Extending ConTEXt MkIV with PARI/GP

Luigi Scarso

Abstract

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

Sommario

Questo articolo mostra come costruire un *binding* a PARI/GP, un noto sistema di computer algebra, per ConTEXt MkIV; mostra inoltre alcuni esempi di come risolvere alcuni problemi algebrici basilari.

1 Introduction

PARI/GPPARI 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 respectable capabilities on symbolic manipulations, it has also an extensive algebraic number theory module, hence it can perform, due to the highly optimised C library, complex numeric calculations very quickly and accurately. 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 also the C was dropped for unknown reasons. The current stable version is `2.3.4`.

ConTEXt MkIV is today the most advanced macro format that uses LuaTEX. It's still under active development: a milestone release is planned for spring 2012, when LuaTEX will reach the `1.0` version (currently is `0.70`). ConTEXt MkIV has an integrated support for Lua and MetaPost and offers a superb (as of today) management of OpenType fonts. Its philosophy is different from LATEX: ConTEXt MkIV is monolithic, while LATEX is more modular.

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 — hopefully it should be simple to adapt them for luaLATEX. We will see how to evaluate summations like $\sum_{k=0}^{30} \frac{4(-1)^k}{2k+1}$, calculating the exact value or an approximation; but also we will able to evaluate series like $\sum_{k=0}^{\infty} \frac{4(-1)^k}{2k+1}$ or symbolic sums as $\sum_{k=0}^{3} \frac{1}{x^2+k}$. PARI/GP is also able to solve equations like $x^5 + \arctan(x)x^3 + 2x^2 + x + 1 = 0$ and we will see how to use it together MetaPost to plot the roots of $P[X, Y] = X^3 - X - Y^2$. Finally, the last section is dedicated to show how to solve a simple problem of algebraic geometry using a few PARI/GP functions: the reader is encouraged to implement by himself the code.

2 Build the Lua binding

2.1 What is a 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 by itself is a complete high level programming language. It's a scripting language, with a syntax quite similar to Pascal, and it's used either as *embedded language* to enhance an application (e.g. to build plug-ins) or as *glue language* to "connect" together several object libraries — exactly the same as in SageSAGE, where the glue language is Python. In the latter case Lua is *extended* with the new libraries that become Lua modules: they can be built in into the `lua` interpreter at compile time (as in the GSL ShellGShell program) or loaded at runtime — which is the case of the extension shown in this paper.

Nowadays it's quite common that a scripting or dynamic language can load at runtime an object library, written in C or C++ and then compiled. Usually it's necessary to write some C code that act as an interface between the library and the language: this process is called "build the binding for the library". Here, the developer decides which symbols of the library (i.e. functions, classes, variables and constants) export and how they are seen from the language (under which name, for example). This is a delicate phase, because one must know the conventions of the language, its interface code (the "Application Program Interface" or API of the language), the API of the target library and write the appropriate C code for each symbol to export: for libraries written in C these APIs are usually organised in the *header files* (with suffix `".h"`) that contain the declarations of each function, variable or constant defined in the library — but not always all of them can be exported: the developer must also know the set of useful symbols to export the set of admissible.

It's clear that each language has its own peculiarities, but many high level languages are implemented in C of C++ and hence shared a common

substrate; also, an object library must rigorously follows the conventions of the host Operating System, hence its structure cannot depend to the high level language.

Here is where SWIG enters to play. SWIG (*Simplified Wrapper and Interface Generator*, seeSWIG) is a program that helps the developer to build bindings and, for some libraries, practically can build automatically a binding only by reading all the headers files. SWIG reads a driver file, the so-called *interface file* `".i"`, executes its instructions and ends with the binding: most of the time it's C code that must be compiled into an object module. The power of SWIG is that the user can select the target high level language so that the same interface file can be used for different languages (i.e. Lua, Python, Ruby and so on); on the other side, the code of the binding is more complicated than an binding manually built by a developer.

2.2 Build the Lua binding for PARI/GP

The Lua API are completely listed in the Lua bookPIL and they describe a simple and robust mechanism: basically every C function that wants to interact with the Lua interpreter uses a stack 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 function of the target library with the right arguments and use the stack to exchange the *in* (input to the target function) and/or *out* (output to the Lua interpreter) values.

The interface file `pari.i` for the binding of `libpari`, the object library of PARI/GP, is shown below and its instructions are quite simples: they basically say "read all 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 `liblua` 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 subfolder `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 straight: almost every symbol of `libpari` has a counterpart in Lua with the same name; the symbols "private" are stated 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 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 a Lua module with `require("pari")`.

As a final note for this section, the same steps can be followed under Windows using MinGWMINGW or with GUBGUB 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`.

3 Examples

3.1 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 in a one-to-one correspondence 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 faster than GP and it has a finer 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 $\sum_{X=a}^{b} expr(X,\cdot)$, where `start` is the initial value of $expr(X,\cdot)$:

```
\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

$$\sum_{k=0}^{30}\frac{4(-1)^k}{2k+1}=\frac{5863013579100197316985228 4}{1847292006410659792986502 5}$$

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

Next, we must avoid conflicts with others Lua functions. A common solution is to define a namespace (`document.lscarso` in this case), and a local function (`sum(X,a,b,expr,start)`) to be stated within 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. Fortunately, 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 $\mathbb{Q}$: if we want an approximated value we just use `start=0.` (`0.` means "zero" as floating point value) 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 define the 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.

Of course this is a well known series: from $\arctan(1) = \pi/4$ one can calculate the Taylor expansion of $\arctan(x)$ 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
```

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

3.14159265358979323846264338327950288419
71693993751058209749445923078164062862089
98628034825342117067982148086513282306647
09384460955058223172535940812848111745028
41027019385211055596446229489549303819
64428810975665933446128475648233786783165
27120190914564856692346034861045432664821
33936072602491412737245870066063155881
74881520920962829254091715364367892590360
01133053054882046652138414695194151160943
30572703657595919530921861173819326117931
05118548074462379962749567351885752724891
22793818301194912983367336244065664308602
13949463952247371907021798609437027705392
17176293176752384674818467669405132000568
12714526356082778577134275778960917363717
87214684409012249534301465495853710507922
79689258923542019956112129021960864034418
15981362977477130996051870721134999999837
29780499510597317328160963186

FIGURE 1: Evaluation of an alternating series with 800 digit precision.

```
   return res
end

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

i.e.

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

The series is convergent in $x = 1$ (there are several proofs about this, e.g. see Leibniz), and the result is exactly $\arctan(1)$, 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 *in practice* it can be used with PARI/GP to give quickly an high precision result — this is the power of the library.

Before going further, let's consider again 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.

3.2 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. $[a_0, a_1, a_2, \ldots, a_n]$ the rational number given by calculating all the intermediate fractions is usually termed as p_n/q_n, i.e.

$$[a_0, a_1, a_2, \ldots, a_n] = \frac{p_n}{q_n}$$

For example [0,3] = 1/3 and it's possible to show that $p_n/q_n = [a_0, a_1, a_2, \ldots, a_n]$ is the fraction in lowest terms. The GF `contfrac` function calculate (the vector of) the continued fraction of a rational number, while `contfracpnqn`, given a (finite vector of) continued fraction, return 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 approximate values are due to the Lua floating point math.

3.3 Equations

Solving numeric equations in PARI/GP require more attention than other packages. The GP `solve(X=a,b,expr)` functions 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, by 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 following 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 and last 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 indeterminate 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:

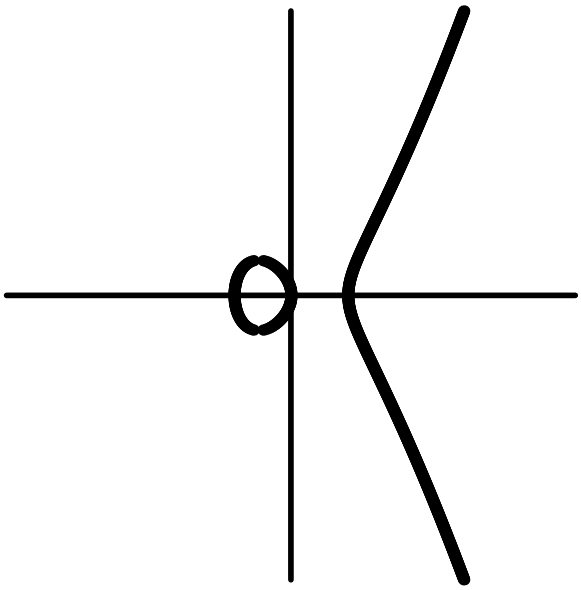
```
local function polroots(poly,prec)
   local avma = pari.avma
   pari.gp_read_str(string.format(
     "default(realprecision,%s)",prec))
   local poly = tostring(poly)
   local prec = tonumber(prec)
   local degree = pari.degree(
      pari.geval(pari.strtoGENstr(poly)))
   local roots = pari.roots(
    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
```

Finally, we need to iterate y over $[a,b]$ and find the roots of $P[X]$. Instead of producing a table, we plot the values with the help of a MetaPost `\startMPpage...\stopMPpage` environment:

```
\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 :



3.4 Solving a simple problem

In this last example we will show how to solve a simple problem of algebraic geometry, the approximation of a quarter of a circumference with a Bezier curve, with a mix of visual, numeric and symbolic techniques. The idea is to show the powerful of the extension *together* with the typographical capabilities of the TEX and MetaPost, rather than present some original ideas.

First, let's recall some basic definitions.
Given a field k (for us $k = \mathbb{Q}$ or $k = \mathbb{R}$), a *plain cubic Bezier curve* in parametric form is a subset $(x, y) \subset k \times k$ where

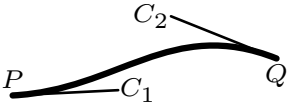
$$x = (1-t)^3 P_x + 3(1-t)^2 t C_{1x} + 3(1-t)t^2 C_{2x} + t^3 Q_x$$

$$y = (1-t)^3 P_y + 3(1-t)^2 t C_{1y} + 3(1-t)t^2 C_{2y} + t^3 Q_y$$

and $t \in [0,1] \subset k$. The points $\mathbf{P}$, $\mathbf{C_1}$, $\mathbf{C_2}$, $\mathbf{Q}$ are called *control points*; $\mathbf{P}$ is the *starting point* ($t = 0$), $\mathbf{Q}$ is the *ending point* ($t = 1$). MetaPost has a built in macro to draw cubic Bezier curves: for example this code

```
\starttext
\startMPpage
pair P,C[],Q;
P:=(0,0); C[1]:=(20,1);
C[2]:=(30,15); Q:=(50,7);
draw P..controls C[1] and C[2]..Q ;
\stopMPpage
\stoptext
```

draws:



A nice property is that the derivative of the curve at $\mathbf{P}$ and $\mathbf{Q}$ are $3(\mathbf{C_1} - \mathbf{P})$ and $3(\mathbf{Q} - \mathbf{C_2})$, as we can qualitatively see in the previous picture.

The next definition is the ring of polynomials in the two indeterminates $k[X, Y]$. Informally, it's the set of the finite expressions $P[X, Y]$ where $P[X, Y] = \sum_{i,j} a_{ij} X^i Y^j$ and $a_{ij} \in k$. The total degree of $P[X, Y]$ is $\max(i + j), a_{ij} \neq 0$ and if all a_{ij} are zero then we have the zero polynomial $\mathbf{0}$ (the zero of the ring $k[X, Y]$) which has degree undefined (but many authors set it to -1 or $-\infty$). Two polynomials $P_1[X, Y]$ and $P_2[X, Y]$ are identical if and only if $P_1[X, Y] - P_2[X, Y] = \mathbf{0}$; two polynomials with different degree are never identical.

Given $(x, y) \in k \times k$ and a polynomial $P[X, Y]$, with an abuse of notation we can consider the function $P(x, y) : k \times k \mapsto k$ where we "replace" X with x and Y with y in $P[X, Y]$ and then calculate the value ($P(x, y)$ it's an expression of k).

Now, a root of $P[X, Y]$ is a point $(x, y) \in k \times k$ for which $P(x, y) = 0$; given a $P[X, Y]$, the set of its roots is the *curve* of the polynomial and describe the properties of a curve is the main scope of the algebraic geometry (of course not only with two indeterminates and not only with $k = \mathbb{Q}$ or $k = \mathbb{R}$). In general it's a difficult task, but anyway we can already say something:

1. $P[X, Y] = X^2 + 1$ has no roots in $k \times k$ (remember that $k = \mathbb{Q}$ or $k = \mathbb{R}$);

2. $P[X, Y] = X$ has infinite roots because for every $y \in k$ $P(0, y) = 0$;

3. $P[X,Y] = \mathbf{0}$ (the zero polynomial) has infinite roots: for every point of $(x,y) \in k \times k$ $P(x,y) = 0$, so in this case the set of roots is $k \times k$.

Hence the curve of $P[X,Y]$ can be empty, infinite but properly included in $k \times k$, or all $k \times k$. Of great interest are also the set of intersections of two curves, which can be empty (i.e. $P_1[X,Y] = Y - 2$ and $P_2[X,Y] = Y$), finite ($P_1[X,Y] = X - Y$ and $P_2[X,Y] = X + Y$) and infinite ($P_1[X,Y] = \mathbf{0}$ and $P_2[X,Y] = Y$)

Let's consider $P[X,Y] = X^2+Y^2-1$. The "brute force" approach to have an idea of its curve $\mathcal{C}$ is, given a tolerance ϵ and a limit R, verify whenever $\left|x^2 + y^2 - 1\right| < \epsilon$ for $(x,y) \in [-R,R] \times [-R,R]$; a bit less primitive way is to solve $x_0^2 + y^2 - 1 = 0$ for y, with a fixed tolerance ϵ and the parameter $x_0 \in [-R,R]$. Both methods was shown in the previous sections, so here we take another way. Let's consider the set $P_t[X,Y] = Y - tX$ indexed by t; it's trivial to show that its curves are the lines $y = tx$. The *intersection* between $P(x,y) = 0$ and $P_t(x,y) = 0$ is also easy to calculate: just replace y^2 with $(tx)^2$. We have hence $x^2 + (tx)^2 - 1 = 0$ and

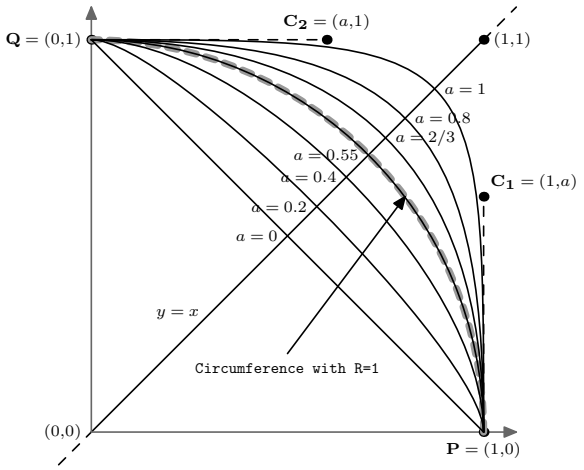
$$\mathcal{C} = \begin{cases} x^2 = & \dfrac{1}{1+t^2} \\ \\ y^2 = & \dfrac{t^2}{1+t^2} \end{cases}$$

or, after few simplifications,

$$\mathcal{C} = \begin{cases} x = & \pm\dfrac{1}{\sqrt{1+t^2}} \\ \\ y = & \pm\dfrac{t}{\sqrt{1+t^2}} \end{cases}$$

where t runs over k. This is called a *parametric form* of the curve $\mathcal{C}$, while $x^2+y^2-1=0$ is called the *implicit form* of the curve (the curve defined implicitly by X^2+Y^2-1). Although it was easy to obtain, it's not a nice formula, due to the presence of the square root factor; maybe we cannot avoid the fractions, but we would like to have integer exponents. In fact, in $k = \mathbb{Q}$, $\sqrt{1+t^2}$ not always is rational: for example for $t = 1/3$ we have the $\sqrt{10}$ factor which is not rational.

We can now come back to our problem, and make some assumptions. First, we will consider a quarter of a circumference $\mathcal{C}$ with $x \geq 0$ and $y \geq 0$ and radius 1. Next, we will assume that MetaPost is able to draw a circumference, that we will use as a reference. In this case, the best candidates seem to be the cubics with control points $\{\mathbf{P}, \mathbf{C_1}, \mathbf{C_2}, \mathbf{Q}\} = \{(1,0), (1,a), (a,1), (0,1)\}$ where $a \in [0,1]$ and hence we can draw these cubics for some values of a, just to see how they look like.



After few tries we can discover that a good approximation can be reach with $a = 0.55$, so we have some hints for a solution; but to better understand the problem we need to find the implicit form of a Bezier curve, at least for those ones that we are studying. Let's rewrite the parametric form:

$$\mathcal{C}_a = \begin{cases} x = & (1-t)^3 + 3(1-t)^2 t + 3(1-t)t^2 a \\ \\ y = & 3(1-t)^2 ta + 3(1-t)t^2 + t^3 \end{cases}$$

First let's note that for $t = 1/2$ we have

```
subst([(1-t)^3+3*(1-t)^2*t+3*(1-t)*t^2*a,
      3*(1-t)^2*t*a+3*(1-t)*t^2+t^3],t,1/2)

 = [3/8*a + 1/2, 3/8*a + 1/2]
```

i.e. the middle points of the curves are along the line $y = x$.

Next let's introduce the *hybrid* form

$$\begin{cases} -X + (1-t)^3 + 3(1-t)^2 t + 3(1-t)t^2 a \\ \\ -Y + 3(1-t)^2 ta + 3(1-t)t^2 + t^3 \end{cases}$$

We would like, if possible, to *eliminate* the parameter t from the hybrid form and obtain a polynomial $P_a[X,Y]$ indexed by a. The PARI-GP function `polresultant` is what we need:

```
PaXY=polresultant(
  -X + (1-t)^3+3*(1-t)^2*t+3*(1-t)*t^2*a,
  -Y + 3*(1-t)^2*t*a+3*(1-t)*t^2+t^3,t);
```

give us
$P_a[X,Y] = (27a^3 - 54a^2 + 36a - 8)X^3 + ((81Y - 108)a^3 + (-162Y + 207)a^2 + (108Y - 126)a + (-24Y + 24))X^2 + ((-81Y + 81)a^4 + (81Y^2 - 162Y + 81)a^3 + (-162Y^2 + 414Y - 252)a^2 + (108Y^2 - 252Y + 144)a + (-24Y^2 + 48Y - 24))X + ((81Y - 81)a^4 + (27Y^3 - 108Y^2 + 81Y)a^3 + (-54Y^3 + 207Y^2 - 252Y + 99)a^2 + (36Y^3 - 126Y^2 + 144Y - 54)a + (-8Y^3 + 24Y^2 - 24Y + 8))$

Now, if $P_a[X,Y]$ has degree 3 then it cannot be identical to $X^2 + Y^2 - 1$, and each approximation

will never be perfect (always remember that $k = \mathbb{Q}$ or $k = \mathbb{R}$ and that they have infinite elements). If we calculate the roots of the coefficient of X^3 with

```
factor(27*a^3 - 54*a^2 + 36*a - 8)
```

we have

```
[3*a - 2 3]
```

i.e $a_0 = 2/3$ is a root of order 3. So, maybe for $a_0 = 2/3$ we have a chance that $P_{a_0}[X, Y]$ has degree 2. Let's substitute a with 2/3 in $P_a[X, Y]$:

```
subst(PaXY,a,2/3)
```

The result is:

```
0.
```

This suggests that the previous parametrization is not valid if $a = 2/3$: in fact if we calculate `polresultant` with $a = 2/3$ we have

```
P23XY=polresultant(
   -X + (1-t)^3+3*(1-t)^2*t+3*(1-t)*t^2*(2/3),
   -Y + 3*(1-t)^2*t*(2/3)+3*(1-t)*t^2+t^3,t)
= X^2 + (-2*Y + 2)*X + (Y^2 + 2*Y - 3)
```

We should be not surprised: after all the curve with $a = 2/3$ is showed in the picture so we already know that *it is* a curve; only it has *order 2*:

```
[(1-t)^3+3*(1-t)^2*t+3*(1-t)*t^2*(2/3),
  3*(1-t)^2*t*(2/3)+3*(1-t)*t^2+t^3]

= [-t^2 + 1, -t^2 + 2*t]
```

i.e. $x = 1 - t^2$, $y = 2t - t^2$, $t \in$ [0,1] (actually, the picture shows that a cubic Bezier curve can be a segment ($a = 0$), a curve of second order ($a = 2/3$) or a curve of third order). But anyway again $X^2+(-2*Y+2)*X+(Y^2+2*Y-3)-X^2-Y^2+1$ is not the zero polynomial.

The conclusion then seems to be: $P_a[X, Y], a \in$ [0,1] *can never be equal to* $X^2 + Y^2 - 1$, or *with a Bezier curve we can never draw a circumference*. But we must consider that it depends from the result of the parametrization, and we have not proved that this is unique — or that one from `polresultant` is exact.

But let's go on. We have already seen that for $t = 1/2$ each curve has $x = y$; another idea from the picture comes if we consider the intersection of the circumference with the line $y = x$, i.e. $(x, y) = (1/\sqrt{2},1/\sqrt{2})$, and *impose* that our Bezier curves pass through $(1/\sqrt{2},1/\sqrt{2})$. This is because the curves in $\mathbf{P}$ and $\mathbf{Q}$ are already a good approximation of the circumference (they have similar derivatives) and they evolve smoothly, so if we choose the middle point as another constrain things *maybe* are better. Theoretically we should go from $P_a[X, Y]$ to $P_a(x, y) = 0$, substitute y with x, substitute x with $1/\sqrt{2}$ and calculate the roots of $P_a(1/\sqrt{2},1/\sqrt{2}) = 0$, but we can collect all these steps with

```
factor(subst(subst(PaXY,Y,X),X,1/sqrt(2)))
```

that gives

```
[a + 0.47140452079104846163831395 38 1]
[a + 0.47140452079101496042294785297 1]
[a - 0.55228474983079339840225163 09 1]
[a - 0.66666666666666666666666666685 1]
```

and we find that, for $a \in$ [0,1] and $a \neq 2/3$,

$a_0^* = 0.55228474983079339840225163 09$

is our only solution.

Now, for $t = 1/2$ the curves P_{a0} must pass in $x = y = 1/\sqrt{2}$, hence, from the parametric form,

$$\frac{3}{8}a + \frac{1}{2} = \frac{1}{\sqrt{2}}$$

which have a unique solution a_0

$$a_0 = 4\frac{\sqrt{2} - 1}{3}$$

It has a nice look (sometimes it is called the "1-2-3-4" formula): its value is

```
4*(sqrt(2)-1)/3
= 0.55228474983079339840225163 23
```

and it is a root of $P_a(x, y)$:

```
subst(subst(subst(PaXY,Y,X),X,1/sqrt(2)),
      a,4*(sqrt(2)-1)/3)
= 1.186446801E-27
```

Hence, whether we follow the implicit way or the parametric way the conclusion is the same: there is a unique Bezier curve that is an optimal approximation of a circumference. The parametric way give us a nice formula for the curve, the implicit way tell us that there are no Bezier curves that are circumferences.

Now, the last step. We would like to have an estimation of the error, i.e. how much our optimal cubic differs from the circumference. We can consider the distance between two points $\mathbf{P_1} = \{y = tx \cap \mathcal{C}\}$ and $\mathbf{P_2} = \{y = tx \cap P_{a_0}(x, y)\}$ and take the max of this distance as an indicator of the error. Recalling that $\mathcal{C}$ has radius 1, it's easy to show that $\vec{\mathbf{P_1}\mathbf{P_2}} = \vec{\mathbf{P_2}} - \vec{\mathbf{P_1}} = \vec{\mathbf{P_2}} - 1$ and the relative percentage error is

$$e_r = \frac{\vec{\mathbf{P_1}\mathbf{P_2}}}{\vec{\mathbf{P_1}}} 100 = 100\left(\vec{\mathbf{P_2}} - 1\right) \qquad (1)$$

The following code print the error e_r vs. the angle ϕ, where $y = x\tan(\phi)$, $0° \leq \phi < 90°$, and the $\max(e_r)$:

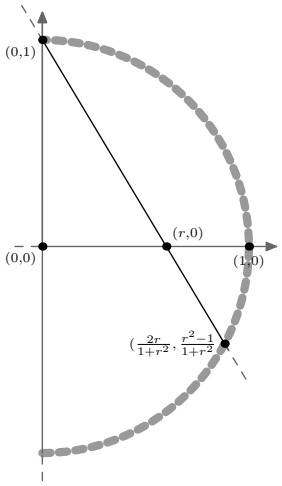
```
B=subst(subst(PaXY,a,
    4*(sqrt(2)-1)/3,),Y,m*X);
MAX=-100;
forstep(f=0,89,1,mf=tan(Pi/180*f);
  roots=polroots(subst(B,m,mf));
  forstep(i=1,3,1,
   xf=real(roots[i]);
  iif=imag(roots[i]);
  if(xf<=1.0 && xf>=0.0 && iif <1e-12,
    yf=mf*xf;  Pf=sqrt(xf^2+yf^2);
    if(MAX<(Pf-1)*100,MAX=(Pf-1)*100,MAX;);
    print(f,",",  (Pf-1)*100,",",MAX))))
```

The error is $e_r = 0.0272\%$, and confirms that a_0 is a good choice.

3.4.1 Some notes

A different parametrization of the circumference is given by the intersection of the line that join (0,1) and (*r*,0), $y = -\frac{1}{r}x + 1$, with $x^2 + y^2 - 1 = 0$:

$$\mathcal{C} = \begin{cases} x = & \dfrac{2r}{1+r^2} \\ \\ y = & \dfrac{r^2-1}{1+r^2} \end{cases}$$



Both x and y are rational if r is rational, so this shows that there are *infinite rational points on a circumference*.

Another question is when $\sqrt{1+t^2}$ is rational if t is rational. Let $t = t_1/t_2, t_1 \geq 0, t_2 \geq 0$: then

$$\sqrt{1+\left(\frac{t_1}{t_2}\right)^2} = \frac{n_1}{n_2} \iff \sqrt{t_1^2+t_2^2} = t_2\frac{n_1}{n_2}$$

i.e. the problem is to find t_1 and t_2 so that $\sqrt{t_1^2+t_2^2} = t_3$ is an integer, or $t_1^2 + t_2^2 = t_3^2$ with t_3 integer. There is an easy "3-4-5" formula to start with: $3^2 + 4^2 = 5^2$. If we rewrite it as $4^2 = 5^2 - 3^2$ or $4 \cdot 4 = (5-3)(5+3) = 2 * 8$, we see that 4 divides $(5+3)$, so $t_1 t_1 = (t_3 - t_2)(t_3 + t_2)$ and t_1 divides $(t_3 + t_2)$ is true at least for the triple $(t_1, t_2, t_3) = (3,4,5)$. Let's then *suppose* that t_1 divides (t_3+t_2), i.e. $t_3+t_2 = k_1 t_1$: with this condition we have

$$\begin{cases} t_1 = & (t_3 - t_2)k_1 \\ \\ k_1 t_1 = & (t_3 + t_2) \end{cases}$$

and hence

$$t_3 = \frac{t_1}{2}\frac{k_1^2+1}{k_1}$$

that is an integer if $t_1 = l_1 k_1$ and $l_1(k_1^2+1)$ is even, with l_1 integer.

The last note is about the field k. We have used $k = \mathbb{Q}$ or $k = \mathbb{R}$ were the distinction between $P[X,Y]$ and $P(x,y)$ seem to be artificial, but there are fields k where distinction matter. For example the classes of residues modulo p with p prime is a field $\mathbb{F}_p$ with the usual operation "$+, \times$" as in $\mathbb{Q}$ or $\mathbb{R}$. Let's consider $\mathbb{F}_5$: denote each class of residue with $\{\overline{0}, \overline{1}, \overline{2}, \overline{3}, \overline{4}\}$ (hence $\overline{4}\cdot\overline{3} = \overline{12} = \overline{2}$ and $\overline{4} + \overline{3} = \overline{7} = \overline{2}$).
The polynomial $P[X] = X(X-\overline{1})(X-\overline{2})(X-\overline{3})(X-\overline{4}) = X^5 + \overline{4}X$ it's not the zero polynomial of $\mathbb{F}_5[X]$, but $P(x) = \overline{0}$ for every element in $\mathbb{F}_5$.

4 Conclusion

One of the main advantages of ConTEXt MkIV is the clear separation between Lua code and TEX code, and in this case it's a very useful 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 crude.

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 multi pass 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 it seems that from this point of view that PARI/GP is really platform-independent.

References

URL `http://pari.math.u-bordeaux.fr`.

URL `http://sagemath.org`.

URL `http://www.nongnu.org/gsl-shell`.

URL `http://www.inf.puc-rio.br/~roberto/pil2`.

URL `http://swig.org`.

URL `http://www.mingw.org`.

URL `http://www.lilypond.org/gub`.

URL `http://www.luatex.org/svn/trunk/manual/luatexref-t.pdf`.

URL `http://en.wikipedia.org/wiki/Leibniz_formula_for_pi`.

▷ Luigi Scarso
`luigi.scarso at gmail dot com`