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_mael2 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/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_gmael(GEN m, long x1, long x2)
{return gmael2(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_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
-1./pari -I/opt/swig-2.0.2/include
-c pari_wrap.c -o pari_wrap.o
gcc -Wall -ansi -shared -I./pari
-1/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 simplic-
ity. 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 cal-
culate exactly a summation. The GP function is
sum(X,a,b,expr,start) that stands for \[ \sum_{X=a}^{b} expr(X,\cdot) \],
where start is the initial value of expr(X,\cdot):
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 \text{T\LaTeX} 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 \text{libpari}, because most of GP functions manage the stack correctly.

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

\[
\sum_{k=0}^{30} \frac{4(-1)^k}{2k + 1} = 3.173842337190749408690224140
\]

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 \text{default(\ldots)} function can be used to get/set some internal constants and \text{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))
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 \text{sumalt(X=a,expr)} function that does the job:

\[
\sum_{k=0}^{30} \frac{4(-1)^k}{2k + 1} = 3.173842337190749408690224140
\]

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

\[
\sum_{k=0}^{30} \frac{4(-1)^k}{2k + 1} = 3.173842337190749408690224140\\
5863013579100197316985228418472920064106597929865025
\]

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 \text{default(\ldots)} function can be used to get/set some internal constants and \text{realprecision} is what we need:

\[
\sum_{k=0}^{30} \frac{4(-1)^k}{2k + 1} = 3.173842337190749408690224140
\]

But we can do things a bit better. First, \text{we want to control} the precision of the result, i.e. how many digits to show. This is quite simple: the GP \text{default(\ldots)} function can be used to get/set some internal constants and \text{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))
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 \text{sumalt(X=a,expr)} function that does the job:

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))
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 \text{sumalt(X=a,expr)} function that does the job:

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))
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 \text{sumalt(X=a,expr)} function that does the job:

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))
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 \text{sumalt(X=a,expr)} function that does the job:

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))
pari.avma = avma
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} = \pi/4
\stopformula

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.

\begin{verbatim}
3.141592653589793238462643383279502884197
1693993751058209749445923078164062862089
98628034825342117067982148086513282300664
7093844609550582231725359408128481117450
284102701938521105596466229489549303819
644288109756659334461284756823378673816
52712019091546586692346034861045432648
2133936072602491412732745870066063155881
7488152092096282925409171536436789259036
0011330530548820466523841469511941511609
433057270365759519350921861173819326117
93105118548074423799627495673518857272
489122793818301194129833673362440656643
0860213949463952247371907021798609437027
7053921717629137675238467481846766940513
2005856217214526356082787571342757789609
1736371787214684409012249534301465495583
71050792297698259235420199561112902196
0864034418159813629774771309960518707211
349999983729780499510597317328160963186
\end{verbatim}

\textbf{Figure 1.} Evaluation of an alternating series with 800 digit precision.

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

\begin{verbatim}
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
\end{verbatim}

\begin{verbatim}
\ctxlua{context(document.lscarso.taylor("atan(x)","x"))}
\end{verbatim}

\[ i.e. \quad \arctan(x) = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \frac{x^9}{9} - \frac{x^{11}}{11} + \frac{x^{13}}{13} - \frac{x^{15}}{15} + O(x^{16}) \]

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

\[ 4 \sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1} = 4\pi/4 = \pi. \]

It's important to note that theoretically this series has a slow convergence to \(\pi\) (it's hence a bad choice to calculate \(\pi\)) but \textit{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:

\begin{verbatim}
\ctxlua{context(document.lscarso.sum("k",0,3,"1/(x^2+k)","0"))}
\end{verbatim}

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.

\textbf{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 \textit{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 \( \frac{p_n}{q_n} \). For example \( [0,3] = \frac{1}{3} \) and it’s possible to show that \( \frac{p_n}{q_n} \) is the fraction in lowest terms. The \( \text{GF contfrac} \) function calculates (the vector of) the continued fraction of a rational number, while \( \text{contfrac}(p_n/q_n) \) 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 \text{libpari bestappr}(x,A) \) function calculates exactly what we need:

```lUA
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 \text{pari.uti_gel}, the \textit{wrapped} version of \text{libpari gel} function, to access an array by components.

Instead of an arbitrary real number, we choose \( \pi \) because the \text{libpari mppi(long)} function gives \( \pi \) with the required precision.

```lUA
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
```

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 \( \text{solve}(x=a,b,\text{expr}) \) GP function implements a very good algorithm but it works with one variable only and it fails if \( \text{expr} \) is not defined in \([a,b]\) and it hasn’t a \textit{variation} in \([a,b]\). This Lua wrapper \text{solve} tries to ensure that at in \([a,b]\) there is a variation evaluating the sign of \( \text{expr}(a) \ast \text{expr}(b) \):

```lUA
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)
end
```

```lUA
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:

<table>
<thead>
<tr>
<th>fraction</th>
<th>approx. value</th>
</tr>
</thead>
<tbody>
<tr>
<td>333/106</td>
<td>3.1415094339623</td>
</tr>
<tr>
<td>104348/33215</td>
<td>3.1415926539214</td>
</tr>
<tr>
<td>16/5</td>
<td>3.2</td>
</tr>
<tr>
<td>13/4</td>
<td>3.25</td>
</tr>
<tr>
<td>22/7</td>
<td>3.1428571428571</td>
</tr>
<tr>
<td>355/113</td>
<td>3.141592920354</td>
</tr>
<tr>
<td>19/6</td>
<td>3.1666666666667</td>
</tr>
<tr>
<td>103993/33102</td>
<td>3.1415926530119</td>
</tr>
</tbody>
</table>

where the approx. values are due to the Lua floating point math.
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:
\[ x^5 + \arctan(x) \cdot x^3 + 2 \cdot x^2 + x + 1 \approx 0 \]
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 \( P[X] \):
\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);","%s",prec))
context(string.format("draw (-%s,0)--(%s,0);","%s",prec))
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 res = pari.gp_read_str(
        string.format("eval(%s)\",expr))
    res = pari.GENtostr(res)
    pari.avma = avma
    return res
end
\stopluacode

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);","%s",prec))
context(string.format("draw (-%s,0)--(%s,0);","%s",prec))
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,"image=",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

context("\stopMPpage")
\stopluacode

With a precision of 12 digits and a square region of \([-5,5]\) we have then:

\begin{equation}
x_t = a_3 t^3 + a_2 t^2 + a_1 t + a_0 = a(t)
y_t = b_3 t^3 + b_2 t^2 + b_1 t + b_0 = b(t)
\end{equation}

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

\begin{align}
f &= f(t, x) = a(t) - x \\
g &= g(t, y) = b(t) - y
\end{align}

and

\begin{align}
h_1(t, x, y) &= (a_3 g - b_3 f) \\
h_2(t, x, y) &= (a_3 t + a_2) g - (b_3 t + b_2) f \\
h_3(t, x, y) &= (a_3 t^2 + a_2 t + a_1) g - (b_3 t^2 + b_2 t + b_1) f
\end{align}

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 \(Q[x]\)) 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

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

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))
  local fx = pari.GENtostr(fx)
  local gx = pari.GENtostr(gx)
  local coeff_f_str =
For a curve $\mathcal{C}$ with $p = (1,1)$, $c_1 = (10,10)$, $c_2 = (-10,10)$, $q = (-15,5)$ we have

$$P[X,Y] = -64X^3 + (2112Y + 312360)X^2 +$$
$$(-23232Y^2 - 67920Y + 4711200)X +$$
$$(85184Y^3 - 4440Y^2 - 5383200Y + 368000)$$

It's easy to plot $\mathcal{C}$ with MetaPost (it's just \texttt{draw (1,1) \ldots controls(10,10) \ldots (-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).

\begin{center}
\includegraphics[width=0.5\textwidth]{curve.png}
\end{center}

\section*{Conclusion}

One of the main benefits of Con\TeX\kern-.1667em\TeX\kern-.125em Xe\kern-.125em Xe 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 Con\TeX\kern-.1667em\TeX\kern-.125em Xe\kern-.125em Xe MKIV without too much work to adapt it to the Con\TeX\kern-.1667em\TeX\kern-.125em Xe\kern-.125em Xe 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 Con\TeX\kern-.1667em\TeX\kern-.125em Xe\kern-.125em Xe 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


Luigi Scarso