`phystricks` Manual
If Sage can compute it, LaTeX can draw it

Laurent Claessens

April 18, 2016

Contents

1 Preparation

1.1 Dependencies and installation

1. You need a working sage installation.
2. Download `phystricks` from github and make it available from Sage (`from phystricks import *` has to work).
3. I don't even speak about having a working LaTeX installation with Tikz installed.

1.2 In your LaTeX file

The preamble of your LaTeX file has to contain

```
\usepackage{calc}
\usepackage{tikz}
\usetikzlibrary{patterns}
\usetikzlibrary{calc}
\newcounter{defHatch}
\newcounter{defPattern}
\setcounter{defHatch}{0}
\setcounter{defPattern}{0}
```

and you (don't really) have to compile with `pdflatex -shell-escape`.

1.3 Where do I find examples ?

You will found (figuratively) tons of examples in the following documents :

1. In the demo document. The sources are included in the `phystricks`'s repository; in the subdirectory `phystricks/testing/demonstration`. Browse the pdf at `http://laurent.claessens-donadello.eu/pdf/phystricks-demo.pdf`.
2. In mazhe. Download the source at `https://github.com/LaurentClaessens/mazhe/` and browse the pdf at `http://laurent.claessens-donadello.eu/pdf/mazhe.pdf`.
3. In smath.Download the source at `https://github.com/LaurentClaessens/smath/` and browse the pdf at `http://laurent.claessens-donadello.eu/pdf/smath.pdf`.

Since every single functionality of `phystricks` is used in at least one picture of `mazhe` or `smath`, we are not going to give so much examples in this document.

You are also invited to read the file `Constructors.py`; the docstring are explaining the creation of most of the graph types.

If you need something special or if you encounter any difficulty, send me an email.

1.4 Structure of your phystricks file

Most of your `phystricks`files will have the following structure :

```
# -*- coding: utf8 -*-
from phystricks import *
def QLXFooBDalHMaT():
    pspict,fig = SinglePicture("QLXFooBDalHMaT")
    #pspict.dilatation_X(1)
    #pspict.dilatation_Y(1)
    pspict.dilatation(1)

    ################################
    # The important lines are here
    # Define here your objects
    # example :
    P=Point(1,3)

    pspict.DrawGraphs(P)
    pspict.DrawDefaultAxes()

    ##############################

    fig.no_figure()
    fig.conclude()
    fig.write_the_file()
```

We will see later the significance of these lines.

2 Draw points

Here is the code corresponding to one red point with two marks.

```
from phystricks import *

def OnePoint():
    pspict,fig = SinglePicture("OnePoint")

    P = Point(1,1)
    P.parameters.color = "red"
    P.put_mark(dist=0.2,angle=30,text="\(P\)",automatic_place=(pspict,""))
    P.put_mark(dist=0.2,angle=-90,text="\(Q\)",automatic_place=(pspict,""))

    pspict.DrawGraph(P)

    fig.no_figure()
    fig.conclude()
    fig.write_the_file()
```

1. Compile it once in the Sage terminal :

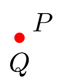
```
sage: attach("phystricksOnePoint.py");OnePoint()

The result is on figure \ref{LabelFigOnePoint}. % From file OnePoint
\newcommand{\CaptionFigOnePoint}{<+Type your caption here+>}
\input{Fig_OnePoint.pstricks}
Warning: the auxiliary file LabelFigOnePoint.phystricks.aux seems not ↩
   to exist.
 Compile your LaTeX file.
This is a second (or more) mark on the same point
--------------- For your LaTeX file ---------------

\begin{center}
   \input{Fig_OnePoint.pstricks}
\end{center}
---------------------------------------------------
sage:
```

2. As suggested by the Sage's output input the file `Fig_OnePoint.pstricks` in your LaTeX document.
3. Compile your document with `pdflatex <mydocument> -shell-escape`
4. Re-do the compilation in Sage
5. Re-do the LaTeX compilation.

If you don't compile twice, some elements can be badly placed, especially the marks that you put on points like the P and Q in this example.

If you want to know why, this is related to the mechanism of catching the LaTeX's internal counters(here the size of the box) by `phystricks`, see section 6.

The result should be



3 Drawing curves

All the curves are internally converted into parametric curve and then transformed into a large number of small segments. Tikz will only see these segments. For that reason, we are able to draw virtually anything that Sage con compute : we are not bound by Tikz's internals, and even less by LaTeX's ~~legacyembarrassingmade me crazy~~ limitations.

3.1 Drawing functions

For drawing the function $x \mapsto x^2$ on $[mx, Mx]$ the syntax is :

```
x=var('x')
f=phyFunction(x**2).graph(mx,Mx)
pspict.DrawGraphs(f)
```

The function itself (what is inside the `phyFunction` argument) is a Sage expression, so respecting the Sage syntax and using any function that Sage know.

In fact you can put inside `phyFunction` (I guess) anything that has a `__call__` method, as long as it returns real numbers.

The following is legal:

```

def fun(b):
    x=var('x')
    f=sin(x)/x
    s=numerical_integral(f,0.1,b)[0]
    return s

def MyPictureName():
    pspict,fig = SinglePicture("MyPictureName")
    f=phyFunction(fun).graph(0,10)

    pspict.DrawGraphs(f)
```

and draws the graph of

$$x \mapsto \int_{0.1}^{x} \frac{\sin(t)}{t} dt. \tag{1}$$

3.2 Parametric curve

For the curve

$$\begin{aligned} \gamma \colon [a,b] &\to \mathbb{R}^2 \\ t &\mapsto (f_1(t), f_2(t)) \end{aligned} \tag{2}$$

the syntax is :

```
f1=phyFunction( ... )
f2=phyFunction( ... )
curve=ParametricCurve( f1,f2,interval=(a,b)  )
```

You can omit the `interval` argument; in this case the interval of `f1` will be used, but such implicit transfer of property is a bad practice[1].

Here is an example code :

```
# -*- coding: utf8 -*-
from phystricks import *
def LARGooSLxQTdPC():
    pspict,fig = SinglePicture("LARGooSLxQTdPC")
    pspict.dilatation(3)

    x=var('x')
    f1=phyFunction( sin(2*x)  )
    f2=phyFunction( cos(3*x) )
    curve=ParametricCurve(f1,f2,interval=(0,2*pi))

    pspict.DrawGraphs(curve)
    pspict.DrawDefaultAxes()
    pspict.comment="There is a lack of plotpoints, and this is normal ↩
        because this picture comes from the documentation."

    fig.conclude()
    fig.write_the_file()
```

The result is on figure 1. You see that too few points are plotted, so that the picture is not quite well curved. This problem can be fixed using the `plotpoints` attribute of the curve; we will see that later.

[1]`phystricks` contains lots of such "if an argument is missing I will search it somewhere" mechanisms.

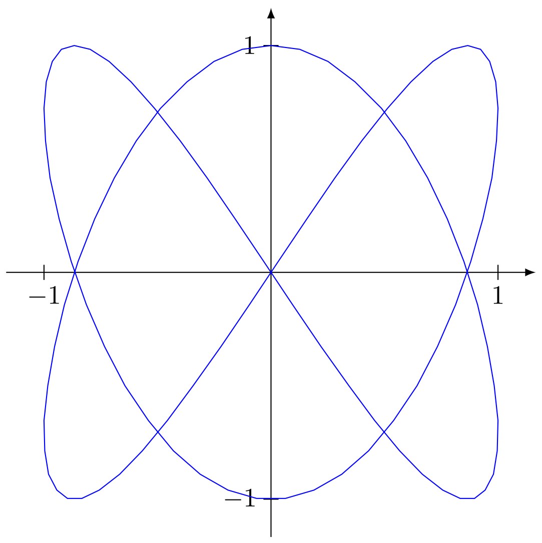


Figure 1: This is a parametric curve, a Lyssajou.

3.3 Interpolation curve

<++>

3.4 Lagrange polynomial, Hermite interpolation

<++>

3.5 Compute more plotpoints (sample)

As seen on figure 1, the default setting does not compute enough «intermediate» points to produce a visually correct result on some curves.

The easiest way to make the curve more smooth is to increase the `plotpoints` attribute; as an example :

```
f=phyFunction( sin(x)/x  ).graph(0.01,5)
f.parameters.plotpoints=500
```

For fixing the ideas, let's say `plotpoints=100`. Then the default behaviour is to consider 100 values *of the parameters* that regularly spaced between its minimum and its maximum. The drawn curve is then the interpolation curve of the corresponding points.

This is not always adapted, and we have two ways to adapt this mechanism to particular cases.

Add selected plot points We can make compute some more points by adding parameters values to the list `added_plotpoints` :

```
curve=ParametricCurve( f1,f2,interval=(0,1)  )
curve.parameters.added_plotpoints=[  0.001,  pi/5    ]
```

In this case we will compute 102 points : the usual 100 plus the ones corresponding to the values 0.001 and $\pi/5$ of the parameters.

Force smoothing We can do

```
curve=ParametricCurve( f1,f2,interval=(0,1)  )
curve.parameters.force_smooth=True
```

In this case, the 100 interpolation points will be taken regularly spaced with respect to the integral of the curvature. In other words $\{x_i\}_{i=1,\ldots,50}$ are chosen in such a way that

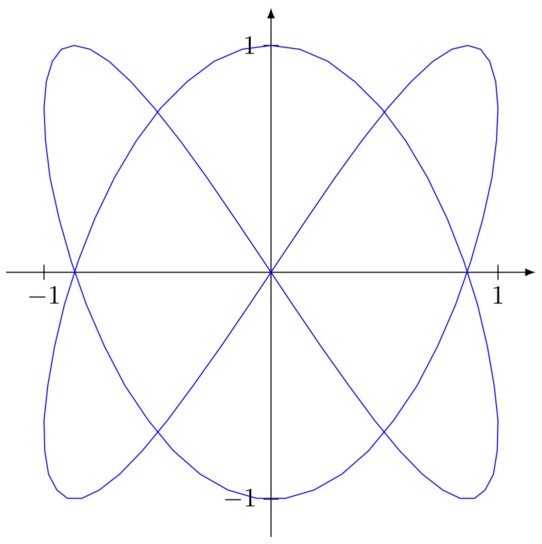
$$\int_{x_i}^{x_{i+1}} c(t)dt \tag{3}$$

is constant with respect to to i.

An example :

```
# -*- coding: utf8 -*-
from phystricks import *
def PBFCooV1PiRBpt():
    pspict,fig = SinglePicture("PBFCooV1PiRBpt")
    pspict.dilatation(3)

    x=var('x')
    f1=phyFunction( sin(2*x)  )
    f2=phyFunction( cos(3*x) )
    curve=ParametricCurve(f1,f2,interval=(0,2*pi))

    curve.parameters.force_smoothing=True

    pspict.DrawGraphs(curve)
    pspict.DrawDefaultAxes()
    pspict.comment="There is a lack of plotpoints, and this is normal ↩
       because this picture comes from the documentation."

    fig.no_figure()
    fig.conclude()
    fig.write_the_file()
```

.



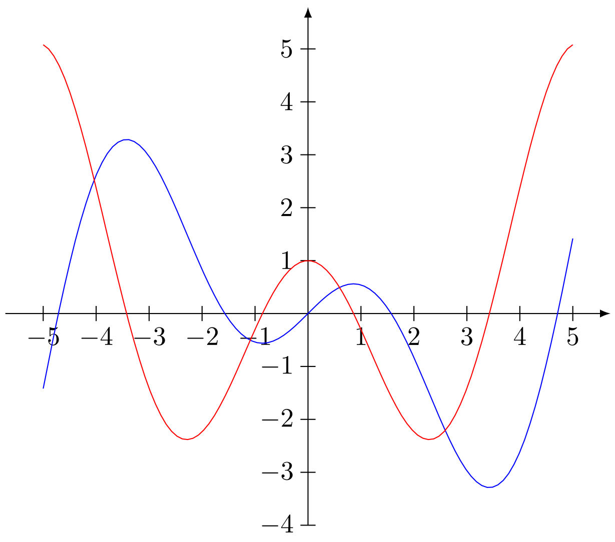
Is it better that figure 1 ? The four angles are for sure smoother. However, the computation of these points at “regular curvature” interval can take forever and it is often much faster to simply add thousands of plotpoints.

3.6 Derivative, tangent, and other differential geometry

A `phyFunctionGraph` object has a method `derivative` that returns a `phyFunction` of the derivative.

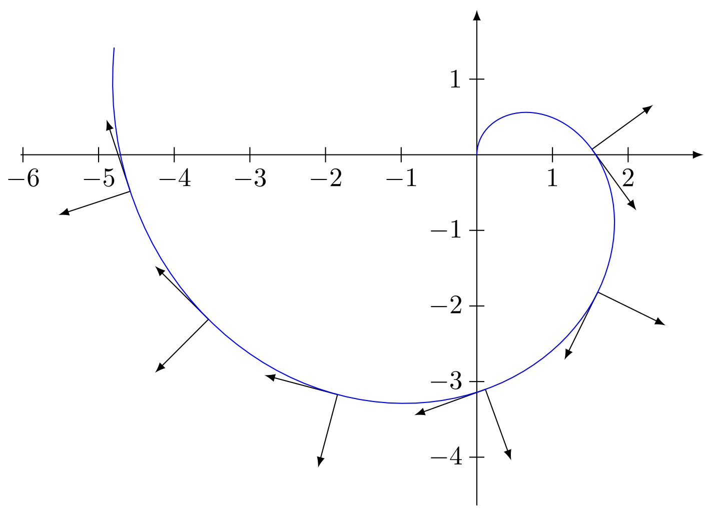
Here is an example code :

```
from phystricks import *
def FunctionThird():
    pspict,fig = SinglePicture("FunctionThird")
    pspict.dilatation(0.7)

    var('x')
    f = phyFunction( x*cos(x) )
    mx = -5
    Mx = 5
    F = f.graph(mx,Mx)
    G = f.derivative().graph(mx,Mx)
    G.parameters.color = "red"
    pspict.DrawGraph(F)
    pspict.DrawGraph(G)

    pspict.DrawDefaultAxes()
    pspict.comment="The function \( x\cos(x)\) (blue) and its derivative (↩
        red)."

    fig.no_figure()
    fig.conclude()
    fig.write_the_file()
```



You also have methods to get the tangent and normal vector. Here is an example of taking some tangent and normal vectors :

```
# -*- coding: utf8 -*-
from phystricks import *
def GKMEooBcNxcWBt():
    pspict,fig = SinglePicture("GKMEooBcNxcWBt")
    var('x')
    f1 = phyFunction( x*sin(x) )
    f3 = phyFunction( x*cos(x) )

    llI = 0
    llF = 5
    F2 = ParametricCurve(f1,f3,interval=(llI,llF))

    for ll in F2.getRegularLengthParameters(llI,llF,2):
        v1 = F2.get_tangent_vector(ll)
        v2 = F2.get_normal_vector(ll)
        pspict.DrawGraphs(v1,v2)

    pspict.DrawGraphs(F2)
    pspict.DrawDefaultAxes()

    fig.no_figure()
    fig.conclude()
    fig.write_the_file()
```



By the way you should note the method `getRegularLengthParameters` that return a list of parameters value such that the corresponding points are regularly spaced on the curve (with respect to the arc length). Namely

```
F2.getRegularLengthParameters(llI,llF,2):
```

returns a list of parameters such that the arc length between two points is 2.

4 Figure, subfigure

1. The caption of the figure is not given in the `phystricks` code, but has to be inserted in the LaTeX document.
2. On the contrary, the subfigures caption are from the `phystricks` code

You should use the utility `new_picture.py` to generate the skeleton of your figure.

5 Put marks on the objects

One can always put a mark on an object; the position is in part automatically determined. The general statement is :

```
      obj.put_mark(dist,angle,text,mark_point=None,automatic_place=False,↩
          added_angle=None,pspict=None)
```

6 How to get the LaTeX counters ?

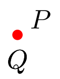
We are going to explain one important mechanism in `phystricks` about its interaction with LaTeX. For we consider the code

```
from phystricks import *

def OnePoint():
    pspict,fig = SinglePicture("OnePoint")

    P = Point(1,1)
    P.parameters.color = "red"
    P.put_mark(dist=0.2,angle=30,text="\(P\)",automatic_place=(pspict,""))
    P.put_mark(dist=0.2,angle=-90,text="\(Q\)",automatic_place=(pspict,""))

    pspict.DrawGraph(P)

    fig.no_figure()
    fig.conclude()
    fig.write_the_file()
```

We compile it in a Sage terminal :

```
sage: attach("phystricksOnePoint.py");OnePoint()

The result is on figure \ref{LabelFigOnePoint}. % From file OnePoint
\newcommand{\CaptionFigOnePoint}{<+Type your caption here+>}
\input{Fig_OnePoint.pstricks}
Warning: the auxiliary file LabelFigOnePoint.phystricks.aux seems not to exist.
 Compile your LaTeX file.
This is a second (or more) mark on the same point
--------------- For your LaTeX file ---------------

\begin{center}
   \input{Fig_OnePoint.pstricks}
\end{center}
---------------------------------------------------
sage:
```

If you input now the file `Fig_OnePoint.pstricks` in your LaTeX document, you'll see a beautiful red point with two marks, a P and a Q.



However, the marks are badly placed, this is the sense of the warning about the existence of the file `LabelFigOnePoint.phystricks.aux`. In fact the file `Fig_OnePoint.pstricks` does not only contains the tikz code for the picture, but also a pure LaTeX code asking latex to write the dimensions of the boxes P and Q in an auxiliary file.

Just in order to make is cryptic, these are lines like :

```
\makeatletter\@ifundefined{writeOfphystricks}{\newwrite{\writeOfphystricks}}{}\makeatother%
\setlength{\lengthOfhomemokyDOTSagesrcbinsageipython}{\totalheightof{\(P\)}}%
\immediate\write\writeOfphystricks{totalheightof1903839d9021e180dd790c4cc63081c63b2fe6f1:\the\lengthOfh
```

Now you can reenter Sage and recompile the picture :

```

sage: attach("phystricksOnePoint.py");OnePoint()

The result is on figure \ref{LabelFigOnePoint}. % From file OnePoint
\newcommand{\CaptionFigOnePoint}{<+Type your caption here+>}
\input{Fig_OnePoint.pstricks}
This is a second (or more) mark on the same point
--------------- For your LaTeX file ---------------

\begin{center}
   \input{Fig_OnePoint.pstricks}
\end{center}
---------------------------------------------------
```

The warning disappeared and now `phystricks` has read the auxiliary file containing the dimensions of the boxes. The P and Q are then now placed taking their *real* dimension into account.

The auxiliary file contains the lines

```
totalheightof1903839d9021e180dd790c4cc63081c63b2fe6f1:6.83331pt-
widthof1903839d9021e180dd790c4cc63081c63b2fe6f1:7.80904pt-
totalheightof15a6448f2b408bb6a0dabb437cc671b7beb909fc:8.77776pt-
widthof15a6448f2b408bb6a0dabb437cc671b7beb909fc:7.90555pt-
```

The box is identified by a hash of its LATEX code. The reason is that almost(?) any string can be valid LATEX code[2], so the parsing of this auxiliary file is more or less impossible if the actual LATEX code is included.

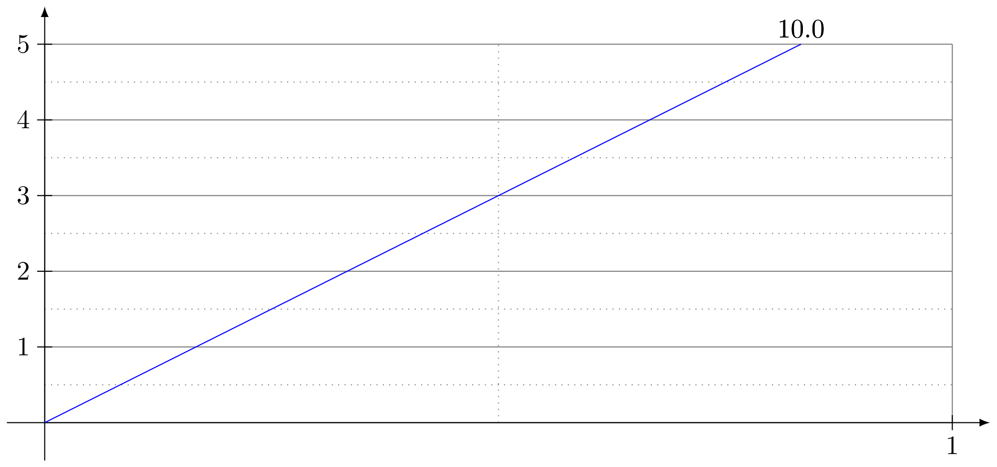
Relaunch pdfLATEX and you'll see the points correctly placed.

Conclusion : when you add some LATEX code in your picture, you need one more pass of pdfLATEXand `phystricks` in order to get the marks right.

This mechanism of making LATEX write values in an auxiliary file is general and any latex internal counters can be accessed in your python code (as Python's `float`).

You don't believe ? Here is a picture with the following specifications :

1. The line slope is the number of the section (here we have 6=6).
2. The line is drawn from $x = 0$ to $x = x_{max}$ computed in such a way that $y_{max} = 5$.
3. A dilatation in the x-direction is computed in such a way that the picture has 10 cm length.
4. The page number is written just on the top of coordinates (x_{max}, y_{max}).



Obviously this kind of picture has to be recompiled each time we change the containing document.

Here is the code :

```python
# -*- coding: utf8 -*-

from __future__ import division
from phystricks import *

def RJDEoob0ibtkfv():
    pspict,fig = SinglePicture("RJDEoob0ibtkfv")

    # Taking the value of the LaTeX's counters "section" and "page"
    section=pspict.get_counter_value("section",default_value=1)
    page=pspict.get_counter_value("page")

    # You compute with is as normal Python float
    xmax=5/section
    pspict.dilatation_X(10/xmax)

    # Create the picture itself using the computed numbers :
    x=var('x')
    f=phyFunction(section*x).graph(0,xmax)
    f.put_mark(0.2,angle=None,added_angle=0,text="\( {} \)".format(page),↩
        automatic_place=(pspict,""))
```

[2]Thanks to the `catcode` mechanism, it seems to me that latex is the most introspective programming language ever.

```python


    pspict.DrawGraphs(f)
    pspict.DrawDefaultAxes()
    pspict.DrawDefaultGrid()
    pspict.comment=r"""
    \begin{enumerate}
    \item
    slope of the line is equal to the section number
    \item
    the page number is written.
    \item
    the X dilatation makes the real picture measure 10cm
    \end{enumerate}
    """
    fig.no_figure()
    fig.conclude()
    fig.write_the_file()
```

The default value for the `section` counter is given to avoid division by zero, because zero is the default-default value : the one which is returned at first compilation. when the auxiliary file does not yet contain the value of the counter (there is a bootstrap here).