Animating Fourier series decomposition of a character with LuaTEX and MPLIB

Maxime Chupin

Abstract

In this article, we will see how, thanks to METAPOST and MPLIB and LuaTEX, we can build an animation illustrating in a mechanical way the Fourier decomposition of a closed contour.

This is a translation by the author of the original article in French, published in La Lettre GUTenberg *number 41 [2] of the French TEX user group.*

1 Introduction

The video artist 3Blue1Brown,[1] mathematical popularizer on YouTube, has made a video illustrating the Fourier decomposition of a closed path by animations of gear mechanisms of circles put end to end. The result is magnificent and bewitching (see figure 1).

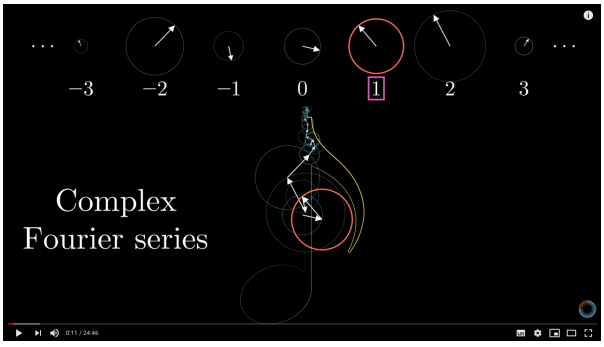


Figure 1: *But what is a Fourier series? From thermal transfer to drawings with circles* from 3Blue1Brown on YouTube.

It is not easy to describe these animations in words, but the idea is to put circles of different diameters with inscribed vectors that rotate at different speeds end to end, and the end of this broken line traces the closed curve (the music note in figure 1).

This viewing made me want to do this with our favorite tools, specifically with LuaLATEX and METAPOST, itself included in LuaTEX via the MPLIB library. I also thought it would be interesting to make these animations with the outline of a glyph of a character (of one part, i.e., connected). So I started working on this project.

2 Mathematical principle

We therefore consider a closed curve in $\mathbf{R}^2$ that can be considered as a periodic function $f : \mathbf{R} \mapsto \mathbf{C}$. The period is considered to be equal to 1. Without going into details, the Fourier series decomposition of f is:

$$\forall t \in [0,1], \quad f(t) = \sum_{n=-\infty}^{+\infty} c_n(f) \mathrm{e}^{\mathrm{i}n2\pi t},$$

where

$$c_n(f) = \int_{-1/2}^{1/2} f(t) \mathrm{e}^{-\mathrm{i}n2\pi t} \mathrm{d}t.$$

Numerically, we will work with discrete versions of this decomposition in Fourier series. Consider two integers N and M large enough and M even. In the discrete world, we will no longer have the continuous f function but samples along the path, which we will denote by $(f_1, f_2, \ldots, f_N)$ where the $f_i \in \mathbf{C}$. We then have:

$$\forall t \in [0,1], \quad f(t) \simeq \sum_{n=-M/2}^{M/2} \tilde{c}_n(f) \mathrm{e}^{\mathrm{i}n2\pi t}, \qquad (1)$$

where

$$\tilde{c}_n(f) = \sum_{k=0}^{N} \frac{1}{N} f_{k+1} \mathrm{e}^{-\mathrm{i}\frac{2nk}{N}}. \qquad (2)$$

The $M+1$ $\tilde{c}_n(f)$ will be called the Fourier coefficients.

Geometrically, we can see the relation (1) as a sum of vectors of $\mathbf{R}^2$ (thus put end to end) with norm the *modulus* of the complex number and, as orientation, its *argument*.

Thus, when t runs through the interval $[0,1]$, these vectors rotate and the end of the last vector draws the closed curve[2] that we have decomposed.

3 Get a set of points of $\mathbf{R}^2$ of the contour

We must therefore construct, from the contour of a glyph, the sequence $(f_1, f_2, \ldots, f_N)$ presented above.

3.1 Thanks to METAPOST

First, we need to obtain a discretization of the closed contour of a given glyph. METAPOST [5], with the MetaFun [3] format, allows us to do this quite easily. For the example, we will take the glyph f, 500 points for the discretization, and a certain homothetic factor set to 0.1 for the display.

The METAPOST code is the following:

```
fontmapfile "=lm-ec.map";
picture lettre; path contourLettre; path p;
lettre := glyph "f" of "ec-lmri10";
nbrPoints := 500; scale := 0.1;
beginfig(1);
for item within lettre:
 contourLettre := pathpart item;
 for i:=1 upto nbrPoints:
  if i=1:
   p := point i/nbrPoints along contourLettre;
```

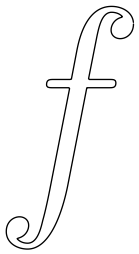
[1] youtube.com/watch?v=-qgreAUpPwM

[2] Or rather an approximation of the contour of the glyph.

doi.org/10.47397/tb/42-1/tb130chupin-fourier

```
  else:
   p := p--(point i/nbrPoints
            along contourLettre);
  fi;
 endfor;
 draw p scaled scale;
endfor; endfig; end;
```

The result is shown below, using the `luamplib` package [4] to use METAPOST directly in this article.



We will not detail this code here. It seems a bit complex but this is due to the METAPOST's glyph structure that (fortunately) allows having several parts for a glyph. Although here we will consider only letters with one (connected) part, we must adhere to the general data structure. The thing to remember is that we have METAPOST code which allows us to obtain a set of points constituting a discretization of the character glyph outline.

3.2 The list of points with Lua

The computation of the Fourier series decomposition of this closed curve is theoretically possible in TEX, but Lua [6] offers us more capabilities, more speed, and easier coding.[3] So, using LuaTEX, we want to retrieve this list of points on the Lua side. Another advantage is that, as we have already said, LuaTEX includes METAPOST via the Lua library, MPLIB (see [7]).

3.3 Search for font files

With METAPOST (or MPLIB), unfortunately we can use only Type 1 fonts. There is a little subtlety concerning the opening of the font file: METAPOST asks for a '`pfb`' file type, while `kpse` asks for a '`type1 fonts`' file type. Taco Hoekwater, on the `metapost@tug.org` mailing list, provided me with the search function that handles this little problem:

```
local mpkpse = kpse.new('luatex', 'mpost')
local function finder(name, mode, ftype)
if mode == "w" then
 return name
else
 if ftype == 'pfb' then
  ftype='type1 fonts'
 end
 return mpkpse:find_file(name,ftype)
end; end
```

[3] At least, for me ...

3.4 From METAPOST to a Lua table

Having passed this small technical difficulty, we will present here a Lua function that allows us to build a Lua table which contains the points generated by METAPOST.

```
function getpathfrommp(s,nbrPoints,scale)
-- define a Lua function which retrieves the list of
-- nbrPoints points made from the outline of the
-- character s; scale is a homothety parameter

-- launch MetaPost session using our search function
local mp = mplib.new({find_file = finder,})
-- metafun Format
mp:execute('input metafun ;')
-- we store the output of the MetaPost code execution
local rettable; rettable = mp:execute(
'fontmapfile "=lm-ec.map";
picture lettre; path contourLettre;
lettre := glyph "' .. s .. '" of "ec-lmri10";
path p; beginfig(1);
for item within lettre:
 contourLettre := pathpart item;
 for i:=1 upto'.. nbrPoints ..':
  if i=1: p := point i/'..nbrPoints..' along
    contourLettre;
  else: p:= p--(point i/'..nbrPoints..' along
    contourLettre);fi;
endfor;
draw p scaled '..scale..';
endfor; endfig; end;') -- MetaPost code as above
output = {} -- initialization
-- if the MetaPost code execution went well
if rettable.status == 0 then
 figures = rettable.fig -- figure list
 figure = figures[1] -- first and only figure
 local objects = figure:objects() -- object list
 -- compose figure from the first and only object:
 local segment = objects[1]
 for point =1, #segment.path do
  output[point] = {}
  output[point].x = segment.path[point].x_coord
  output[point].y = segment.path[point].y_coord
 end
end
else print("error") end;
return output
end
```

To roughly explain the above code, the purpose is to get the output of the execution of a code by METAPOST (MPLIB here). This output has a Lua structure (see the LuaTEX documentation [7], section `mplib`). So we browse this structure to extract

the list of points we are interested in: first of all the list of figures, which here is limited to a single one, then inside the first figure, we look for the `objects` which again are limited to a single `object` (our closed curve), then we browse the `path` (METAPOST) of the object, named here `segment`, and finally we retrieve the x and y coordinates that we store in our output variable `output`. For a description of the Lua functions allowing us to browse the structure of the object produced by the execution of the METAPOST code, please refer to the LuaTEX documentation.

We pass in three parameters: the character whose outline we want to trace (`s`), the number of points (`nbrPoints`), and the homothety (`scale`). Our code is not robust, because if the glyph corresponding to the character `s` is not connected, there is a strong chance that the code will not work.

4 Fourier series decomposition with Lua

4.1 Call to an external library

The Fourier series decomposition is done with complex numbers as presented previously. Complex numbers are not natively managed by Lua, but many libraries are available on the Web that implement computation functions on complex numbers. I chose the `complex.lua` file available at `lua-users.org/wiki/ComplexNumbers`.

To use this library, we need:

1. on the LaTeX side, to load the `luapackageloader` package (see the end of this article for the complete LaTeX code);
2. on the Lua side, to call the file `complex.lua` via the following code:

```
complex = require "complex"
```

4.2 Convert the list of coordinates of $\mathbf{R}^2$ into a list of complex numbers

To ease the computations, a function is created to convert the list of coordinates obtained by the Lua function `getpathfrommp` into a list of complex numbers. This is done by the following code, which needs no further explanation.

```
function pathToComplex(path)
local complexPath
complexPath = {}
for i=1,#path do
 complexPath[i]
   = complex.new(path[i].x,path[i].y)
end
return complexPath
end
```

4.3 Implementation of Fourier coefficients calculation

The computation of the Fourier coefficients $\tilde{c}_n(f)$ of equality (2) is easily implemented with Lua, as shown in the following code.

```
function cn(f,n)
local CN = complex.new(0.0,0.0)
local N = #f
for i=0,N-1 do
 exposant = complex.new(0.0,-2.0*math.pi*n*i/N)
 Exp = complex.exp(exposant)
 CN = complex.add(CN,complex.mulnum(complex.mul
    (f[i+1],Exp),1.0/N))
end
return CN; end
```

From this, we need to build the list

$$(c_{-M/2}, c_{-M/2+1}, \ldots, c_{-1}, c_0, c_1, \ldots, c_{M/2}),$$

which is done by the following Lua function:

```
function cnList(f)
local CNlist = {}
local M = #f
for i=0,M do
 CNlist[i] = cn(f,math.floor(i-M/2))
end
return CNlist; end
```

5 Plot with mplibcode

Once all these code bricks are prepared, we just have to implement the drawing with the help of the `mplibcode` environment of the `luamplib` package [4] (or we could use Ti*k*Z). This function has several arguments:

- a discretized `path`, i.e., the set of coordinates (x, y) of the contour of the glyph;
- its conversion into complexes (`complexPath`);
- a list of Fourier coefficients (`cnList`);
- a desired number of Fourier coefficients ($M + 1$ in the previous equations), i.e., the number of circles and vectors drawn (`nbrFourier`);
- a time $t \in [0, 1]$.

```
function coreDecomp(path, complexPath, cnList,
    nbrFourier, t)
-- path: list of R^2 points
-- complexPath: complex list of these points
-- cnList: list of Fourier coefficients
-- nbrFourier: number of Fourier coefficients
-- initialization
local str
local cnListRotated = {}
local zero = math.floor(#cnList/2)
local NFourier = math.floor(nbrFourier/2)
```

```
cnListRotated[zero] = cnList[zero]
-- multiplication by e^{2i k pi t}
for k=1,zero do
 cnListRotated[zero+k] = complex.mul(
    cnList[zero+k],complex.exp(complex.new(0.0,
                               k*2*math.pi*t)))
 cnListRotated[zero-k] = complex.mul(
    cnList[zero-k],complex.exp(complex.new(0.0,
                              -k*2*math.pi*t)))
end
-- beginning of mplibcode
local str = "\\begin{mplibcode}\nverbatimtex
  \\leavevmode etex; beginfig(1);"
-- MetaPost code of the glyph to draw
local mpCodeLetter = mpCodePath(path)
str = str..mpCodeLetter -- concatenation
-- complex current point at which
-- we draw the next circle
local currentC = complex.new(0,0)
-- add the drawing of the circle and the vector
str = str .. mpCodeCircle(cnListRotated[zero],
                          currentC)
currentC = complex.add(currentC,
                       cnListRotated[zero])
for i=1,NFourier do -- for all Fourier coeff
 str = str..mpCodeCircle(cnListRotated[zero+i],
                          currentC)
 currentC = complex.add(currentC,
                        cnListRotated[zero+i])
 str = str..mpCodeCircle(cnListRotated[zero-i],
                          currentC)
 currentC = complex.add(currentC,
                        cnListRotated[zero-i])
end
str = str.."endfig;\n\\end{mplibcode}\n"
      .."\\newpage" -- closing
return str; end
```

To help in reading this code: `cnListRotated[i]` corresponds to the terms in the sum (1) of $\tilde{c}_n(f)\mathrm{e}^{2\mathrm{i}k\pi t}$, since the multiplication by $\mathrm{e}^{2\mathrm{i}k\pi t}$ can be seen as a rotation in the complex plane.

The main purpose of this function is to construct a string containing the `mplibcode` that will be sent to LaTeX via the Lua `tex.sprint()` function. The `coreDecomp` function above calls two other Lua functions that produce the METAPOST code of the drawing:

- the function `mpCodePath(path)`, which takes as argument the list of the contour points of the glyph and draws the glyph;[4]
- the function `mpCodeCircle(cn,shift)`, which takes as argument a coefficient of Fourier `cn` (after rotation) and an $\mathbf{R}^2$ shift which is the end of the broken line where the vector and the circle must be drawn.

The code for these two functions is below. They mainly consist of the concatenation of strings to produce METAPOST code.

```
function mpCodePath(path)
-- plot the path contour
local str = ""
str = str.."path p; p:="
for i=1,#path do
 str = str.."(..string.format("%f",path[i].x)
       ..",..lstring.format("%f",path[i].y)
       ..")--"
end
str = str.."cycle; draw p;\n"
str = str.."pair ll,lr,ur,ul; ll:=llcorner p;"
      .."ur:=urcorner p; lr:=lrcorner p;"
      .."ul:=ulcorner p;\n"
str = str.."Wdth := abs(xpart lr - xpart ll);"
      .."Hght := abs(ypart ul- ypart ll);"
      .."prcW := 0.8; prcH := 0.3;\n"
str = str.."draw (ll+(-prcW*Wdth,-prcH*Hght))"
      .."--(lr+(+prcW*Wdth,-prcH*Hght))"
      .."--(ur+(+prcW*Wdth,+prcH*Hght))"
      .."--(ul+(-prcW*Wdth,+prcH*Hght))"
      .."--cycle;\n"
return str; end

function mpCodeCircle(cn,shift)
-- draw the circle and the vector corresponding to
-- the Fourier coefficient centered at points shift
local str
local abs,arg
abs,arg = complex.polar(cn)
str = "draw fullcircle scaled "
      ..string.format("%f",2*abs).."shifted ("
      ..string.format("%f",shift[1])..","
      ..string.format("% f",shift[2])
      ..") withcolor (0.7,0.7,0.7);\n"
str = str.."drawarrow ((0,0)--("
      ..string.format("%f",cn[1])..","
      ..string.format("%f",cn[2]).."))shifted("
      ..string.format("% f",shift[1])..","
      ..string.format("%f",shift[2])
      ..") withpen pencircle scaled 1pt "
      .."withcolor (0.7,0.3,0.3);"
return str; end
```

5.1 Generate images for any $t \in [0,1]$

To create the animation, we generate the images with a discretization of the time interval $[0,1]$. This can be done with the following function.

```
function plotDecompAnim(letter,nbrPoints,
    nbrFourier,scale,nbrFrame)
-- letter: character that we want to decompose
-- nbrPoints: number of points in discretization
```

[4] There is also a frame drawn around it to make sure that all images have the same size and thus be able to chain the images to produce an animation.

```
-- nbrFourier: number of Fourier coefficients
-- scale: homothetic coefficient
-- nbrFrame: frame number
local str
local path = getpathfrommp(letter,nbrPoints,
                           scale)
local complexPath = pathToComplex(path)
local cnList = cnList(complexPath)
for frame=0,nbrFrame-1 do
 t = frame/nbrFrame
 str = coreDecomp(path,complexPath,cnList,
    nbrFourier,t)
 tex.sprint(str)
end
end
```

The Lua functions presented are all put in a single `Fourier.lua` file.

6 Animations and code

Once all these Lua functions are implemented, we just have to load them and call the Lua function `plotDecompAnim` using the command `\directlua`, as shown in the following code.

```
\documentclass{article}
\usepackage{luapackageloader}
\usepackage{luamplib}
\directlua{dofile("Fourier.lua")}
\pagestyle{empty}
\begin{document}
\directlua{
  plotDecompAnim("f",300,50,0.26,360)
}
\end{document}
```

To conclude, we have considered only three files: the LaTeX file above `fourier.tex`, the `Fourier.lua` file which contains all our Lua functions presented here, and the `complex.lua` file retrieved from the web. To compile and produce the PDF, which contains as many pages as there are images, we put these three files in the same directory and run `lualatex` on our `fourier.tex` file:

```
$ lualatex fourier.tex
```

If you read this article as a PDF file with Acrobat Reader, you will be able to see the generated animation (cf. figure 2) with the `animate` package [1]. Otherwise, all the code and the animation are visible and downloadable here:

`fougeriens.org/~mc/?page=exemples&dir=fourier`

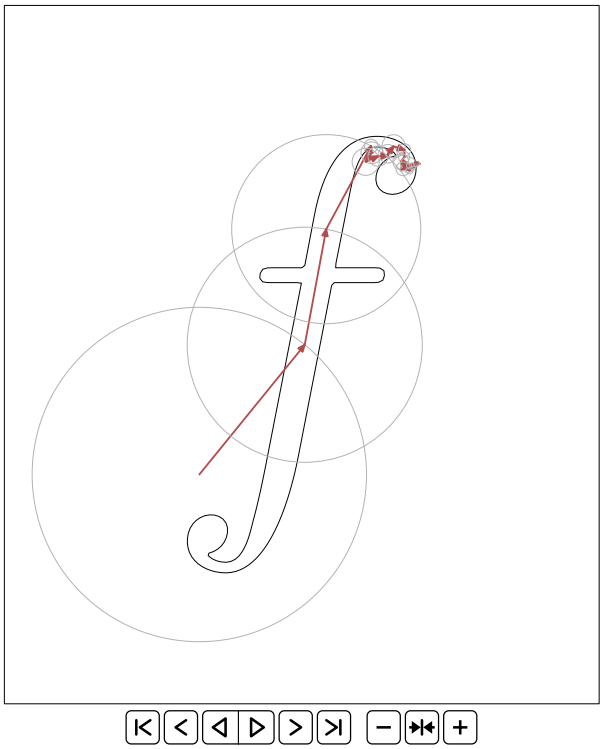


Figure 2: Animation

References

[1] A. Grahn. The `animate` package, 2020. `https://ctan.org/pkg/animate`.

[2] Association GUTenberg. *La Lettre GUTenberg* numéro 41, Décembre 2020. `https://www.gutenberg.eu.org/IMG/pdf/lettre41.pdf`.

[3] H. Hagen. *Metafun*. `http://www.pragma-ade.com/general/manuals/metafun-p.pdf`, 2020. v. 2.11.3.

[4] H. Hagen, T. Hoekwater, et al. The `luamplib` package. `https://ctan.org/pkg/luamplib`, 2020. v. 2.11.3.

[5] J.D. Hobby, MetaPost Development Team. *MetaPost, a user's manual*. `https://ctan.org/pkg/metapost`, 2019. v. 2.0.

[6] R. Ierusalimschy. *Programming in Lua*. `Lua.org`, 2016.

[7] LuaTeX development team. *LuaTeX Reference Manual*. `http://www.luatex.org/svn/trunk/manual/luatex.pdf`, March 2020. v. 1.12.

◇ Maxime Chupin
29 rue Pierre et Marie Curie
91400 Orsay
France
`mc (at) melusine dot eu dot org`
`https://fougeriens.org/~mc/`