On the usage of the Lout @Graph package

Eric Boucharé

28 June, 2020

Abstract

This report introduces a modified Lout @Graph package with slighly enhanced tick and grid features, ensuring compatibility with the standard package provided with Lout. It explores, as well, through examples, how to draw complex graphes using an outer program. Many of the techniques presented here are already usable on the standard package.

Contents

1. Package installation and usage

The modified package uses the same files as the standard package : `graph`, `graphf`, `graph.etc` and `graph.lpg`. They should be copied in the `include` directory of Lout as a drop replacement for the standard package or in the current direct as it is in the current example, or in a custom directory (though you will have to modify some paths in the files to reflect the location).

Depending on the location, to use the @Graph package, you will have to include the graph definitions from the system directory or a local directory.

The display of a graph is a matter of including

```
@Graph
  <graph option parameters>
{
  @Data <data display options> {x1 y1 x2 y2 ...}
}
```

2. Styles, grid, ticks and labels

Styles are setup withe the *style* option. Supported styles include *frame*, *axes*, *none*, *xygrid*, *xgrid*, *ygrid*.

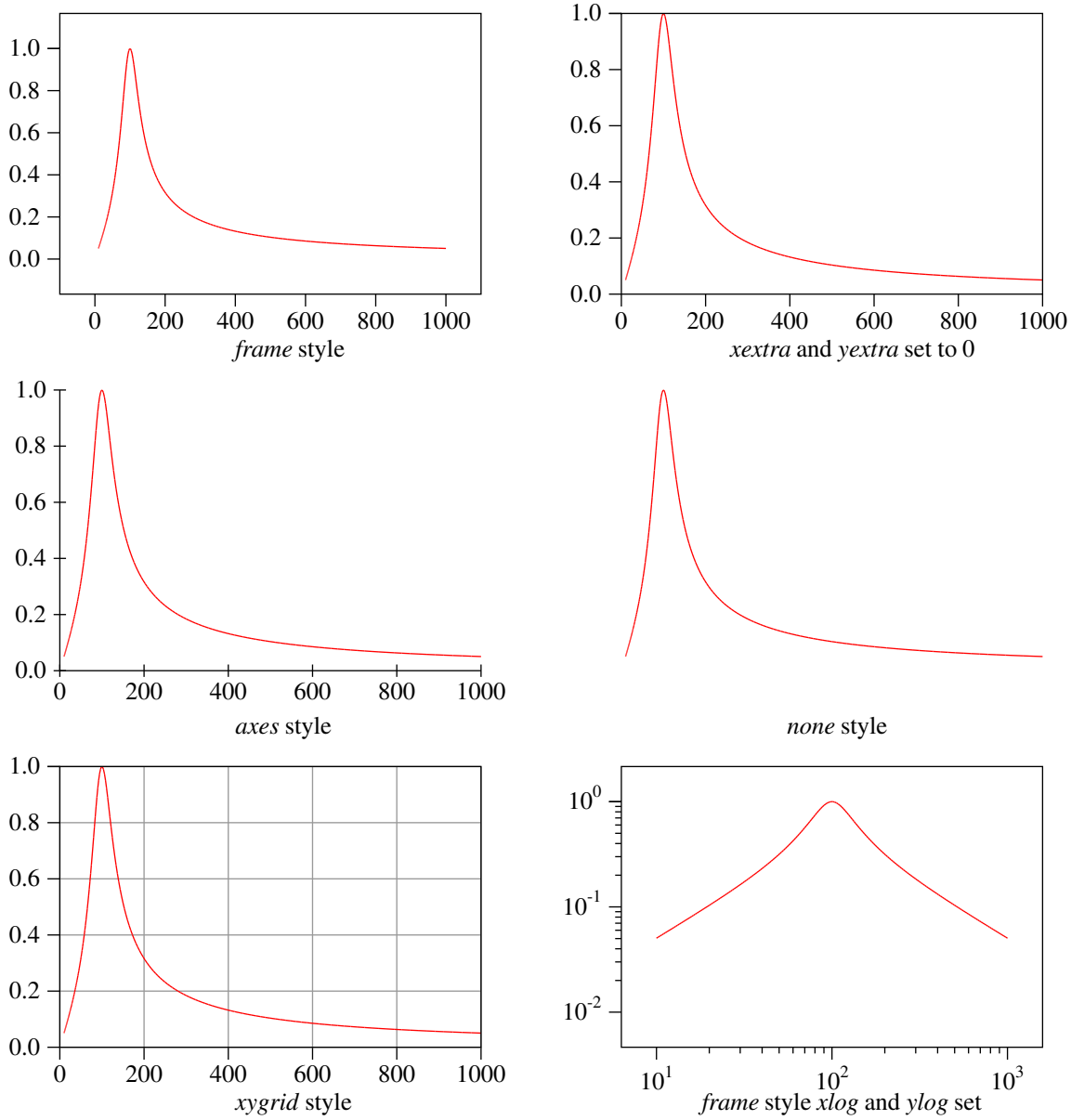
Frame style enables the drawing of a frame around the graph, with ticks and labels. Two option parameters, *xextra* and *yextra*, control the size of a margin around the graph. It defaults to 0.5c.

Axes style enables the drawing of axes with ticks and labels defined by option values *xorigin* and *yorigin*. No

frame is drawn.

xgrid, *ygrid* and *xygrid* styles enable the drawing of a grid vertically, horitontally or in both directions. A frame with ticks and labels is, as well, drawn around the graph.

none style does not draw anything: no frame, no axes, no grid, no ticks, no labels.



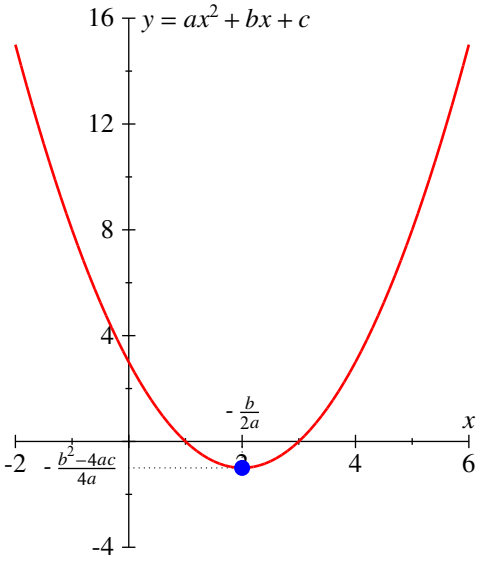
The above graphs are defined with the code (change commented options to draw the different graphs).

```
@Graph
  style{frame}
# style{frame} xextra{0c} yextra{0c}
# style{axes} xorigin{0} yorigin{0}
# style{none}
# style{xygrid}
# xlog{10} ylog{10}
{
  @Data pairs{solid} color{red} {
    @Include{"data/data1"}
  }
}
```

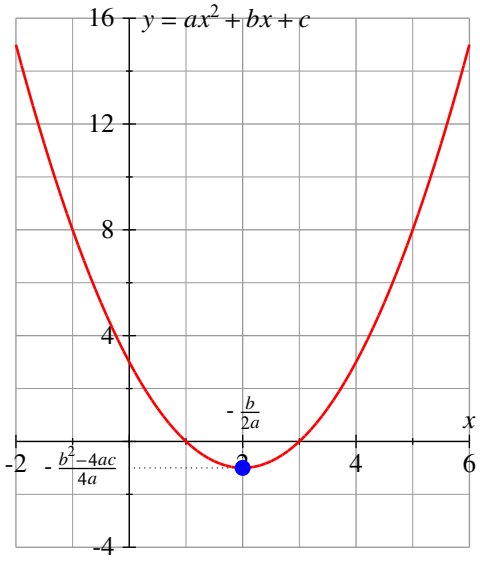
The @Graph symbol has also *width* and *height* options which allows you to set the size of the graph, actually, the area used to draw the curve. Lets's explore now the interaction between the ticks, labels and the grid with the following example:

```
@Graph
  style{axes} width{6c} height{7c}
  grid{no} label{yes}        # default values
  objects {
    @N at{6 0} margin{0.5f} @M{x}
    @E at{0 16} margin{0.5f} @M{y=a x sup 2 + b x +c}
    @N at{2 0} @M{"-" ` b over 2a}
    @W at{0 -1} margin{0.5f} @M{"-" ` {b sup 2 - 4 a c} over {4a}}
  }
  xorigin{0.0} yorigin{0.0}
  xticksep{2} xnsubtick{2}   # xticks{-2@ -1 0@ 1 2 () 3 4@ 5 6@}
  yticksep{4} ynsubtick{2}
{
  @Data pairs{dotted} linewidth{0.5p} { 0 -1 2 -1 2 0 }
  @Data color{red} pairs{solid} linewidth{1p} {
    xloop from {-2} to {6} by { 0.1 } do {
      x {{x - 2} * {x - 2} - 1}
    }
  }
  @Data color{blue} points{filledcircle} symbolsize{0.3f} { 2 -1 }
}
```

Display of a grid with ticks, subticks



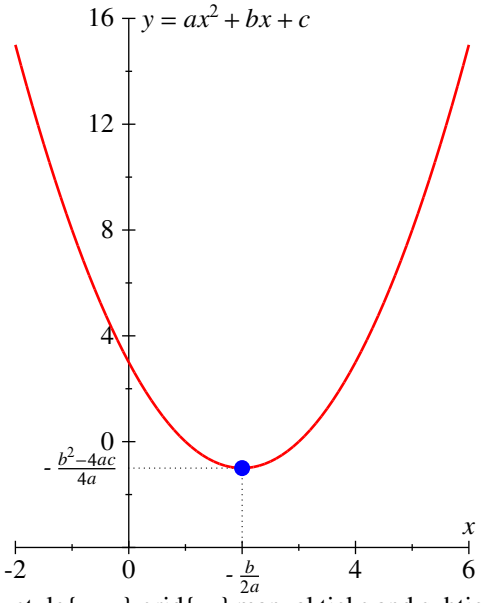
style{*axes*} grid{*no*}



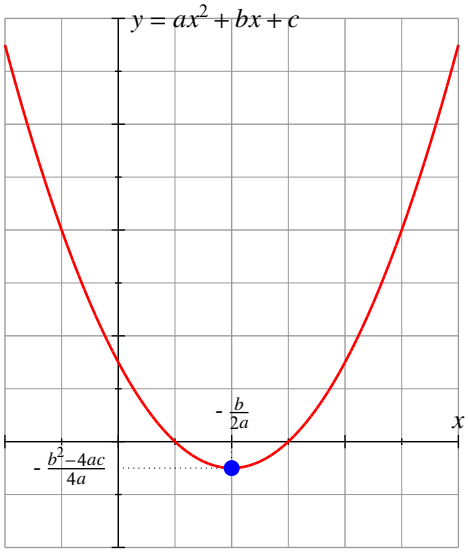
style{*axes*} grid{*yes*}

Note the effect of `xnsubtick{2}` and `ynsubtick{2}` which divides each interval between two subsequent ticks to put a subtick (smaller than a tick, and without label). When a grid is drawn, it results in the drawing of a subgrid.

Display of a grid with ticks, subticks



style{*axes*} grid{*no*} manual ticks and subticks



style{*axes*} grid{*yes*} label{*no*}

Ticks can be defined as usual with *xmin*, *xmax*, *xticksep* or nothing for automatic ticks or *xticks* for manual ticks. a tick is associated with a label displayed wth the tick.

The label is determined automatically according to the tick position on the axis. When defining ticks manually, for example

```
xticks{-2@ -1 0@ 1 2 (data1) 3 4@ 5 6@}
```

'`-2@`' identifies the tick at position *-2*, the `@` indicating that the value -2 is to be displayed. '`2 (data1)`' defines a tick at position *2* and with label *data1*. Using '`2 ()`' is a way to display a tick without any label, allowing us, in this case, to replace the label '2' with the expression $-\frac{b}{2a}$.

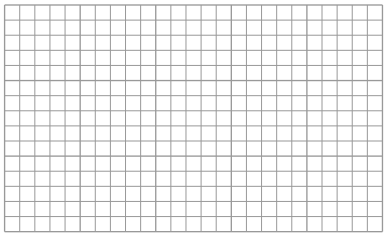
'`-1`' defines a subtick which appears smaller on the graph, and without any label. Subticks can be defined automatically using the *xnsubtick* which sets how many subranges would there be between two subsequent ticks.

Finally a *label* option, with possible values *yes*, *no*, *x* or *y*, controls globally if labels are displayed along the axis.

The *grid* option can take one the following values: *yes*, *no* (the default), *x* or *y*. It uses the ticks and subticks definitions allowing to have a coarse and a thiner grid. Hence, the graph

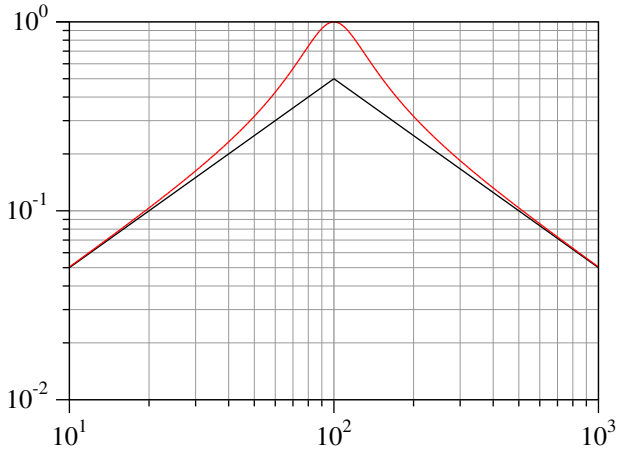
```
@Graph
  style{none} width{5c} height{3c} grid{yes}
  xmin{0} xmax{5} xticksep{1} xnsubtick{5}
  ymin{0} ymax{3} yticksep{1} ynsubtick{5}
{}
```

can just display a 5x3 cm 1 cm-based grid with a 2 mm subgrid.



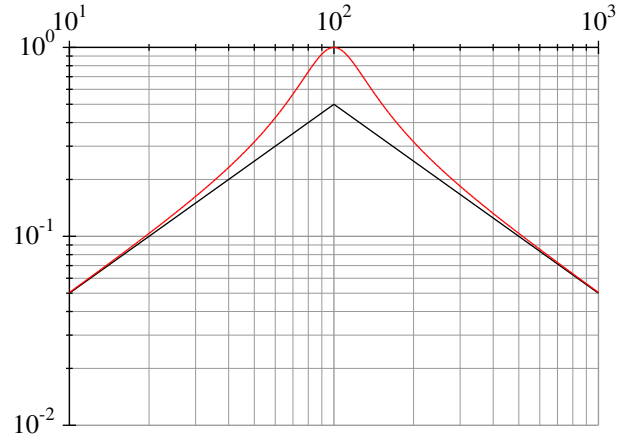
Note that when the *grid* option is used, it bypass the grid styles. It is worth using the *grid* option instead of the grid styles.

An example with a log scale, now.




```
@Graph
  width{7c} height{5c}
  style{frame} xextra{0c} yextra{0c}
  grid{yes}
  xlog{10} ylog{10}
{
  @Data pairs{solid} color{black}{
    10 0.05 100 0.5 1000 0.05
  }
  @Data pairs{solid} color{red} {
    @Include{"data/data1"}
  }
}
```

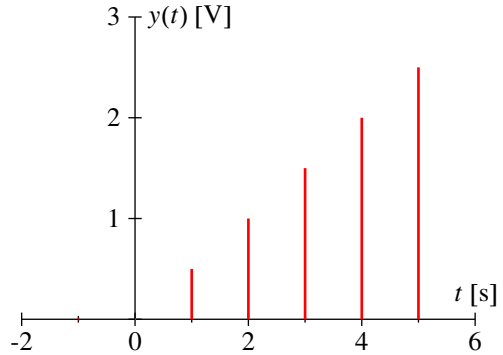
A subgrid is automatically drawn between the log scale ticks. the subgrid cannot be changed. One has to note that, for such a graph, the data have to be stored in a file that is generated by an external tool (computing environment, scripting languages, …) and included. The data in the file must be put in 'x y' tuples.



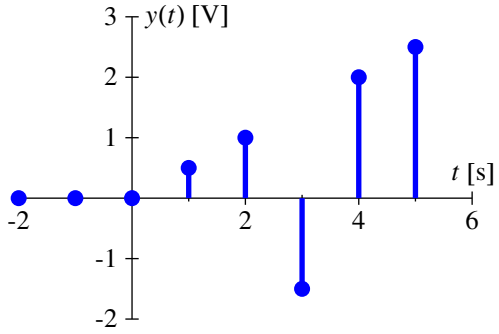

```
@Graph width{7c} height{5c}
  style{axes} xorigin{10} yorigin{1}
  grid{yes}
  xlog{10} ylog{10}
{
  @Data pairs{solid} color{black}{
    10 0.05 100 0.5 1000 0.05
  }
  @Data pairs{solid} color{red} {
    @Include{"data/data1"}
  }
}
```

Here, as well, a noteworthy feature of the `axes` style: when the x axis is at the top of the graph, the labels are displayed above the axis, else they appear below the axis.

3. Comb data visualization

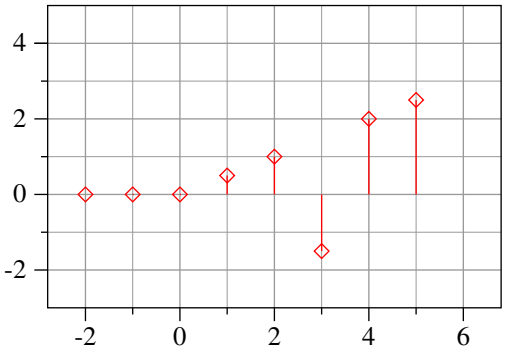



```
@Graph
  style{axes} xorigin{0} yorigin{0}
  xticksep{2} xnsubtick{2}
  objects{
    @N at{6 0} margin{0.5f} @M{t`[@R{s}]}
    @E at{0 3} margin{0.5f} @M{y(t)`[@R{V}]}
  }
  color{red}
{
  @Data pairs{comb} linewidth{1p} {
    -2 0 -1 0 0 0 1 0.5 2 1 3 1.5 4 2 5 2.5
  }
}
```



The line drawn with each point is referenced to the position of the x axis (defined by the *yorigin* option), or the axis defined by the `ymin` value defined or calculated if `yorigin` is not set.

The *points* option allow to put a symbol associated to the point, as usual.

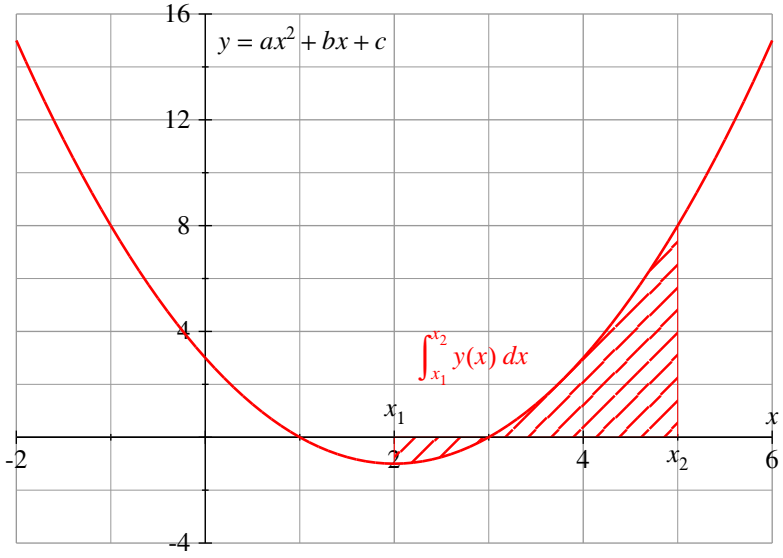


Even though the *axes* style is not used.

```
@Graph
  style{frame} grid{yes}
  xorigin{0} yorigin{0}
  xticksep{2} xnsubtick{2}
  yticksep{2} ynsubtick{2}
{
  @Data pairs{comb} points{diamond}
    symbolsize{0.3f} color{red}{
    -2 0 -1 0 0 0 1 0.5 2 1 3 -1.5 4 2 5 2.5
  }
}
```

4. On the use of textures

An example on using a texture to emphasize an area

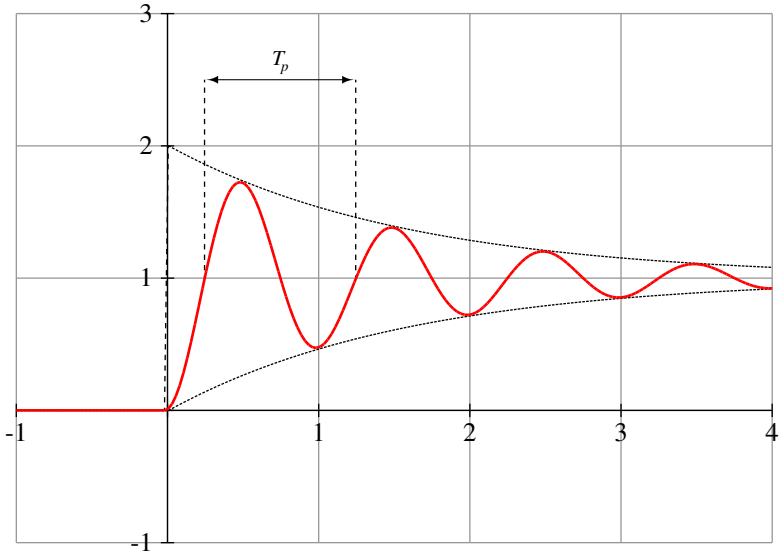



```
@Graph
  style{axes} width{10c} height{7c}
  grid{yes}
  xorigin{0.0} yorigin{0.0}
  xticksep{2} xnsubtick{2}
  yticksep{4} ynsubtick{2}
  objects{
    @N at{6 0} margin{0.5f} @M{x}
```

```
    @E at{0 15} margin{0.5f} @M{y=a x sup 2 + b x +c}
    @N at{2 0} margin{0.5f} @M{x sub 1}
    @S at{5 0} margin{0.5f} @M{x sub 2}
    @E at{3.5 3} red @Color @Math{int from{x sub 1} to{x sub 2} y(x) ` dx}
  }
{
  @Data color{red} pairs{solid} linewidth{0.2p}
    texture{ striped angle { 45d } gap{5p}} paint{yes}{
    xloop from {2} to {5} by { 0.1 } do {
      x {{x - 2} * {x - 2} - 1}
    }
    # close the surface
    5 0 2 0 2 -1
  }
  # Redraw the x axis (nicer)
  @Data pairs{solid} {-2 0 6 0}
  @Data color{red} pairs{solid} linewidth{1p} {
    xloop from {-2} to {6} by { 0.1 } do {
      x {{x - 2} * {x - 2} - 1}
    }
  }
}
```

We draw the curve two times. The first is to draw the textured area. Note that we terminate by adding points to close the surface. The second time, we draw the curve on the full x axis.

5. Using @Diag to add information onto a graph




```
@Diag{
  @Graph
    style{axes} width{10c} height{7c}
    grid{yes}
    xorigin{0.0} yorigin{0.0}
    objects {
      @CTR at{0.245 2.5} {A::@Node margin{0.01f} outlinestyle{noline}}
```

```
      @CTR at{1.245 2.5} {B::@Node margin{0.01f} outlinestyle{noline}}
    }
  {
    @Data pairs{dashed} {0.245 1 0.245 2.5}
    @Data pairs{dashed} {1.245 1 1.245 2.5}
    @Data pairs{dashed} {
      # include {"data/data3"}
    }
    @Data pairs{dashed} {
      # include {"data/data4"}
    }
    @Data color{red} pairs{solid} linewidth{1p} {
      include {"data/data2"}
    }
  }
  @Link arrow{both} arrowstyle{curvedsolid} backarrowstyle{curvedsolid}
    ylabel{@M{T tsub p}} from{A} to{B}
}
```

Arrows can't be added through the @Graph symbol, so we use the @Diag symbol, define the two points *A* and *B*, and use them to draw the arrow between thes two points.

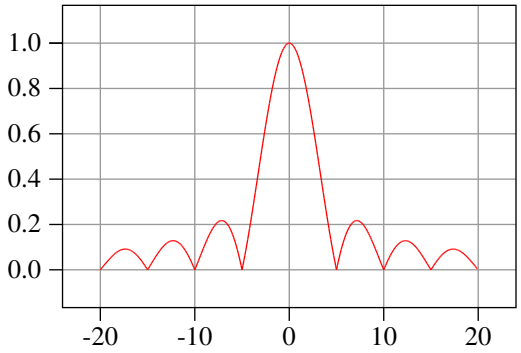
6. Defining a new function

One can define new functions at the begining of the file

```
import @Graph @Data
def sinc
  precedence 40
  right x
{
  if cond{ abs{x} > 0.001 } then { sin{x * 180 / pi} / x} else { 1 }
}
```

Then graph the following function

```
@Graph grid{yes} {
  @Data pairs{solid} color{red} {
    xloop from {-20} to {20} by { 0.1 } do
    { x {abs{sinc{x*pi/5}}} }
  }
}
```



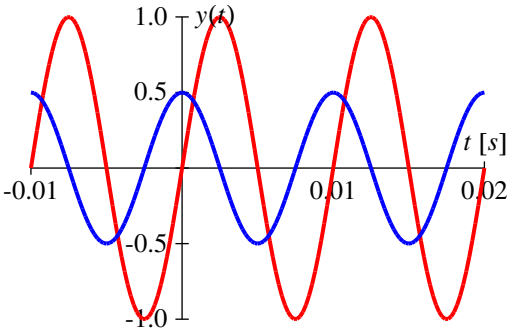
7. Using the *retro* calculator to generate plots

retro (`https://github.com/ebcfr/retro/`) is a small matrix calculator that can handle real and complex calculation. The program must be in the system research PATH variable.

Add the following definition at the begining of the file or in your *mydefs* file.

```
import @BasicSetup
def @Calc
  right x
{
  def @Filter {retro -q < @FilterIn > @FilterOut}
  x
}
```

Some quick examples




```
@Calc {
  load lout
  T=0.01;f=1/T;t=-T:T/50:2*T;x=sin(2*pi*f*t);
  xlabel="t [s]";
  graph(grid="no",style="axes")
  graphobj("N",[2*T,0],"@M{t ` [s]}")
  graphobj("E",[0,1],"@M{y(t)}")
  begingraph()
  graph2d(t,x)
  graph2d(t,0.5*cos(2*pi*f*t),color="blue")
  endgraph()
}
```

The sequence used by functions `graph`, `begingraph`, `graph2d`, and `endgraph` handle the generation of the Lout code for drawing the graph. The `graphobj` function is used to place objects on the graph.

Here's a decription of the API defined in the `lout.e` file.

`graph(<parameters>)` starts a graph: there may be multiple plots on the same graph. Extra named parameters are defined after the option parameters of the Lout @Graph symbol.

- *width*, *height* [string] : size of the graph, ex: `width="8c"`
- *style* [string] : one of "frame", "axes", … ex: `style="axes"`
- *xextra*, *yextra* [string] : the extra space around the graph,
- *xorigin*, *yorigin* [number] : the location of the origin of axes,
- *xlog*, *ylog* [0/1] : boolean switch. Only 10-based logscales supported,
- *xticks*, *xticksep*, *xnsubtick*, *yticks*, *yticksep*, *ynsubtick*: provide tick and subtick information,
- *color* [string] : global setting for color,
- *xlabel*, *ylabel* [string] : labels for the x and y axes.

`graphobj(dir,point,obj)` Place the Lout object *obj* described by a string at the location *point* using the direction *dir*.

`begingraph()` begins the plot description

`endgraph()` ends the plot description

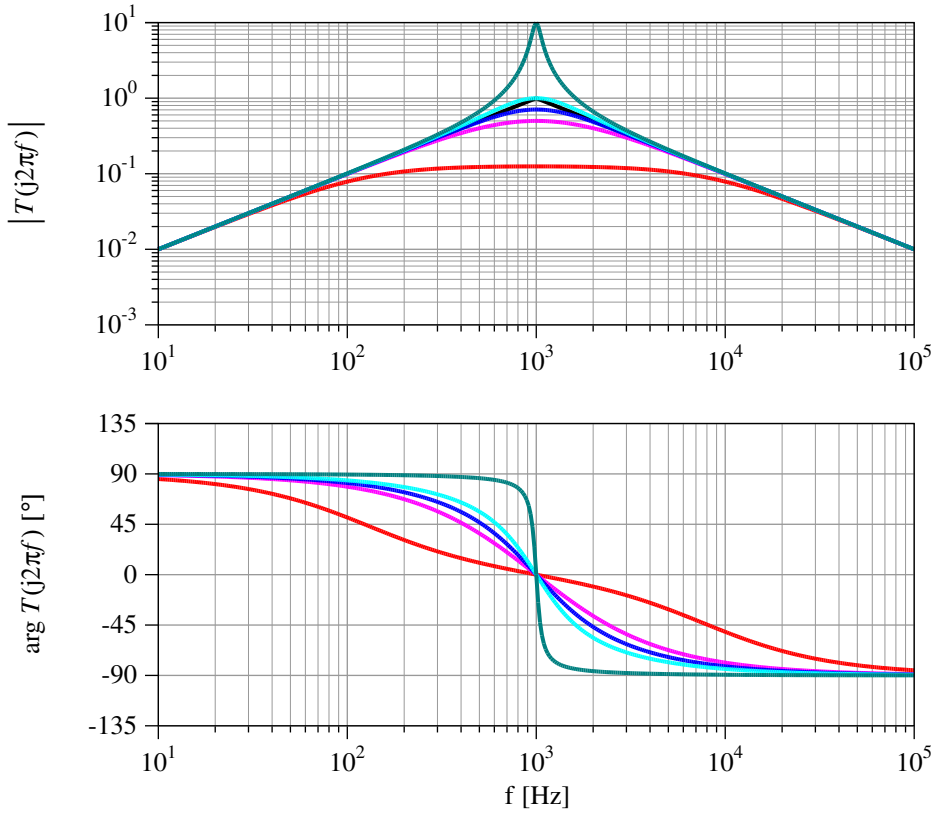
`graph2d(x,[y,[<parameters>]])` plots a curve using real vectors or matrix *x* and *y*, or a complex vector *x* (then plots real and imaginary part of *x*). If *x* and *y* are matrices, each line of the matrices define a curve to be displayed. Extra named parameters:

- *pairs*, *linewidth* [string] : Lout pairs and linewidth options, ex: `linewidth="1p"`,
- *points*, *symbolsize* [string] : Lout points and symbolsize options, ex: `symbolsize="0.3f"`
- *color* [string] or [real vector] : set the color of maps a color defined by the index in the vector to each curve.

Another example. Graphing the Bode Diagrams $|T(j\omega)|$ and arg $T(j\omega)$ where $T(j\omega)$ is the frequency response of a second order band-pass system for which the transfer function is

$$T(s) = \frac{T_i \frac{s}{\omega_0}}{1 + 2m \frac{s}{\omega_0} + \frac{s^2}{\omega_0^2}}$$

and for différent values of the damping factor, is a quite simple program using the vector language of *retro*.




```
@Calc {
  load lout
  f=10^(1:0.01:5);
  Ti=1;f0=1k;m=[4,1,1/sqrt(2),0.5,0.05]';j=1i;
  T=Ti*j*f/f0/(1+2*m*j*f/f0+(j*f/f0)^2);
  ylabel="@M{abs{T(j 2 pi f)}}";
  graph(width="10c",grid="yes",xlog="yes",ylog="yes",ylabel=ylabel)
  begingraph()
```

```
  graph2d([10,1k,100k],[1e-2,1,1e-2],color="black")
  graph2d(f,abs(T),color=1+(1:length(m)))
  endgraph()
  "/2f"
  xlabel="f [Hz]";
  ylabel="@M{arg ` T(j 2 pi f)``[degree]}";
  graph(width="10c",grid="yes",xlog="yes", ..
    yticks="-135@ -90@ -45@ 0@ 45@ 90@ 135@",xlabel=xlabel,ylabel=ylabel)
  begingraph()
  graph2d(f,arg(T)/pi*180,color=1+(1:length(m)))
  endgraph()
}
```

One can note there are two graphes with, between them, the vertical alignment symbol `/2f` which set the second graph at a distance of 2 times the current font height beneath the first one.

For what it's worth, the *retro* calculator can also be used to inline calculations and render the dynamic content.

$$\begin{pmatrix} 1 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{pmatrix}^{-1} = \begin{pmatrix} 1 & -2 & 1 \\ -2 & 1.33 & -0.333 \\ 1 & 0.333 & -0.333 \end{pmatrix}$$

$$\begin{pmatrix} 1.00 & 1.00 & 2.00 \\ 3.00 & 4.00 & 5.00 \\ 6.00 & 7.00 & 8.00 \end{pmatrix}^{-1} = \begin{pmatrix} 1.00 & -2.00 & 1.00 \\ -2.00 & 1.33 & -0.33 \\ 1.00 & 0.33 & -0.33 \end{pmatrix}$$

Here's the code

```
@Math { @Calc {
  function lmat(A)
    "ppmatrix{"
    for i=1 to rows(A)
      " row "
      for j=1 to cols(A)
        " rcol {",A[i,j],"}"
      end
    end
    "}"
    return A;
  endfunction

  A=[1,1,2;3,4,5;6,7,8];
  format("STD",[0,3]);
  lmat(A);" sup {-1} ^= `" lmat(inv(A));
  "/0.5f"
  format("FIXED",[0,2]);
  lmat(A);" sup {-1}  ^= `" lmat(inv(A));
}}
```

8. Using a scripting language to generate plot points

The same principle as above can be used to make a helper program to do calculations and generate the Lout code to draw the graphs. Here's an example for the *python* scripting language.

Define a @Python symbol

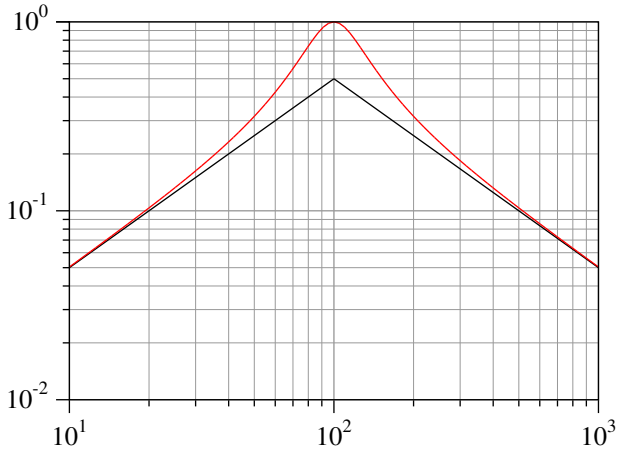
```
import @BasicSetup
def @Python
    right x
{
    def @Filter {python < @FilterIn > @FilterOut}
    x
}
```

Calling @Python:

```
@I @Python{
# my python code
print("Hello World!")
for i in range(0,5):
    print(i)
}
```

which produces: *Hello World! 0 1 2 3 4*

Using python to plot a graph:

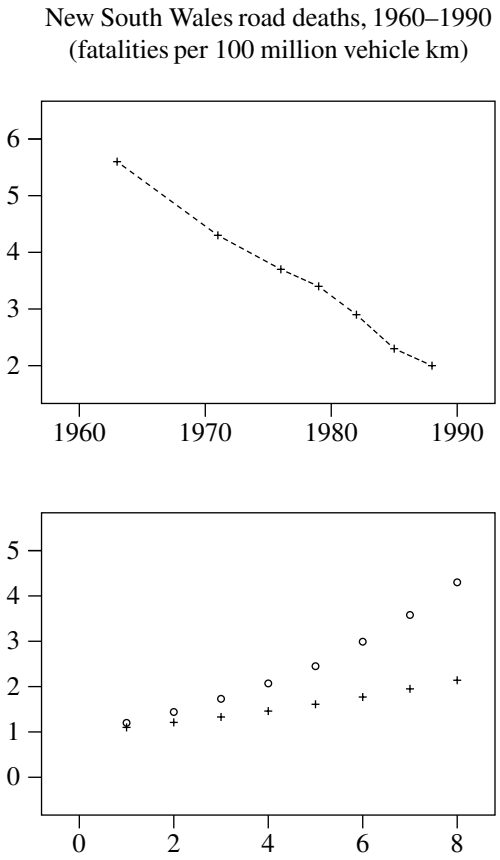


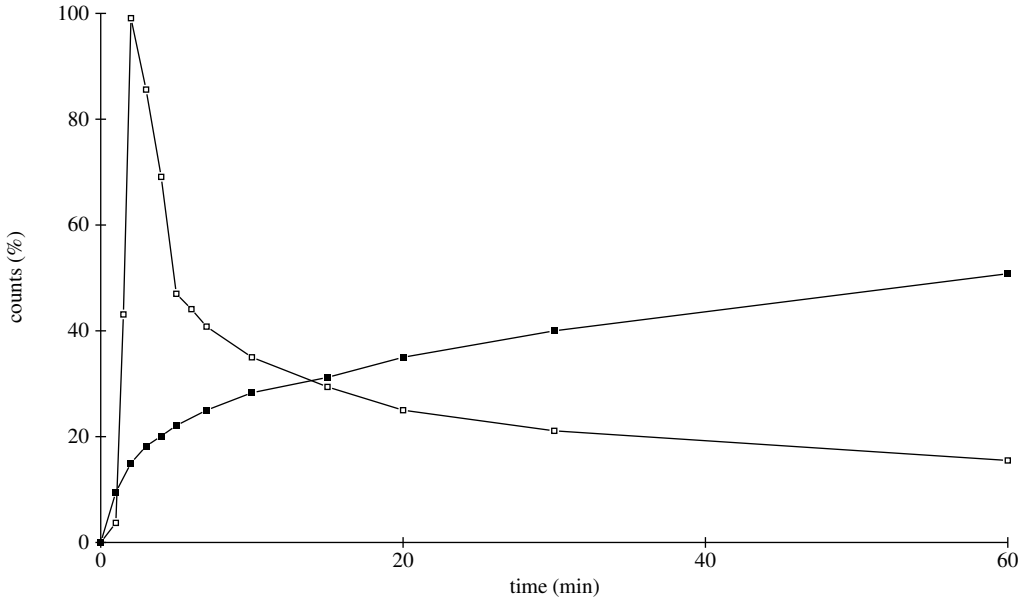

```
@Graph
  width{7c} height{5c}
  style{frame} xextra{0c} yextra{0c}
  grid{yes} label{yes}
  xlog{10} ylog{10}
{
  @Data pairs{solid} color{black}{
    10 0.05 100 0.5 1000 0.05
  }
  @Data pairs{solid} color{red} {
    @Python {
from numpy import *
j=complex(0,1)
# define the frequency vector
f=10**linspace(1,3,51)
m=0.25;f0=100
# Calculate the magnitude of the
# transfer function
T=abs(1/(1+j/(2*m)*(f/f0-f0/f)))
# print the points to stdout
for i in range(f.size):
  print("%g %g " % (f[i],T[i]))
    }
  }
}
```

Python executes the specified code and returns the result on its standard output (the calculated points) back to Lout.

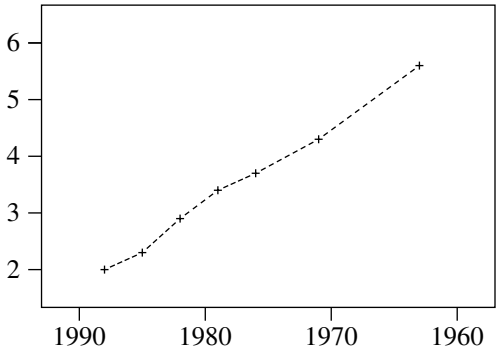
9. Lout User's Guide examples

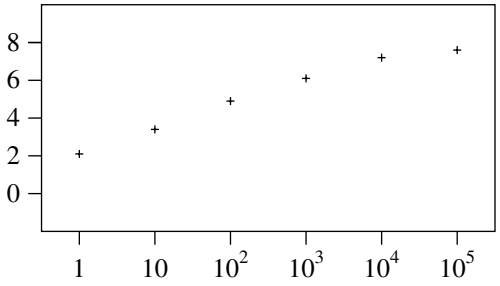
For memory, the figures used in the User's Guide rendered by the modified module are unchanged.

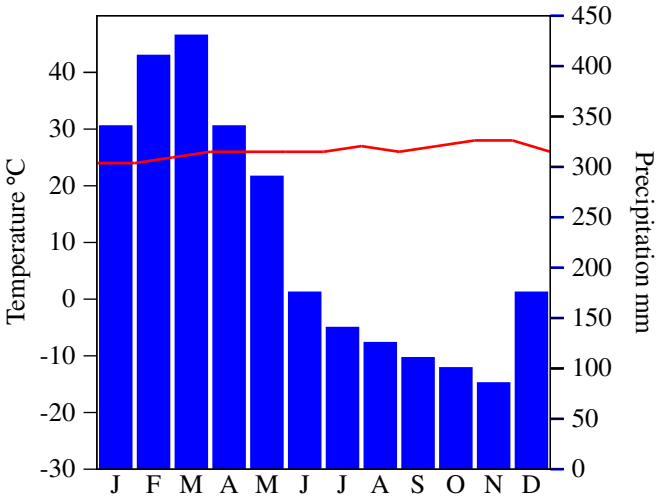




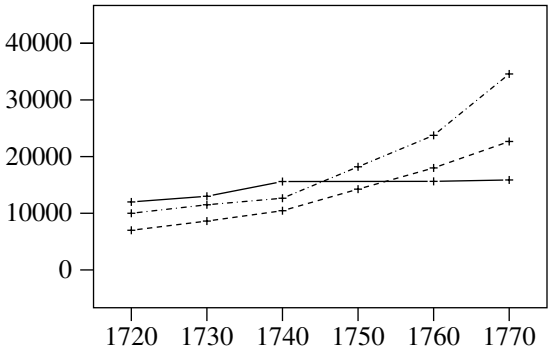
New South Wales road deaths, 1960–1990
(fatalities per 100 million vehicle km)



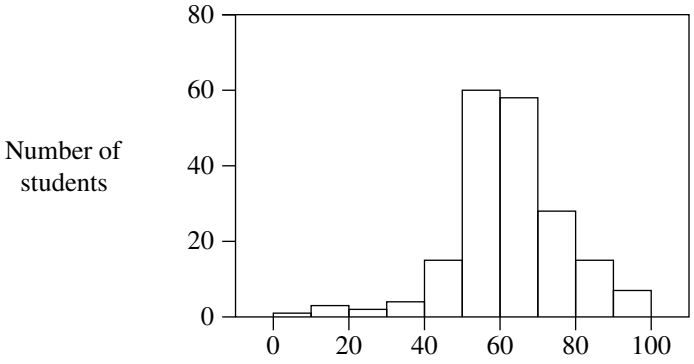




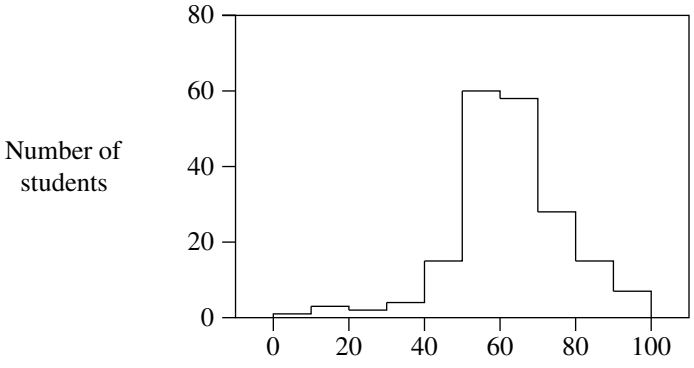
Estimated population of Boston, New York, and Philadelphia



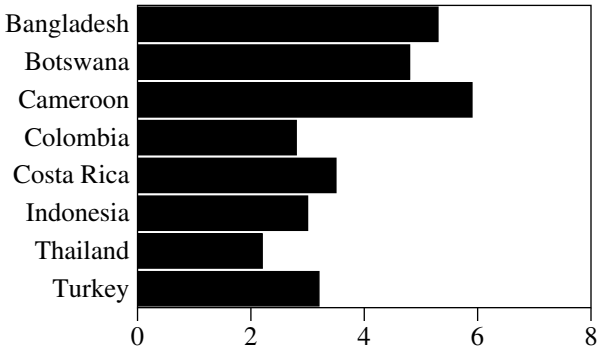
Computer Science 3 Results (1993)

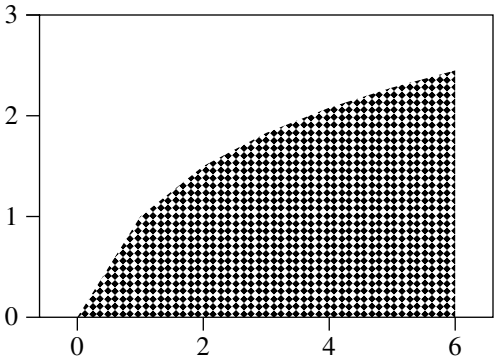


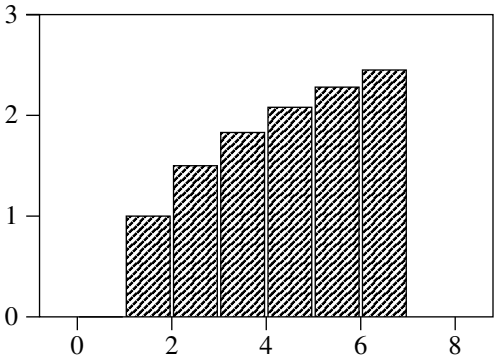
Computer Science 3 Results (1993)



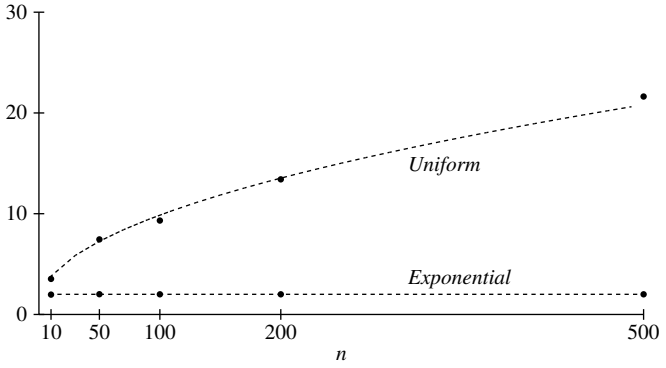
Fertility rates in some developing countries

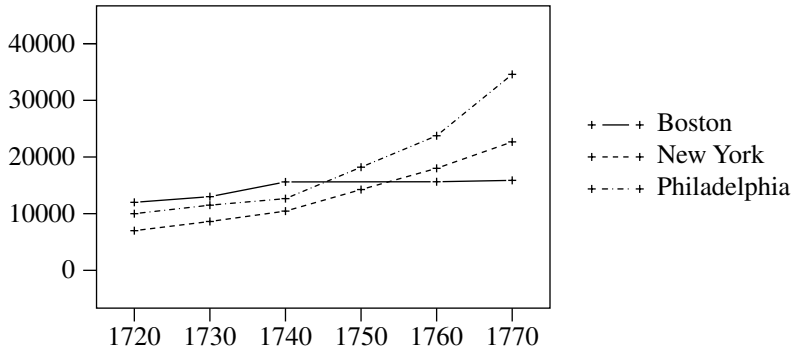






Right shell nodes




40000
30000
20000
10000
0
1720
1730
1740
1750
1760
1770
Boston
New York
Philadelphia