
%{
#include "pari.h"
ulong overflow;
%}

%ignore gp_variable(char *s);
%ignore setseriesprecision(long n);
%ignore  killfile(pariFILE *f);

%include "pari/paritype.h";
%include "pari/parisys.h";
%include "pari/parigen.h";
%include "pari/paricast.h";
%include "pari/paristio.h";
%include "pari/paricom.h";
%include "pari/parierr.h";
%include "pari/paridecl.h";
%include "pari/paritune.h";
%include "pari/pariinl.h";

%inline %{
GEN uti_mael2(GEN m,long x1,long x2)
{return mael2(m,x1,x2);}
GEN uti_mael3(GEN m,long x1,long x2,long x3)
{return mael3(m,x1,x2,x3);}
GEN uti_mael4(GEN m,long x1,long x2,long x3,
long x4)
{return mael4(m,x1,x2,x3,x4);}
GEN uti_mael5(GEN m,long x1,long x2,long x3,
long x4,long x5)
{return mael5(m,x1,x2,x3,x4,x5);}
GEN uti_mael(GEN m,long x1,long x2)
{return mael2(m,x1,x2);}
GEN uti_gmael1(GEN m,long x1)
{return gmael1(m,x1);}
GEN uti_gmael2(GEN m,long x1,long x2)
{return gmael2(m,x1,x2);}
GEN uti_gmael3(GEN m,long x1,long x2,long x3)
{return gmael3(m,x1,x2,x3);}
GEN uti_gmael4(GEN m,long x1,long x2,long x3,
long x4)
{return gmael4(m,x1,x2,x3,x4);}
GEN uti_gmael5(GEN m,long x1,long x2,long x3,
long x4,long x5)
{return gmael5(m,x1,x2,x3,x4,x5);}
GEN uti_gmael(GEN m,long x1,long x2)
{return gmael2(m,x1,x2);}
GEN uti_gel(GEN m,long x1)
{return gmael1(m,x1);}
GEN uti_gcoeff(GEN a,long i,long j)
{return gcoeff(a,i,j);}
GEN uti_coeff(GEN a,long i,long j)
{return coeff(a,i,j);}
%};
```

The binding is quite straightforward: almost every symbol of `libpari` has a counterpart in `Lua` with the same name; the symbols 'private' are exposed in `paripriv.h` which is not listed in `pari.i` — they aren't exported and hence they are not reachable from `Lua`.

The build script (for Linux) assumes the latest SWIG and PARI/GP installed under `/opt/swig-2.0.2`:

```
/opt/swig-2.0.2/bin/swig  -lua pari.i
gcc -ansi \
    -I./pari -I/opt/swig-2.0.2/include \
    -c pari_wrap.c  -o pari_wrap.o
gcc -Wall -ansi -shared -I./pari \
    -I/opt/swig-2.0.2/include -L./ \
    -L/opt/swig-2.0.2/lib pari_wrap.o \
    -lpari -lm -o pari.so
```

Once compiled, the `pari.so` is suitable to be loaded as `Lua` module with `require("pari")`.

As a final note for this section, the same steps can be followed under Windows using MinGW[6] or with GUB[7] to cross-compile the library in a host system (Linux) for a target system (Windows) — as is the case of this paper, where the examples use a cross-compiled `dll libpari`.

Examples

Summations

As we said briefly in the introduction, PARI/GP has its own language GP, with more than 450 functions, and its interpreter, the gp program. Most of the time these functions are one-to-one with the functions exported by the library `libpari`, but sometimes there are some 'sugar syntactic' constructs for the sake of simplicity. In any case, `libpari` has the `gp_read_str(char *)` function that evaluates a GP sentence and returns the result, so that on the Lua side it's possible to use both the library and the GP language. The library is usually quicker than GP and it has a finer grain control — which usually also means that it's necessary to write more code.

In this first example, we will see how to calculate exactly a summation. The GP function is `sum(X,a,b,expr,start)` that stands for $\sum_{X=a}^{b} \mathrm{expr}(X,\cdot)$, where `start` is the initial value of $\mathrm{expr}(X,\cdot)$:

```
\startluacode
require("pari")
pari.pari_init(4000000,500000)
document = document or {}
document.lscarso= document.lscarso or {}
local function sum(X,a,b,expr,start)
   local avma = pari.avma
   local start = start or '0.'
   local res =
    pari.gp_read_str(string.format(
     "sum(%s=%s,%s,%s,%s)",X,a,b,expr,start))
   res = pari.GENtoTeXstr(res)
   pari.avma = avma
   return res
end
document.lscarso.sum = sum
\stopluacode
```

```
\starttext
\startTEXpage
\startformula
\sum_{k=0}^{30}\frac{4(-1)^k}{2k+1}=
  \ctxlua{context(document.lscarso.sum(
   "k",0,30,"4*(-1)^k/(2*k+1)","0"))}
\stopformula
\stopTEXpage
\stoptext
```

that gives

$$\sum_{k=0}^{30}\frac{4(-1)^k}{2k+1}=\frac{58630135791001973169852284}{18472920064106597929865025}$$

Let's explain the code step by step. First we need to load the module with `require("pari")` — assuming that the library is in the standard path or in the current folder (cfr. `CLUAINPUTS` in [8] for more details).

Next, we must avoid conflicts with other Lua functions. A common solution is to define a namespace (`document.lscarso` in this case), a local function (`sum(X,a,b,expr,start)`) and expose it with the namespace (`document.lscarso.sum= sum`). This is a general issue when one defines its own module, not only for PARI/GP — it's the same problem of redefining TEX macros.

There is another issue with PARI/GP. Like Lua, PARI/GP also uses a stack but it has not a garbage collector, and every time it makes a calculation the result is not deleted; after a while the stack is full and the process aborts. Luckily it's easy to clear the stack: at the beginning of every function it's sufficient to record the initial position on the stack with `local avma=pari.avma` and then reset the stack with `pari.avma=avma` just before the return statement of the function. This is an issue with `libpari`, because most of GP functions manage the stack correctly.

After these notes, calling the GP `sum` function is a matter of calling `gp_read_str(char *)` with the right formatted string which is trivial thanks to `string.format`, a standard LuaTEX function. Last but not least is `pari.GENtoTeXstr(GEN)`, a `libpari` function that translates a pari object (e.g a fraction) into a TEX expression. It's important to note that the result is exact because we have imposed with `start=0` that all the values are in **Q**: if we want an approximated value just use `start=0.` and the result is

$$\sum_{k=0}^{30}\frac{4(-1)^k}{2k+1}=3.1738423371907494086902241\,40$$

But we can do things a bit better. First, we want to control the *precision* of the result, i.e. how many digits to show. This is quite simple: the GP `default(.,.)` function can be used to get/set some internal constants and `realprecision` is what we need:

```
local function set_precision(prec)
   local avma = pari.avma
   local prec = math.floor(prec+0.5) or 28
   local res = pari.gp_read_str(
    string.format("default(realprecision,%s)",
      prec))
   res = pari.GENtostr(pari.gp_read_str(
    "default(realprecision)"))
   pari.avma = avma
   return res
end
```

```
local function get_precision(prec)
   local avma = pari.avma
   local res = pari.GENtostr(
    pari.gp_read_str(
     "default(realprecision)"))
   pari.avma = avma
   return res
end
```

Once we have the notion of precision, we can extend the summation to 'infinity', i.e. until the partial sums are stable within the precision. Of course this depends on the character of the series — in our case it's an alternating series. For this kind of series GP has the `sumalt(X=a,expr)` function that does the job:

```
local function sum_alternate(X,a,expr,prec)
   local avma = pari.avma
   local gp = document.lscarso.get_precision
   local oldprec = gp(prec)
   document.lscarso.set_precision(prec)
   local res=pari.GENtostr(pari.gp_read_str(
    string.format("sumalt(%s=%s,%s)",
                           X,a,expr)))
   document.lscarso.set_precision(oldprec)
   pari.avma = avma
```

```
  res=string.gsub(res,"(%d)","%1\\hskip0sp")
  return res
end
```

We can hence try to calculate the series with a precision of `800` digits:

```
\startformula
\sum_{k=0}^{\infty}\frac{4(-1)^k}{2k+1}=
\stopformula
\ctxlua{context(
  document.lscarso.sum_alternate(
  "k",0,"4*(-1)^k/(2*k+1)",800))}
```

Given that the result is quite long (see fig.1) with `string.gsub(res,"(%d)","%1\\hskip0sp")` we insert an invisible skip to help TEX to break the expression.

$$\sum_{k=0}^{\infty}\frac{4(-1)^k}{2k+1}\approx$$

3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862803482534211706798214808651328230664709384460955058223172535940812848111745028410270193852110555964462294895493038196442881097566593344612847564823378678316527120190914564856692346034861045432664821339360726024914127372458700660631558817488152092096282925409171536436789259036001133053054882046652138414695194151160943305727036575959195309218611738193261179310511854807446237996274956735188575272489122793818301194912983367336244065664308602139494639522473719070217986094370277053921717629317675238467481846766940513200056812714526356082778577134275778960917363717872146844090122495343014654958537105079227968925892354201995611212902196086403441815981362977477130996051870721134999999837297804995105973173281609631860

Figure 1. Evaluation of an alternating series with 800 digit precision.

Of course this is a well known series: from $\mathit{arctan}(1) = \frac{\pi}{4}$ one can calculate the Taylor expansion of $\mathit{arctan}(x)$ around $x = 0$ with `taylor(expr,x)`:

```
local function taylor(expr,x)
  local avma = pari.avma
  local res = pari.gp_read_str(
   string.format("taylor(%s,%s)",expr,x))
  res = pari.GENtoTeXstr(res)
  pari.avma = avma
  return res
end
```

```
$\mathrm{arctan}(x)=$
\startformula
\ctxlua{context(document.lscarso.taylor(
        "atan(x)","x"))}
\stopformula
```

i.e.

$\mathrm{arctan}(x) =$

$$x - \frac{1}{3}x^3 + \frac{1}{5}x^5 - \frac{1}{7}x^7 + \frac{1}{9}x^9 - \frac{1}{11}x^{11} + \frac{1}{13}x^{13} - \frac{1}{15}x^{15} + O(x^{16})$$

The series is convergent in $x = 1$ (there are several proofs about this, e.g. see [9]), hence

$$4\sum_{k=0}^{\infty}\frac{(-1)^k}{2k+1} = \sum_{k=0}^{\infty}\frac{4(-1)^k}{2k+1} = 4\frac{\pi}{4} = \pi \quad .$$

It's important to note that theoretically this series has a slow convergence to π (it's hence a bad choice to calculate π) but *practically* it can be used with PARI/GP to give quickly an high precision result — this is the power of the library.

Before continuing, let's consider this summation:

```
\startformula
\sum_{k=0}^{3}\frac{1}{x^2+k}=
\ctxlua{context(document.lscarso.sum(
  "k",0,3,"1/(x^2+k)","0"))}
\stopformula
```

that gives

$$\sum_{k=0}^{3}\frac{1}{x^2+k} = \frac{4x^6+18x^4+22x^2+6}{x^8+6x^6+11x^4+6x^2}$$

PARI/GP is also capable of some symbolic calculations — it's not only a numeric library.

Continued fractions

A simple *finite* (canonical) continued fraction is a rational number q given by

$$q = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{\ddots + \cfrac{1}{a_n}}}}$$

where a_0 is an integer and $a_{j,j>0}$ are strictly positive integers. Every rational number can be expressed with a finite continued fraction; if we consider a succession of finite continued fractions for $n \to \infty$ we have an *infinite* (canonical) continued fraction, and every irrational number has an unique infinite continued fraction. For a finite c.f. $[a_0, a_1, a_2, ..., a_n]$ the rational number given by calculating all the intermediate fractions is usually termed as p_n/q_n i.e. $[a_0,a_1,a_2,...,a_n] = \frac{p_n}{q_n}$. For example $[0,3] = \frac{1}{3}$ and it's possible to show that p_n/q_n is the fraction in lowest terms. The GF `contfrac` function calculates (the vector of) the continued fraction of a rational number, while `contfracpnqn` given a (finite vector of) continued fraction returns p_n, q_n but the interesting point here is to use, given a *real* number with a fixed precision, the continued fraction to find its best rational approximation. The `libpari bestappr(x,A)` function calculates exactly what we need:

```
local function bestappr(x,A)
   local avma = pari.avma
   local x = tostring(x) or nil
   local A = math.floor(A+0.5)
   local res, bestx
   if x == nil then return nil,nil end

   bestx = pari.bestappr(pari.geval(
     pari.strtoGENstr(x)),
     pari.geval(
      pari.strtoGENstr(tostring(A))))
   res = {}
   res[1] = pari.GENtostr(bestx)
   res[2] = pari.GENtostr(
             pari.uti_gel(bestx,1))
   res[3] = pari.GENtostr(
             pari.uti_gel(bestx,2))
   pari.avma = avma
   return res[1],res[2],res[3]
end
```

Note that the return value is an array with 3 components, namely $p_n/q_n, p_n, q_n$. We also use `pari.uti_gel`, the *wrapped* version of `libpari gel` function, to access an array by components.

Instead of an arbitrary real number, we choose π because the libpari `mppi(long)` function gives π with the required precision .

```
\startluacode
local collect = {}
local avma = pari.avma
local prec = 800
document.lscarso.set_precision(prec)
avma = pari.avma
local pi = pari.mppi(prec)
local pi_str = pari.GENtostr(pi)
pari.avma = avma
--print("=====>pi:",pi_str)
for d = 4,50000,1 do
   res,num,den =
     document.lscarso.bestappr(pi_str,d)
   collect[res] = {num,den,d}
end
context("\\starttabulate[|l|l|]")
context("\\HL")
context(string.format(
         "\\NC %s \\NC %s \\NC\\NR",
        "fraction","approx. value"))
context("\\HL")
for k,v in pairs(collect) do
   print( k, v[1]/v[2],v[3])
   -- context(k, v[1]/v[2],v[3])
   context(string.format(
    "\\NC %s \\NC %s \\NC\\NR",k,v[1]/v[2]))
end
context("\\stoptabulate")
\stopluacode
```

Note that we use p_n,q_n as a key for the dictionary `collect`, so we have just the set of results – i.e. we drop the same best approximations for different denominators. For a precision of 800 digits and a range of denominators between 4 and 50000 we have hence:

fraction	approx. value
333/106	3.1415094339623
104348/33215	3.1415926539214
16/5	3.2
13/4	3.25
22/7	3.1428571428571
355/113	3.141592920354
19/6	3.1666666666667
103993/33102	3.1415926530119

where the approx. values are due to the Lua floating point math.

Equations

Solving numeric equations in PARI/GP required more attention than other packages. The `solve(X=a,b,expr)` GP function implements a very good algorithm but it works with one variable only and it fails if `expr` is not defined in $[a,b]$ and it hasn't a *variation* in $[a,b]$. This Lua wrapper `solve` tries to ensure that at in $[a,b]$ there is a variation evaluating the sign of `expr(a)*expr(b)`:

```
function solve(expr,X,a,b,prec)
   local av = pari.avma
   pari.gp_read_str(
    string.format(
     "default(realprecision,%s)",prec))
   local tr,res
   pari.gp_read_str(string.format("f(%s)=%s",
                  X,expr))
   tr = pari.gp_read_str(
    string.format(
     "if(f(%s)*f(%s)<0,1,0)",a,b))
   tr = pari.GENtostr(tr)
```

```
   tr = tonumber(tr)
   res = nil
   if (tr==1) then
    local expr=string.format(
     "solve(%s=%s,%s,%s)",X,a,b,expr)
    res = pari.gp_read_str(expr)
    res = pari.GENtostr(res)
   end
   return res,
    pari.GENtoTeXstr(
     pari.strtoGENstr(expr))
end
```

The next code tries to solve

$$x^5 + x^3 \arctan(x) + 2x^2 + x + 1 = 0$$

for $x \in [-100,100]$ with a precision of 12 digits:

```
\startluacode
local solve = document.lscarso.solve
for a=-100,99,1 do
 local res,TeX,aa,bb =
   solve('x^5+atan(x)*x^3+2*x^2+x+1',
    "x",a,(a+1),12)
 if res  ~= nil then
  context(string.format(
   "$%s\\approx 0$ \\crlf
     for $x\\approx%s$\\par",
    TeX,res))
 else
     -- print("TeX="..TeX)
 end
end
\stopluacode
```

We have hence:
$x^5 + atan(x) * x^3 + 2 * x^2 + x + 1 \approx 0$
for $x \approx -1.47704735548$

PARI/GP has a rich set of functions for polynomials, and solve is not necessarily the best choice to find the roots of multivariate polynomials; the next example will show how to draw the real roots of P[X,Y] with a given precision in a square region $[a,b] \times [a,b]$. First of all, we need to understand that with a fixed precision there is also an associated `zero`: with `precision=12` then `zero=1E-96`. Next, PARI/GP finds the complex roots of a univariate polynomial, so we need a `get_value` wrapper to evaluate $P(x,y)$ for $y \in [a,b]$ (with a given precision), so we have an expression in the x variable that we will consider as a polynomial P[X]:

```
local function get_value(expr,X,a,prec)
   local avma = pari.avma
   pari.gp_read_str(string.format(
    "default(realprecision,%s)",prec))
   pari.gp_read_str(string.format(
    "%s=%s",X,a))
   local res = pari.gp_read_str(
    string.format("eval(%s)",expr))
   res  = pari.GENtostr(res)
   pari.avma = avma
   return res
end
```

The `polroots` function evaluates the roots and returns an array of roots where each root is separated into the real and complex components:

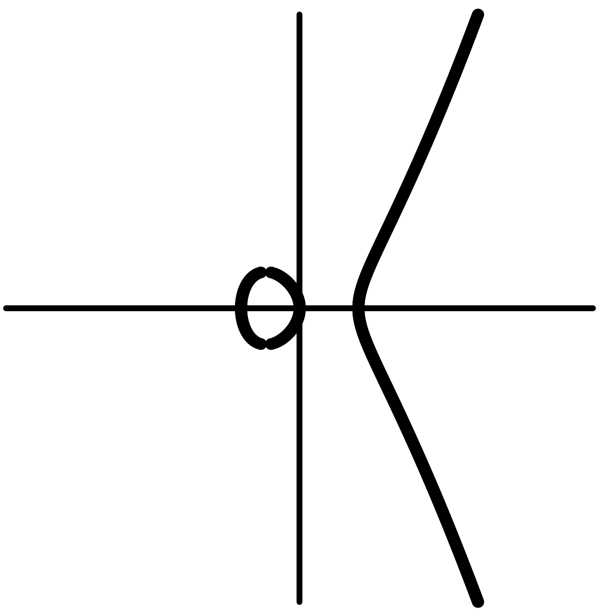
```
local function polroots(poly,prec)
   local avma = pari.avma
   pari.gp_read_str(string.format(
     "default(realprecision,%s)",prec))
   local poly = tostring(poly)
   local prec = tonumber(prec)
   local degree = pari.degree(
      pari.geval(pari.strtoGENstr(poly)))
   local roots = pari.roots(
    pari.geval(pari.strtoGENstr(poly)),prec)
   local res ={}
   for i=1,degree do
     local real_part,im_part =
      pari.GENtostr(pari.uti_gel(
      pari.uti_gel(roots,i),1)),
      pari.GENtostr(pari.uti_gel(
      pari.uti_gel(roots,i),2))
     res[#res+1]={real_part,im_part}
   end
   pari.avma = avma
   return res
end
```

Last we need to iterate y over $[a,b]$ and find the roots of P[X]. Instead of producing a table, we plot the value by a MetaPost page:

```
\startluacode
local poly = "x^3-x-y^2"
local step= 1/2^6
local results = {}
local limit = 5
local zero = '0.E-96'
local prec = 12
get_value = document.lscarso.get_value
polroots = document.lscarso.polroots
context("\\startMPpage")
context("pickup pencircle scaled 0.1pt;")
context(string.format("draw (-%s,0)--(%s,0);",
    limit,limit))
context(string.format("draw (0,-%s)--(0,%s);",
    limit,limit))
context("pickup pencircle scaled 0.2pt;")
for y=-limit,limit,step do
 local poly_x = get_value(poly,'y',y,prec)
 -- print("poly_x="..poly_x,y)
```

```
 local roots = polroots(poly_x,prec)
 for _,root in pairs(roots) do
  local real,imag = root[1],root[2]
  -- print("real="..real,"imag="..imag)
  if imag == zero then
     if real ==  zero then real = '0' end
     --print(string.format("(%s,%s)",real,y))
     context(string.format("draw (%s,%s);",
       real,y))
  end
 end
end
context("\\stopMPpage")
\stopluacode
```

With a precision of 12 digits and a square region of [−5,5] we have then :



Implicitization of a cubic bezier curve

The next and last example will show how to find, given $\mathbf{p}$,$\mathbf{c}_1$,$\mathbf{c}_2$,$\mathbf{q}$ the points of a cubic Bezier curve in parametric form ($\mathbf{p}$ start point, $\mathbf{c1}$ and $\mathbf{c2}$ control points and $\mathbf{q}$ end point), a polynomial P[X,Y] that is the implicit form of the curve. Given the parametric form of a cubic Bezier $\mathcal{C} \in \mathbf{Q}$

$$\mathcal{C} = \Big\{ (1-t)^3\mathbf{p} + 3(1-t)^2t\mathbf{c_1} + 3(1-t)t^2\mathbf{c_2} + t^3\mathbf{q}, \\ t \in [0,1] \Big\}$$

for a point $(x_t,y_t) \in \mathcal{C}$ we have

$$x_t = a_3t^3 + a_2t^2 + a_1t + a_0 = a(t)$$
$$y_t = b_3t^3 + b_2t^2 + b_1t + b_0 = b(t)$$

Following Sederberg([10], chap. "Algebraic Geometry for CAGD"), let

$$f = f(t,x) = a(t) - x$$
$$g = g(t,y) = b(t) - y$$

and

$$h_1(t,x,y) = (a_3g - b_3f)$$
$$h_2(t,x,y) = (a_3t + a_2)g - (b_3t + b_2)f$$
$$h_3(t,x,y) = (a_3t^2 + a_2t + a_1)g - (b_3t^2 + b_2t + b_1)f$$

In PARI/GP every indeterminate has an order and the first indeterminate is x, so it's better rename $(t,x,y) \rightarrow$ (x,X,Y) so that each h_j can be seen as a polynomial $(h_j[X,Y])[x]$ with at most degree 2 with respect to x. If we are able to find $h_1[x] = h_2[x] = h_3[x] = 0$ (the *null polynomial* of $\mathbf{Q}[\mathbf{x}]$) then we have found the implicit form of our curve. It can be demonstrated that, if $h_{j,n}$ is the coefficient of x^n of h_j,

$$\begin{pmatrix} h_{1,2}[X,Y] & h_{1,1}[X,Y] & h_{1,0}[X,Y] \\ h_{2,2}[X,Y] & h_{2,1}[X,Y] & h_{2,0}[X,Y] \\ h_{3,2}[X,Y] & h_{3,1}[X,Y] & h_{3,0}[X,Y] \end{pmatrix} \begin{pmatrix} x^2 \\ x \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

if and only if

$$\begin{vmatrix} h_{1,2}[X,Y] & h_{1,1}[X,Y] & h_{1,0}[X,Y] \\ h_{2,2}[X,Y] & h_{2,1}[X,Y] & h_{2,0}[X,Y] \\ h_{3,2}[X,Y] & h_{3,1}[X,Y] & h_{3,0}[X,Y] \end{vmatrix} = 0$$

and hence this determinant is our P[X,Y].

The code is quite long, but not complicated:

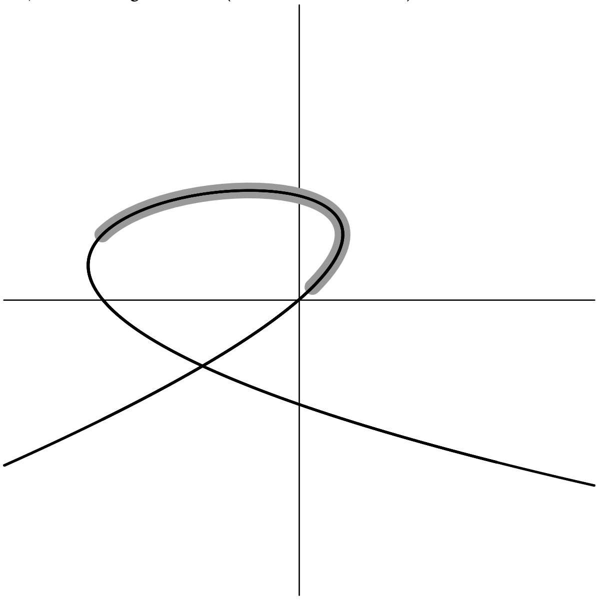
```
local function bezier_impl(p,c1,c2,q)
  local avma = pari.avma
  local f = string.format(
    "(1-t)^3*%s+3*(1-t)^2*t*%s+3*(1-t)*t^2*%s+t^3*%s",
    p[1], c1[1], c2[1], q[1])
  local g = string.format(
     "(1-t)^3*%s+3*(1-t)^2*t*%s+3*(1-t)*t^2*%s+t^3*%s",
     p[2], c1[2], c2[2], q[2])
  local fx =
   pari.gp_read_str(string.format("X-Pol(%s,x)", f))
  local gx =
   pari.gp_read_str(string.format("Y-Pol(%s,x)", g))
  fx =  pari.GENtostr(fx)
  gx =  pari.GENtostr(gx)
  local coeff_f_str =
```

```
 string.format("A=Vec(%s);B=if(poldegree(%s)==3,
  A,if(poldegree(%s)==2,[0,A[1],A[2],A[3]],
  if(poldegree(%s)==1,[0,0,A[1],A[2]],
  if(poldegree(%s)==0,[0,0,0,A[1]],[0,0,0,0]))));B",
  fx,fx,fx,fx,fx)
local coeff_g_str =
 string.format("A=Vec(%s);B=if(poldegree(%s)==3,
 A,if(poldegree(%s)==2,[0,A[1],A[2],A[3]],
 if(poldegree(%s)==1,[0,0,A[1],A[2]],
 if(poldegree(%s)==0,[0,0,0,A[1]],[0,0,0,0]))));B",
 gx,gx,gx,gx,gx)
local coeff_f = pari.gp_read_str(coeff_f_str)
local coeff_g = pari.gp_read_str(coeff_g_str)
local a3,a2,a1 =
 pari.uti_gel(coeff_f,1), pari.uti_gel(coeff_f,2),
 pari.uti_gel(coeff_f,3)
local b3,b2,b1 =
 pari.uti_gel(coeff_g,1), pari.uti_gel(coeff_g,2),
 pari.uti_gel(coeff_g,3)
local h1 =
  pari.gp_read_str(string.format("%s*(%s)-%s*(%s)",
   pari.GENtostr(a3), gx, pari.GENtostr(b3),fx))
local h2 =
  pari.gp_read_str(
   string.format("(%s*x+%s)*(%s)-(%s*x+%s)*(%s)",
   pari.GENtostr(a3), pari.GENtostr(a2), gx,
   pari.GENtostr(b3), pari.GENtostr(b2), fx))
local h3 =
  pari.gp_read_str(string.format(
   "(%s*x^2+%s*x+%s)*(%s)-(%s*x^2+%s*x+%s)*(%s)",
   pari.GENtostr(a3), pari.GENtostr(a2),
   pari.GENtostr(a1),gx, pari.GENtostr(b3),
   pari.GENtostr(b2), pari.GENtostr(b1),fx))
local h1_v = pari.gtovec(h1)
local h2_v = pari.gtovec(h2)
local h3_v = pari.gtovec(h3)
local idmat= pari.gp_read_str("idmat=matid(3)")
pari.gp_read_str(string.format("idmat[1,]=%s",
  pari.GENtostr(h1_v)))
pari.gp_read_str(string.format("idmat[2,]=%s",
  pari.GENtostr(h2_v)))
pari.gp_read_str(string.format("idmat[3,]=%s",
  pari.GENtostr(h3_v)))
idmat = pari.gp_read_str("idmat")
idmat_det = pari.gp_read_str("matdet(idmat)")
local PXY = pari.GENtostr(idmat_det)
local PxY =
 pari.gp_read_str(string.format("subst(%s,X,x)",PXY))
local Pxy =
 pari.gp_read_str(string.format("subst(%s,Y,y)",
  pari.GENtostr(PxY)))
local res = pari.GENtostr(Pxy)
local resTeX = pari.GENtoTeXstr(Pxy)
pari.avma = avma
return res,resTeX
end
```

For a curve $\mathcal{C}$ with $\mathbf{p} = (1,1)$, $\mathbf{c_1} = (10,10)$, $\mathbf{c_2} = (-10,10)$, $\mathbf{q} = (-15,5)$ we have

$$\begin{aligned} P[X,Y] = & -64X^3 + (2112Y + 312360)X^2 + \\ & (-23232Y^2 - 67920Y + 4711200)X + \\ & (85184Y^3 - 4440Y^2 - 5383200Y + 368000) \end{aligned}$$

It's easy to plot $\mathcal{C}$ with MetaPost (it's just `draw (1,1) .. controls(10,10) and (-10,10) .. (-15,5)`) so the next picture shows the MetaPost curve (thick, color gray) and the roots of $P[X,Y]$ for $-15 <= x <= 15, -15 <= y <= 15$ (thin, color black).



Conclusion

One of the main benefits of ConTEXt MKiV is the clear separation between Lua code and TEX code, and in this case it's a good thing that we can import a pari-lua script into ConTEXt MKiV without too much work to adapt it to the ConTEXt MKiV machinery — i.e. we have an high degree of code reuse. PARI/GP has also a nice TEX formatter, even if in some situations things are a bit raw. On the other side, solving numerical problems *always* requires some amount of theoretical analysis *before* doing the computation, as in the case of `solve` — in some circumstances PARI/GP abruptly aborts if it finds an error. Some computations can require a long time to finish, and given that ConTEXt MKiV is a multipass system a caching mechanism should be provided to solve these situations. Numeric results *can* (but they shouldn't) depend on the compiler and/or platform, but from this point of view it seems that PARI/GP is platform-independent.

Bibliography

[1] http://pari.math.u-bordeaux.fr.
[2] http://sagemath.org.
[3] http://www.nongnu.org/gsl-shell.
[4] http://www.inf.puc-rio.br/~roberto/pil2.
[5] http://swig.org.
[6] http://www.mingw.org.
[7] http://www.lilypond.org/gub.
[8] http://www.luatex.org/svn/trunk/manual/luatexref-t.pdf.
[9] http://en.wikipedia.org/wiki/Leibniz_formula_for_pi.
[10] tom.cs.byu.edu/~557/text/cagd.pdf

Luigi Scarso