

# phystricks Manual

## An object-oriented way of thinking pstricks

Laurent Claessens

April 11, 2012

### Abstract

When the examples are in contradiction with the text, you have to believe the examples because they are regularly re-compiled and the source that are printed here are verbatim input of the sources that are producing the figures.

This documentation as well as the source code is hosted on [gitorious](#).

Many of the examples come from a course of [analytic geometry](#) (in French).

### WARNING:

This document is dedicated to show the result in an actual  $\text{\LaTeX}$  document. The full documentation is located at [http://student.ulb.ac.be/~lclaessee/phystricks-documentation/\\_build/html/index.html](http://student.ulb.ac.be/~lclaessee/phystricks-documentation/_build/html/index.html)

## Contents

### 0.1 Hello world!

```
#!/usr/bin/sage -python
# -*- coding: utf8 -*-

from phystricks import *

def OnePoint():
    pspict,fig = SinglePicture("OnePoint")           # Generic name of the figure

    ##### The main lines are here
    P = Point(1,1)
    P.parameters.color = "red"
```

```

pspict.DrawGraph(P)
#####

fig.conclude()
fig.write_the_file()

```

OnePoint()

The result is given on the figure 1



Figure 1: This is my first point

Some customisations are exemplified in the following code:

```

from phystricks import *
def MarkOnPoint():
    pspict,fig = SinglePicture("MarkOnPoint")

    P = Point(0,0)
    P.parameters.color = "blue"
    P.put_mark(0.3,180,r"$f_i$")

    Q = Point(1,1)
    Q.put_mark(0.3,0,"$q$")
    Q.parameters.symbol = "diamond"

    pspict.DrawGraphs(P,Q)
    pspict.DrawGraph(P.mark.bounding_box(pspict))

    fig.conclude()
    fig.write_the_file()

```

The result is on figure ??.

◇ *q*



Figure 2: A point with a mark. Notice that the mark text is a string that is to be interpreted by L<sup>A</sup>T<sub>E</sub>X.

## 0.2 Positioning the mark

We already saw that the basic way of positioning a mark on a point is to precise a distance and an angle :

```
P.put_mark(0.3,45,"$A$")
```

will put a  $A$  at distance 0.3 and angle  $45^\circ$  from the point  $A$ . More precisely, the *center* of the box will be put at that place.

The mark can also be put more or less automatically taking into account the size of the  $\text{\LaTeX}$  box containing the mark. The first way is to use `automatic_place=pspict`. Writing

```
P.put_mark(0.3,30,"$A$",automatic_place=pspict)
```

the *bottom left* corner of the box will be placed at position (0.3;30). There are four corners; the correct one is automatically chosen.

In the following example, we use the method `angle`. Writing `P.angle()` returns the angle of the vector  $OP$ .

```
from phystricks import *
def DefinitionCartesiennes():
    pspict,fig = SinglePicture("DefinitionCartesiennes")

    def PlacePoint(x,y,nom,color):
        M=Point(x,y)
        #print M,numerical_approx(M.angle())
        M.put_mark(0.1,M.angle(),"$(%s,%s)$"%(str(x),str(y)),automatic_place=pspict)
        Px=M.projection(pspict.single_axeX)
        Py=M.projection(pspict.single_axeY)
        seg1=Segment(M,Px)
        seg2=Segment(M,Py)
        seg1.parameters.color=color
        seg1.parameters.style="dashed"
        seg2.parameters=seg1.parameters
        pspict.DrawGraphs(seg1,seg2,M,M.mark.bounding_box(pspict))

    PlacePoint(3,1,"A","blue")
    PlacePoint(-1.5,-2.5,"B","green")
    PlacePoint(-1,2.5,"C","brown")
    PlacePoint(1.5,-1,"D","cyan")
    pspict.DrawDefaultAxes()
    pspict.dilatation(1)
```

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

The result is on figure ??

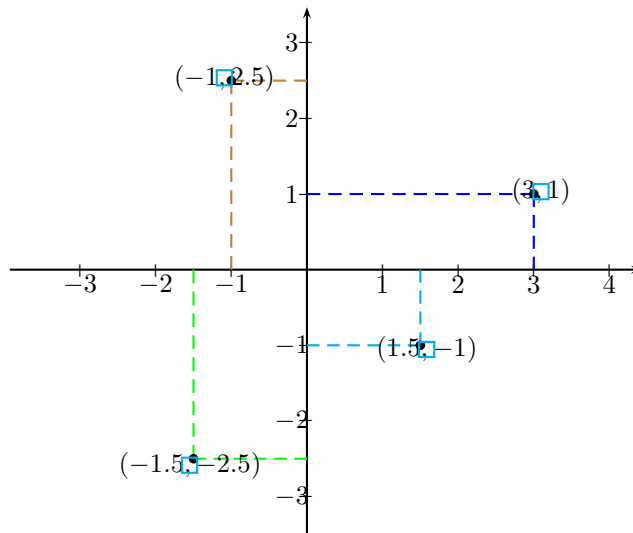


Figure 3: Intelligent positioning of the mark. Notice that you can draw the box (in the L<sup>A</sup>T<sub>E</sub>X sense of the term) of the textes.

Instead of force the position of the corner, you can also force the position of the center of one of the four sides of the box. The syntax is

```
P.put_mark(<distance>,<angle>,<text>,automatic_place=(<pspict>,<position>))
```

where position is one of N, S, W, E following the side you want to force.

```
from physticks import *
def EdgePosition():
    pspict,fig = SinglePicture("EdgePosition")

    P=Point(0,1)
    Q=Point(1,0)
    R=Point(0,-1)
    S=Point(-1,0)
    text=r"$\alpha_y\alpha_zv$"
    P.put_mark(0.1,90,text,automatic_place=(pspict,"S"))
    Q.put_mark(0.1,0,text,automatic_place=(pspict,"W"))
    R.put_mark(0.1,-90,text,automatic_place=(pspict,"N"))
```

```

S.put_mark(0.1,180,text,automatic_place=(pspict,"E"))

pspict.DrawGraphs(P,Q,R,S)
bbP=P.mark.bounding_box(pspict)
bbQ=Q.mark.bounding_box(pspict)
bbR=R.mark.bounding_box(pspict)
bbS=S.mark.bounding_box(pspict)
pspict.DrawGraphs(bbP,bbQ,bbR,bbS)
#pspict.DrawDefaultAxes()
pspict.dilatation(1)
fig.conclude()
fig.write_the_file()

```

The result is on figure ??.

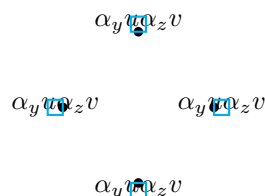


Figure 4: Examples of intelligent placing of the mark. The center of the chosen edge is at the given place. This requires two compilations (python+ $\text{\LaTeX}$ ).

### 0.3 Sequence of points

You know that python is a programming language<sup>1</sup>. The ultimate purpose of `physticks` is to allow the use of that powerful programming language in order to create figures.

The following code draw the first 10 points of the sequence  $x_n = \frac{(-1)^n}{n}$ .

```

from physticks import *
def Sequence():
    pspict,fig = SinglePicture("Sequence")

    nmax = 10
    P = []
    for i in range(1,nmax):

```

---

<sup>1</sup>The author, mainly by ignorance, is used to troll against  $\text{\LaTeX}$  as programming language.

```

x = i
y = ((-1)**i)/float(i)
P = Point(x,y)
P.put_mark(0.3,90*(-1)**i,"$P_{%s}$"%(str(i)))
pspict.DrawGraph(P)

fig.conclude()
fig.write_the_file()

```

The result is on figure ??.

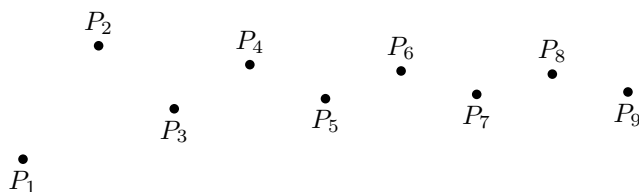


Figure 5: The first points of  $P_n = (-1)^n/n$ .

A small remark about the bounding box. If you compile this figure by the sequence

```

./MyScript.py --pdf
pdflatex MyDocument.tex

```

then the texts on the points will not be taken into account in the bounding box (and can be cut). You have to do the whole twice:

```

./MyScript.py --pdf
pdflatex MyDocument.tex
./MyScript.py --pdf
padlatex MyDocument.tex

```

In the following example, we use the Sage's function `repr` in order to write down the numbers.

```

# -*- coding: utf8 -*-
from physticks import *
def SuiteUnSurn():
    pspict,fig = SinglePicture("SuiteUnSurn")

    def suite(i):
        return SR(1)/i

```

```

n=10
for i in range(1,n+1):
    y=suite(i)
    P=Point(i,float(y))
    P.put_mark(0.3,90,"$%s$"%(repr(y)))
    pspict.DrawGraph(P)

pspict.axes.no_graduation()
pspict.DrawDefaultAxes()

pspict.dilatation_Y(3)

fig.conclude()
fig.write_the_file()

```

The result is on figure ??.

## 0.4 Points in polar coordinates

A point in polar coordinates is created by `PolarPoint(r,theta)`. It returns an instance of the class `Point`, so that

```
print PolarPoint(1,45).coordinates()
```

will still print the Cartesian coordinates of that point.

## 1 Draw lines

The module `phystricks` makes usually the difference between two kind of objects:

1. the “geometric” object. Here, a line.
2. The graph of the object. Here, a line endowed with a line style, a color, ...

The class `Segment` implements what you need to deal with lines. A segment is defined by two points: the initial point and the final point.

The following code shows the coordinates of the initial and final point of a segment.

```
A = Point(1,1)
B = Point(5,2)
```

```

segment = Segment(A,B)
print segment.I.coordinates()
print segment.F.coordinates()

```

Here, `segment.I` is the initial point of `segment`. The method `coordinates` returns the coordinates of a point.

In the following example, we define two segments and we draw them. The first is drawn with default options while the second is dotted in red. The intersection point is also computed by the function `LineInterLine`.

We also draw a wavy line using the method `wave`. When `S` is an instance of `GraphOfASegment`, we make it wavy by writing

```
S.wave(dx,dy)
```

where `dx` is the wavelength and `dy` is the amplitude of the wave we want to draw. The wave will always begins on the initial point of the segment and end on the last point. Thus the last wavefront can be smaller than `dx`.

The result is on figure ??.

```

from phystricks import *
def Lines():
    pspict,fig = SinglePicture("Lines")

    A = Point(1,1)
    B = Point(5,2)
    C = Point(2,5)
    A.put_mark(0.3,180,"$A$")
    B.put_mark(0.3,0,"$B$")
    C.put_mark(0.3,180,"$C$")
    pspict.DrawGraph(A)
    pspict.DrawGraph(B)
    pspict.DrawGraph(C)
    segment1 = Segment(A,B)
    segment2 = Segment(C,Point(3,-1))

    pspict.DrawGraph(segment1)

    segment2.parameters.color = "red"
    segment2.parameters.style = "dotted"
    pspict.DrawGraph(segment2)

    P = Intersection(segment1,segment2)[0]

```

```

P.put_mark(0.3,-45,"P$")

segment3 = Segment(A,C)
S3 = segment3.graph()
S4 = segment3.graph()
S3.wave(0.2,0.1)
S3.parameters.color = "cyan"
S4.parameters.color = "blue"
pspict.DrawGraph(S3)
pspict.DrawGraph(S4)

fig.conclude()
fig.write_the_file()

```

## 2 Rectangles

```

from phystricks import *
def RectangleOne():
    pspict,fig = SinglePicture("RectangleOne")

    rect=Rectangle( Point(1,1),Point(2,3) )
    rect.parameters.hatched()
    rect.parameters.hatch.color="red"
    rect.parameters.style="dotted"
    rect.parameters.color="blue"

    pspict.DrawGraphs(rect)

    pspict.DrawDefaultAxes()
    pspict.dilatation(1)

    fig.conclude()
    fig.write_the_file()

```

The result is on figure ??

## 3 Angles

```

from phystricks import *
def TriangleRectangle():

```

```

pspict,fig = SinglePicture("TriangleRectangle")

B=Point(0,0)
C=Point(1,0)
A=Point(0.5,0.5*sqrt(3))

A.put_mark(0.3,90,"$A$")
B.put_mark(0.3,180,"$B$")
C.put_mark(0.3,0,"$C$")

AB=Segment(A,B)
AC=Segment(A,C)
BC=Segment(C,B)

H=BC.center()
hauteur=Segment(A,H)
hauteur.parameters.style="dotted"
hauteur.parameters.color="blue"
H.put_mark(0.3,-90,"$H$")

angleS=Angle(A,C,H)
angleT=Angle(H,A,C)
angleS.parameters.color="red"
angleT.parameters.color="cyan"
angleS.put_mark(0.3,angleS.advised_mark_angle,"$60$")
angleT.put_mark(0.3,angleT.advised_mark_angle,"$30$")

pspict.DrawGraphs(AB,AC,BC,hauteur,A,B,C,H,angleS,angleT)
pspict.dilatation(4)
fig.conclude()
fig.write_the_file()

```

The result is on figure ??.

Sometimes, for aesthetics reasons you don't want the mark of the angle to be put at the center. The following code uses

```
phi.set_mark_angle(new_angle)
```

to put the mark at `new_angle` instead of the middle. Since `phi.angleF.degree` is the final angle of `phi` (in degree), writing

```
phi.set_mark_angle(0.5*(90+phi.angleF.degree))
```

put the mark at the middle between the vertical and the second segment of the angle.

```
from physticks import *
def TgCercleTrigono():
    pspict,fig = SinglePicture("TgCercleTrigono")

    O=Point(0,0)
    X=Point(1,0)
    Cercle=Circle(0,2)
    Q=Cercle.get_point(150)
    vQ=Vector(Q)

    Cercle.parameters.color="lightgray"

    phi=Angle(X,O,Q,0.5)
    phi.set_mark_angle(0.5*(90+phi.angleF.degree))
    phi.put_mark(0.3,phi.advised_mark_angle,r"$\varphi$")

    M=phi.mark_point()

    vQ.parameters.color="blue"
    M.parameters.color="brown"
    phi.parameters.color=vQ.parameters.color

    pspict.DrawGraphs(phi,vQ)
    pspict.DrawGraphs(M)
    pspict.axes.no_graduation()
    pspict.DrawDefaultAxes()
    pspict.dilatation(1)
    fig.conclude()
    fig.write_the_file()
```

The result is on figure ??.

## 4 Circles

There exists the class `Circle`. We create a circle with its center (class `Point`) and its radius. It has the following attributes that do not need explanations :

```
Circle.centre  
Circle.radius
```

The method

```
Circle.ParametricCurve()
```

returns the parametric curve (class `ParametricCurve`) that describes the circle. See the (not yet existing) documentation of that class.

For drawing circle follows the same rules as the segments.

```
from phystricks import *  
def exCircle():  
    pspict,fig = SinglePicture("exCircle")  
  
    C = Circle(Point(1,1),2)  
    C.parameters.color = "magenta"  
  
    pspict.DrawGraph(C)  
  
    fig.conclude()  
    fig.write_the_file()
```

The result is represented on ??.

Notice that the circle is centred at (1,1), while it appears centred on the page. It is due to the fact that `phystricks` computes the bounding box with the elements that are actually drawn.

An arc of circle is simply obtained by changing the values of `circle.angleI` and `circle.angleF` of the class `circle`. We get a point on the circle by giving its angle:

```
circle.get_point(theta)
```

returns the point on the circle at angle `theta`.

These features are illustrated by the following example (we anticipate on the axes):

```
from phystricks import *  
def exCircleTwo():  
    pspict,fig = SinglePicture("exCircleTwo")  
  
    C = Circle(Point(0,0),1.5)  
    C.style = "dotted"  
    C.angleI = 20
```

```

C.angleF = 100

for angle in [10,30,75,130,300,350] :
    P = C.get_point(angle)
    P.put_mark(0.5,angle,"$P_{%s}$"%str(angle))
    pspict.DrawGraph(P)

pspict.DrawGraph(C)

fig.conclude()
fig.write_the_file()

```

The result is presented on figure ??.

An arc of circle can also be wavy as shown in the following code:

```

from phystricks import *
def exCircleThree():
    pspict,fig = SinglePicture("exCircleThree")

    circle = Circle(Point(0,0),1.5)
    C = circle.graph(45,360)
    C.wave(0.1,0.1)
    C.color = "green"
    D = circle.graph(C.angleF.degree-360,C.angleI)
    D.wave(C.waviness.dx,C.waviness.dy)
    D.color = "red"

    pspict.DrawGraphs(C,D)
    pspict.DrawDefaultAxes()

    fig.conclude()
    fig.write_the_file()

```

The result is on figure ??.

If some parameters are not available from `parameters`, we can use `add_option`. When the latter is used, the string is immediately passed to `pstricks`. As an example, two ways to fill a circle on figure ??.

```

from phystricks import *
def FilledCircle():
    pspict,fig = SinglePicture("FilledCircle")

```

```

C=Circle(Point(0,0),1)
C.parameters.filled()
C.parameters.fill.color="yellow"
C.parameters.color="blue"

D=Circle(Point(2,0),1)
D.parameters.hatched()
D.parameters.hatch.angle=0
D.parameters.hatch.color="green"

E=Circle(Point(4,0),1)
E.add_option("fillstyle=solid")
E.add_option("fillcolor=green")
E.parameters.color="red"

pspict.DrawGraph(C)
pspict.DrawGraph(D)
pspict.DrawGraph(E)

pspict.dilatation(1)

fig.conclude()
fig.write_the_file()

```

## 5 Vectors

### 5.1 Vectors alone

Vectors are much like segments. Formally speaking, an object of the class `Vector` represents an element of the *affine* space of  $\mathbb{R}^2$ . It has a fixed origin. A vector is created by

```
v = Vector(P1,P2)
```

where `P1` and `P2` are of type `Point`. If you want to draw it, you have to create the object to be drawn as follows:

```
V = Graph(v)
```

Now, you can custom the graph. If you want to give a name or to put a mark on the vector, use the method `mark` as

```
V.mark(dist,angle,mark)
```

Rotated, orthogonal, dilated and resized vectors are obtained by the following method)

```
v.rotation(theta)
v.orthogonal()
v.dilatation(factor)
v.fix_size(size)
```

where `theta` is an angle *in degree*, `factor` is the dilatation factor (can be negative) and `size` is the size of the new vector. Here is an example:

```
from physticks import *
def VectorOne():
    pspict,fig = SinglePicture("VectorOne")

    O = Point(0,0)
    A = Point(1,1)
    B = Point(-4,-1)
    C = Point(-2,3)

    V = []
    V.append(AffineVector(O,A).fix_size(3))
    V.append(AffineVector(A,C))
    V.append(AffineVector(B,C))
    V.append( V[1].rotation(30).dilatation(0.5) )
    V.append( V[1].rotation(-30) )
    V.append( V[1].orthogonal() )

    V[1].put_mark(0.3,45,"$v$")
    V[1].parameters.color="brown"
    V[2].parameters.color="red"
    V[2].parameters.style = "dotted"
    V[3].parameters.color="blue"
    V[4].parameters.style = "dashed"
    V[4].parameters.color=V[1].parameters.color
    V[5].parameters.color=V[1].parameters.color

    for i in range(0,len(V)):
        pspict.DrawGraph(V[i])
```

```
pspict.DrawDefaultAxes()
```

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

The result is on figure ??

Here is an illustration of the dilatation method on segments and vectors:

```
from phystricks import *  
def Dilatation():  
    pspict,fig = SinglePicture("Dilatation")  
  
    A = Point(1,0)  
    B = Point(1,2)  
    v=AffineVector(A,B)  
    w1=v.dilatation(0.5)  
    w2=v.dilatation(1.5)  
    v.parameters.color="blue"  
    w1.parameters.color="green"  
    w2.parameters.color="red"  
    pspict.DrawGraphs(w2,v,w1)  
  
    C = Point(2,0)  
    D = Point(2,2)  
    s=Segment(C,D)  
    t1=s.dilatation(0.5)  
    t2=s.dilatation(1.5)  
    s.parameters.color="blue"  
    t1.parameters.color="green"  
    t2.parameters.color="red"  
    pspict.DrawGraphs(t2,s,t1)  
  
    pspict.axes.single_axeX.no_numbering()  
    pspict.DrawDefaultAxes()  
  
    fig.conclude()  
    fig.write_the_file()
```

The result is on figure ??.

## 5.2 Vector fields

```
from phystricks import *
def ChampVecteursDeux():
    pspict,fig = SinglePicture("ChampVecteursDeux")

    x,y=var('x,y')

    Field=VectorField( sin(x)/2,sin(y)/2 )
    F=Field.graph(xvalues=(x,-6,6,20),yvalues=(y,-6,6,20))

    pspict.DrawGraphs(F)
    pspict.dilatation(1)
    fig.conclude()
    fig.write_the_file()
```

The result is on figure ??.

Providing by hand the points to be drawn can makes good pictures :

```
from phystricks import *
def ChampVecteurs():
    pspict,fig = SinglePicture("ChampVecteurs")

    x,y=var('x,y')

    draw_points=[]
    for i in range(1,10):
        C=Circle(Point(0,0),float(i)/2)
        pts=C.get_regular_points(0,360,0.5)
        draw_points.extend(pts)

    l=(1.0/3)
    Field=VectorField(l*y/sqrt(x**2+y**2),l*-x/sqrt(x**2+y**2))
    F=Field.graph(draw_points=draw_points)

    pspict.DrawGraphs(F)
    pspict.dilatation(1)
    fig.conclude()
    fig.write_the_file()
```

The result is on figure ??.

## 6 Axes and grid

### 6.1 Good news: no bounding boxes

If you don't know what a bounding box is, I have a good news: you don't have to know it. If you know what the bounding box is, I have the same good news: `phystricks` computes it for you. When you draw a function (see below) `phystricks` relies on Sage to numerically compute the position of the maximum and minimum of the function. If you print the name of some points, the size of the letter is not taken into account in the computation of the bounding box, as in the example of figure ?? where the  $P$  is not taken into account in the bounding box.

### 6.2 The automatic way for the axes

The axes follow the same good rule: `phystricks` knows your picture and then knows the size of the axis you need. You can of course modify them by hand<sup>2</sup> but most of time, you just need the axis centered at zero and whose have the size of your picture.

The axes are automatically drawn if you use the method `DrawDefaultAxes()` on the object `pspict`. There are no mystery about how it works.

```
from phystricks import *
def Axes():
    pspict,fig = SinglePicture("Axes")

    P = Point(-4,-2)
    L = Segment( Point(0,0),Point(1.6,3) )

    P.put_mark(0.3,135,"$P$")
    pspict.DrawGraph(P)
    L.parameters.color = "brown"
    pspict.DrawGraph(L)

    pspict.DrawDefaultAxes()

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

The result is given on the figure ??

---

<sup>2</sup>Write an email to the author or read the code if you want to know how.

### 6.3 Some personalisation of the axes

There is a class `Axes`. The “default” axes we spoke about in the previous subsection is an object of that class and is a method of the class `pspict`

```
pspict.axes
```

Thus the easiest way to have customised axes is to custom these ones. Here is an example where we remove the graduation of the axes and where we put labels. In order to remove the graduation, we use the method `no_graduation()` of the class `Axes`

```
pspict.axes.no_graduation()
```

and in order to add labels, we use the methods `add_label_X` and `add_label_Y` in the following way

```
pspict.axes.add_label_X(d,alpha,mark)
```

where `d` is the distance between the arrow and the mark, `alpha` is the angle and `mark` is the name one wants to see. Typically, `mark` is a  $\LaTeX$  expression.

```
from physticks import *
def AxesSecond():
    pspict,fig = SinglePicture("AxesSecond")

    for i in range(-10,10):
        x = float(i)/5
        P=Point(2*x,sinh(x))
        P.parameters.symbol="*"
        pspict.DrawGraph(P)

    pspict.axes.no_graduation()
    pspict.axes.single_axeX.put_mark(0.3,-45,"$x$")
    pspict.axes.single_axeY.put_mark(1.3,0,"$y=\sinh(x)$")

    pspict.DrawDefaultAxes()

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

The result is on figure ??

The following code shows how to change the graduation of the axes.

```

from phystricks import *
def AxesThird():
    pspict,fig = SinglePicture("AxesThird")

    x=var('x')
    f=phyFunction(x*sin(1/x)).graph(-1,1)
    f.plotpoints=1000

    pspict.DrawGraphs(f)

    pspict.axes.Dx=0.5
    pspict.axes.Dy=0.3
    pspict.DrawDefaultAxes()
    pspict.dilatation(3)

    fig.conclude()
    fig.write_the_file()

```

The result is on figure ??

## 6.4 Trigonometric graduation, and other graduations

It is often useful to print the graduation of the axes by  $\frac{\pi}{2}$  instead than one by one. `phystricks` provides the class `AxesUnit` that implements that kind of wishes. The object

```
AxesUnit(pi, "\\pi")
```

represents a unit whose numerical value is `pi` while its `LATEX`'s representation is “ $\pi$ ”. Notice the double backslash and the fact that you must not add the “\$”. In order to take that as “unit” of the  $Ox$  axes, just write

```
axes.axes_unitX=AxesUnit(pi, "\\pi")
```

At this point, the role of `axe.Dx` has to be understood as the step *of the given unit* between each printed number.

Notice that everything regarding the  $X$  axe has to be done via

```
.single_axeX
```

```
from phystricks import *
```

```
def AxesFourth():
```

```

pspict,fig = SinglePicture("AxesFourth")

x=var('x')
f=phyFunction( cos(x) ).graph(-2*pi,2*pi)
pspict.axes.single_aveX.axes_unit=AxesUnit(pi,"\\pi")
pspict.axes.single_aveX.Dx=0.5

pspict.DrawGraphs(f)
pspict.DrawDefaultAxes()
pspict.dilatation(1)
fig.conclude()
fig.write_the_file()

```

The result is on figure ??

Notice that we wrote `pspict.axes.Dx=0.5`, and not `pspict.axes.Dx=1/2`. Remember that Python evaluates `1/2` to zero. The conversion between the decimal value and the fraction is performed by Sage (and is the source of some trouble in my mind ... to be fixed soon).

You can also make the graduation with  $e$  :

```

from phystricks import *

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

    x=var('x')
    epsilon=0.1
    f=phyFunction( ln(x) ).graph(epsilon,2*e)
    pspict.axes.single_aveX.axes_unit=AxesUnit(e,"e")
    pspict.axes.single_aveX.Dx=1

    pspict.DrawGraphs(f)
    pspict.DrawDefaultAxes()
    pspict.dilatation(1)
    fig.conclude()
    fig.write_the_file()

```

The result is on figure ??

If you have your own unit, let's say  $\Omega_k$  whose numerical value is 4.34, you can use it this way:

```

pspict.axes.single_aveX.axes_unit=AxesUnit(4.34,"\\Omega_k")

```

```

from phystricks import *

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

    x=var('x')
    f=phyFunction( x/2+2*sin(3*x) ).graph(-5,5)
    pspict.axes.single_aveY.axes_unit=AxesUnit(2.34,"\\sigma")
    pspict.axes.single_aveY.Dx=0.5

    pspict.DrawGraphs(f)
    pspict.DrawDefaultAxes()
    pspict.dilatation(1)
    fig.conclude()
    fig.write_the_file()

```

The result is on figure ??

## 6.5 Only one axe

You can draw only one axe if you want. This is the class `SingleAxe`. By the way, the axes we saw were a superposition of two single axes.

```

from phystricks import *
def ExSingleAxe():
    pspict,fig = SinglePicture("ExSingleAxe")

    base=Vector(1,2).normalize()
    C=Point(0,0)
    axe=SingleAxe(C,base,-1,1.5)

    pspict.DrawGraphs(axe,axe.bounding_box(pspict))
    pspict.dilatation(1)
    fig.conclude()
    fig.write_the_file()

```

The result is on figure ??.

## 6.6 The grid

Even if, as the axes, the grid exists as an independent object (class `Grid`), we are going to show only the automatic grid. It is drawn by the method

```
pspict.DrawDefaultGrid()
```

It is better to draw the grid before the axes<sup>3</sup> because we do not want the grid to fit even the graduation marks:

```
    pspict.DrawDefaultGrid()
    pspict.DrawDefaultAxes()
```

The examples anticipate on the drawing of functions (section ??). Here is the first:

```
from phystricks import *
def GridOne():
    pspict,fig = SinglePicture("GridOne")

    x=var('x')
    f = phyFunction( x**2-x-1 )
    F=f.graph(-1.5,1.7)

    pspict.DrawGraph(F)

    pspict.DrawDefaultAxes()
    pspict.DrawDefaultGrid()

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

The result is on figure ??

The default grid is the object `pspict.grid`. One custom the grid by acting on that object.

As you see, a grid is composed of a “main” grid (style solid) and a “subgrid” (style dotted). One can separately le parametrise them. For the distances, the parameters are

1. `grid.Dx`. The spacing between two vertical lines in the main grid. (similarly for `grid.Dy`)
2. `grid.num_subX`. The number of subdivision that the subgrid makes between each vertical lines of the main grid. If you don't want vertical subgrid, put zero.

Let us draw the same picture as figure ??, but with different parameters of the grid spacing.

```

from phystricks import *
def GridTwo():
    pspict,fig = SinglePicture("GridTwo")

    x=var('x')
    f = phyFunction( x**2-x-1 )
    F =f.graph(-3,3)

    pspict.DrawGraph(F)

    pspict.grid.Dx = 2
    pspict.grid.Dy = 3
    pspict.grid.num_subX = 0
    pspict.grid.num_subY = 5

    pspict.DrawDefaultAxes()
    pspict.DrawDefaultGrid()

    # The following vertically contracts the figure with a factor 2.
    pspict.dilatation_Y(0.5)

    fig.conclude()
    fig.write_the_file()

```

The result is on figure ??

Notice that, for visual convenience, we dilated the whole figure with a factor 0.5 using the method

```
pspict.dilatation_Y(0.5)
```

The shape and the color of the grid can be customised. The vertical lines of the main grid are “cloned” from `grid.main_vertical`. The latter is the graph of a segment, so that all the customisation of lines are available. For example

```
grid.main_vertical.parameters.color = "red"
```

makes the vertical lines of the main grid red. Similarly the other lines are cloned from the following

```

grid.main_vertical
grid.main_horizontal
grid.sub_vertical
grid.sub_horizontal

```

---

<sup>3</sup>It was not like that in preceding versions

The lines can even be made wavy, but I don't see in which context it can be useful, or even beautiful... Here is an example.

```
from phystricks import *
def GridThree():
    pspict,fig = SinglePicture("GridThree")

    x=var('x')
    f = phyFunction(2*x*sin(x))
    F =f.graph(-pi-0.5,pi+0.5)
    pspict.DrawGraph(F)

    pspict.grid.num_subX = 2
    pspict.grid.num_subY = 3
    pspict.grid.sub_vertical.parameters.color = "green"
    pspict.grid.sub_horizontal.parameters.color = "magenta"
    pspict.grid.main_horizontal.parameters.style = "dashed"

    pspict.DrawDefaultAxes()
    pspict.DrawDefaultGrid()

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

The result is on figure ??

## 7 Drawing functions

### 7.1 General stuff

This is the main part of `phystricks` that defines the class `phyFunction`. An instance of that class is created from an instance of the class

```
sage.symbolic.expression.Expression.
```

Pragmatically, we build the function  $x \mapsto x^2$  by the code

```
var('x')
f = phyFunction(x**2)
```

The double star is the exponentiation in the Sage's language<sup>4</sup>. Let us give a first typical example: the sine function. The code is

```

from phystricks import *
def FunctionFirst():
    pspict,fig = SinglePicture("FunctionFirst")

    x=var('x')
    f = phyFunction( sin(x) )
    mx = -2*pi
    Mx = 2*pi
    F =f.graph(mx,Mx)
    pspict.DrawGraph(F)

    pspict.DrawDefaultAxes()

    fig.conclude()
    fig.write_the_file()

```

The result is given on figure ??

As far as the syntax is concerned,

```
var('x')
```

is a Sage way of declaring that  $x$  will be a variable. Then, we define  $f$  by

```
f = phyFunction( sin(x) )
```

The argument of `phyFunction` has to be an expression of  $x$  that is to be interpreted by Sage. At this moment,  $f$  is the abstract function. The effective object that will be drawn is the one of the class `GraphOfAFunction`. It has to be defined with

```
F = Graph(f,mx,Mx)
```

where  $f$  is the function,  $mx$  is the value of  $x$  where the drawing has to begins and  $Mx$  is the end value of  $x$ .

Of course, we have many methods on the class `GraphOfAFunction` that allows to change the way of drawing. Here are some.

1. `F.style` corresponds to the `linestyle` of `pstricks`. Default is “solid”
2. `F.color` corresponds to the `linecolor` of `pstricks`. Default is blue.
3. `F.plotpoints` gives the number of points to be plotted. It can be useful to increase the number of points if the function presents a strong curvature somewhere, like an asymptote. Think about that if you want to draw a logarithm close to zero.

---

<sup>4</sup>There exists an explanation for that, but I don't know it.

4. `F.wave(1,A)` produces a wavy graph of wavelength 1 and amplitude A

Here is an example of some of these features.

```

from phystricks import *
def FunctionSecond():
    pspict,fig = SinglePicture("FunctionSecond")

    x=var('x')
    f = phyFunction( log(x) )
    mx = 0.1
    Mx = 10
    F = f.graph(mx,Mx)
    G = f.graph(mx,Mx)
    F.parameters.color = "red"
    F.wave(0.3,0.1)
    F.parameters.style = "dashed"
    pspict.DrawGraph(F)
    pspict.DrawGraph(G)

    pspict.DrawDefaultAxes()

    fig.conclude()
    fig.write_the_file()

```

The result is on figure ??

By the way, the `log` command in Sage uses the Euler basis, while the `log` command for `pstricks` uses the basis 10. Here, we use the Euler's basis.

## 7.2 Draw more points

It can happen that a function looks bad because of the sampling (100 points on each graph by default) is insufficient at some places. The solution is to ask to compute more points using `F.plotpoints`.

```

from phystricks import *
def MorePoints():
    fig = GenericFigure("MorePoints")

    x=var('x')
    a=0.2
    f=phyFunction(x*sin(1/x)).graph(-a,a)

    dil=8
    Dx=0.2

```

```

Dy=0.1
ssfig1 = fig.new_subfigure("Not enough points.", "LabelMorePointsA")
pspict1 = ssfig1.new_ypicture("NotEnough")
pspict1.DrawGraph(f)
pspict1.axes.Dx=Dx
pspict1.axes.Dy=Dy
pspict1.DrawDefaultAxes()
pspict1.dilatation(dil)

ssfig2 = fig.new_subfigure("Enough", "LabelMorePointsB")
pspict2 = ssfig2.new_ypicture("Enough")
f.plotpoints=3000
pspict2.DrawGraph(f)
pspict2.axes.Dx=Dx
pspict2.axes.Dy=Dy
pspict2.DrawDefaultAxes()
pspict2.dilatation(dil)

fig.conclude()
fig.write_the_file()

```

The result is on figure ??.

## 7.3 Operations on functions

### 7.3.1 Derivative

One obtain the derivative of a function by

```
f.derivative()
```

As an example, the function  $x \cos(x)$  and its derivative.

```

from phystricks import *
def FunctionThird():
    pspict,fig = SinglePicture("FunctionThird")

    x=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.dilatation(0.7)

fig.conclude()
fig.write_the_file()

```

The result is on figure ??

### 7.3.2 Put a point on the graph

If  $f$  is of the class `phyFunction`, we evaluate  $f$  at the point  $x$  using

```
f.eval(x)
```

For example,

```

var('x')
f = phyFunction( x**2 )
print f.eval(2)

```

returns 4.

If  $f$  is an instance of `phyFunction`, the line

```
f.get_point(x)
```

returns the point at the coordinate  $(x, f(x))$

The following code draws the function  $x \mapsto x \sin(x)$  and puts a point on the integer values of  $x$ .

```

from phystricks import *
def FunctionFour():
    pspict,fig = SinglePicture("FunctionFour")

    x=var('x')
    f = phyFunction( x*sin(x) )
    mx = -5
    Mx = 5
    F = f.graph(mx,Mx)

```

```

points = []
for i in range(mx,Mx) :
    points.append(f.get_point(i))

pspict.DrawGraph(F)
for i in range(0,len(points)):
    points[i].put_mark(0.3,points[i].advised_mark_angle,"$P_{%s}$"%str(i))
    pspict.DrawGraph(points[i])

pspict.DrawDefaultAxes()

fig.conclude()
fig.write_the_file()

```

The result is on figure ??

One can put points regularly spaced points with respect to the arc length. For this, we use the method

```
f.get_regular_points(mx,Mx,d1)
```

That returns a list of points that are on the graph of  $f$  with  $x$  coordinates between  $mx$  and  $Mx$ . The distance between two of these points is given by  $d1$ . As an example:

```

from phystricks import *
def FunctionFive():
    pspict,fig = SinglePicture("FunctionFive")

    x=var('x')
    f = phyFunction( x*sin(x) )
    mx = -5
    Mx = 5
    F = f.graph(mx,Mx)

    points = f.get_regular_points(mx,Mx,1.5)

    pspict.DrawGraph(F)
    for i in range(0,len(points)):
        P = points[i]
        P.put_mark(0.3,P.advised_mark_angle,"$P_{%s}$"%str(i))
        pspict.DrawGraph(P)

    pspict.DrawDefaultAxes()

```

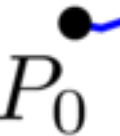


Figure 33: Some points spaced with respect to the arc length.