In the name of Allah,

Most Gracious, Most Merciful.

Article Title :

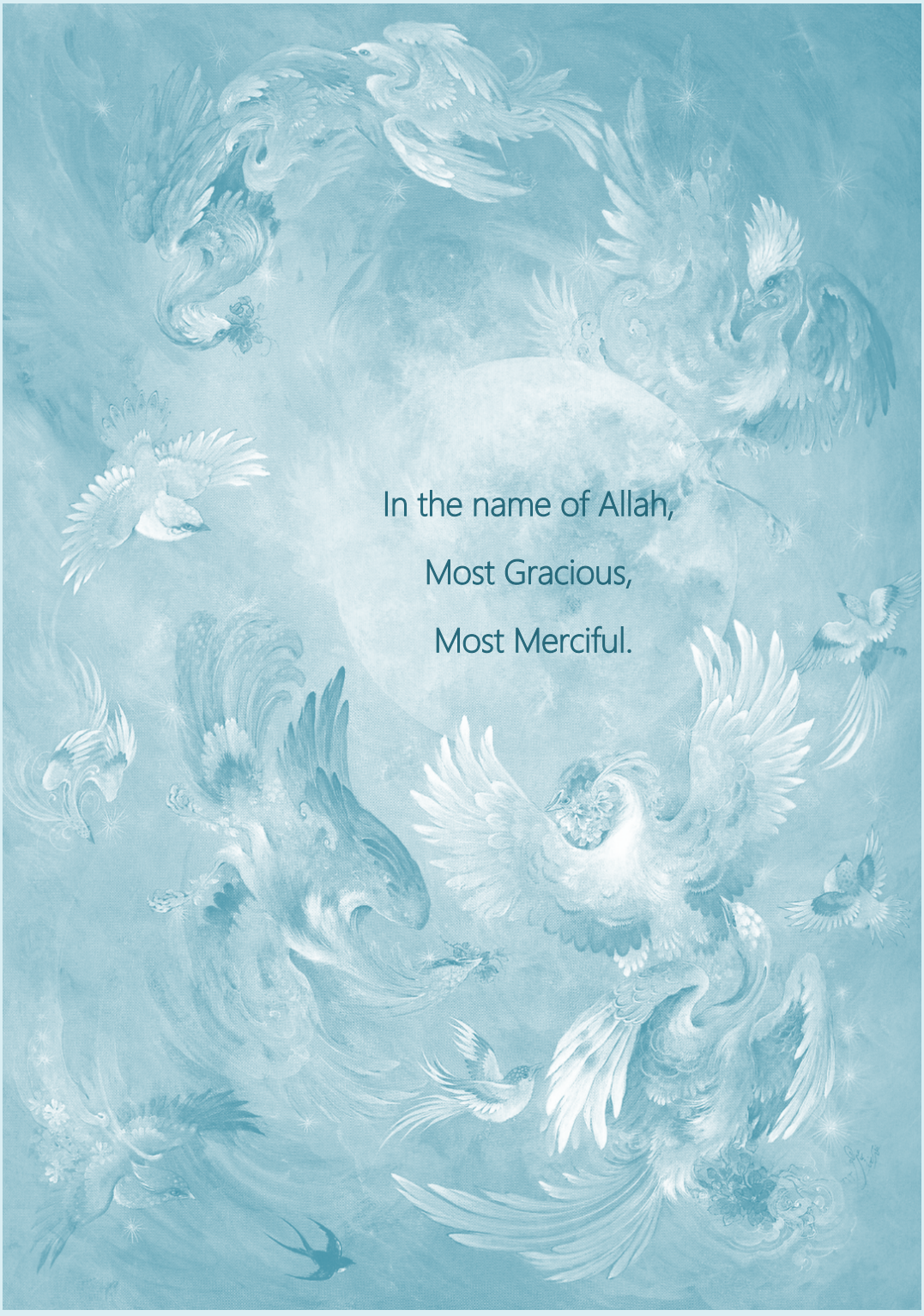
Miniatoric Infinity Plot And It,s Applications

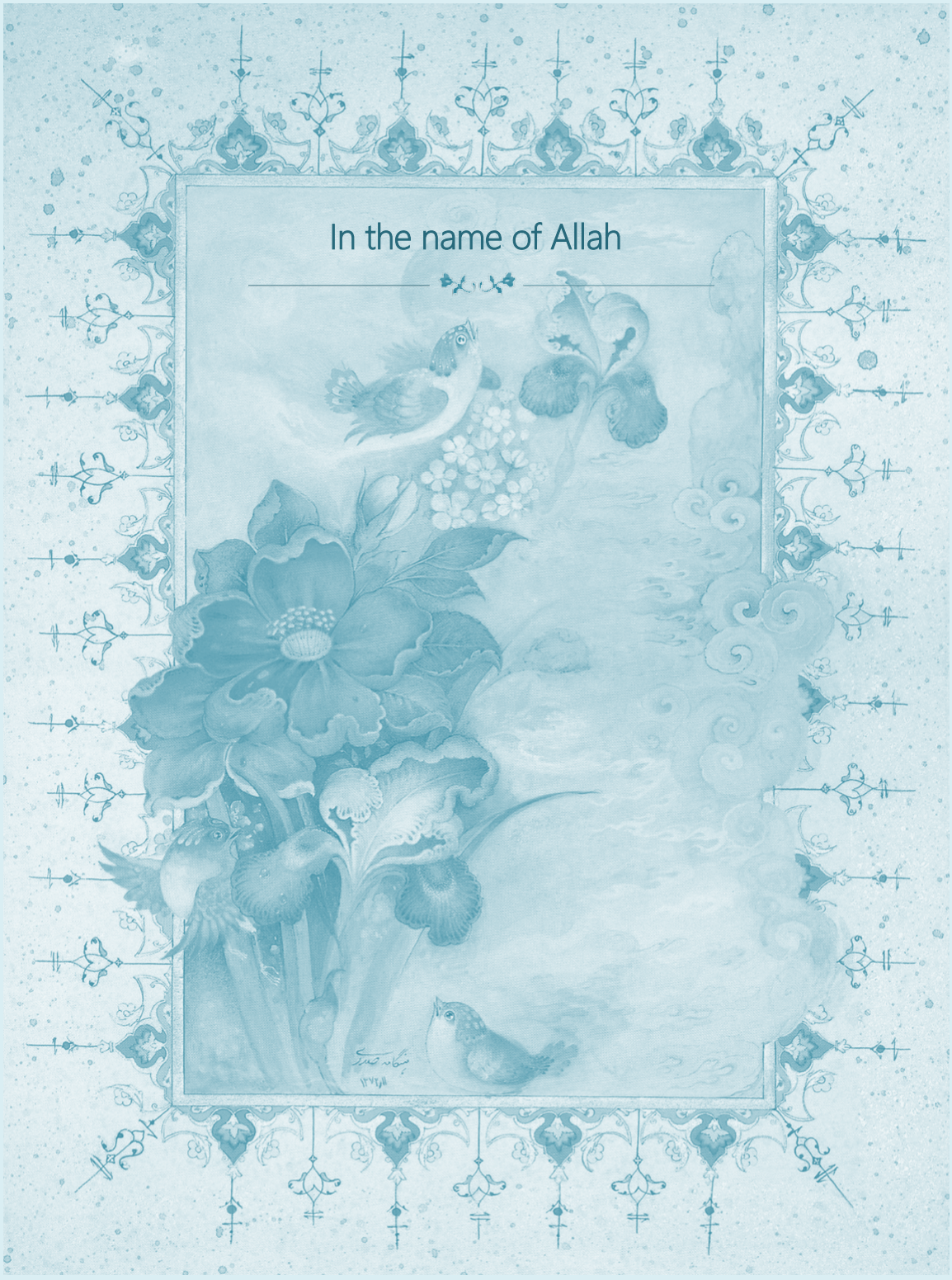
Digest Description Of Miniatoric Infinity Plot Concept :

- If either of the range end points of the horizontal range contains ±infinity, an infinity plot is generated.
- An infinity plot is obtained by transforming -infinity .. +infinity to $-\frac{\pi}{2}..\frac{\pi}{2}$ by a transformation that approximates arctan. This is a nice way of getting the entire picture of f(x) on the display.
- Such a graph, although distorted near x = -infinity and +infinity, contains a lot of information about the features of f(x).

The Article Is A Gift
From The GOD
To All The Believers In GOD

Do Not Forget The Ordained Prayer To GOD
Because Of This Great Manifest Signs Of The GOD.


In the name of Allah,
Most Gracious,
Most Merciful.


In the name of Allah

In the name of Allah


In the name of Allah

INFINITY PLOT APPLICATIONS

- ***NewtonRGhofrani Method to find roots***

*The function **NewtonR** computes a root of a given function ,starting from an initial guess of the root, and using the Newton-Raphson method. This method is widely used due to its convergence speed. However, the method requires the derivative of the function, and may fail to converge, if the initial guess is bad.*

Syntax

root = NewtonR(ftrans,guess,dftransdx,maxit,show)

computes a **root** of the transformed function **ftrans** with the following formula

math.atan(f(math.tan(x)))

where **f** is the main function, starting from an initial guess, **root=guess**.The arguments **dftransdx, eps , maxit**, and **show** are optional. **dftransdx** is the name of the function defining the derivative of **ftrans** ; it can be set to "auto" (the default), which means that no derivative is supplied (in this case, the derivative of the function will be computed numerically). **eps** defines the desired accuracy (default: **Epsilon**, i.e. 1.12×10^{-16}); it can be set to "**auto**", which is equivalent to the default accuracy. **maxit** defines the maximum number of iterations (default: 100). **show** is a boolean argument that controls whether progress of the iterative will be displayed or not (default: **false**); if set to **true**, the function value, f(root), at each iteration will be displayed.

Example

The example program **XNRGh** uses the function **NewtonR** to find the root of the transformed function math.atan(f(math.tan(x))) where f is the function $\boldsymbol{e^{x^3} - x + 1.5}$, starting from the initial guess **x=0.7**. The derivative of the transformed function is :

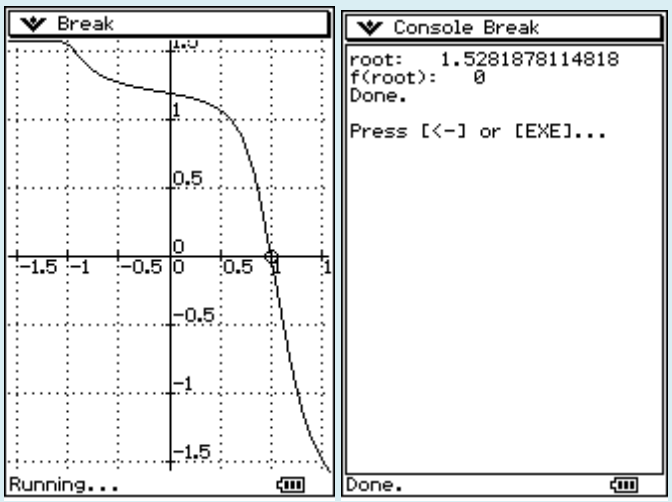
$$dftransdx(\mathrm{x}) = \frac{\left(\frac{1}{Cos(x)}\right)^2 (-1 - 3\mathrm{e}^{-\mathrm{Tan(x)}^3}\mathrm{Tan(x)}^2)}{1 + (1.5 + \mathrm{e}^{-\mathrm{Tan(x)}^3} - \mathrm{Tan(x)})^2}$$

And is defined analytically. In order to check the result, the program computes the function value at the root obtained, and shows the result graphically.

```
require("LNAplot/PlotFunc","LNAplot/PlotData","LNA/NewtonR")

local function f(x) return math.exp(-x^3)-x+1.5 end


local function ftrans(x) return math.atan(f(math.tan(x))) end

local function dftransdx(x) return (((1/math.cos(x))^2*(-1-3*math.exp(-
(math.tan(x))^3)*(math.tan(x))^2))/(1+(1.5+math.exp(-(math.tan(x))^3)-math.tan(x))^2)) end

local root,error,axesdata

root,error=NewtonR(ftrans,0.7,dftransdx)
print("root:",math.tan(root))
print("f(root):",f(math.tan(root)))

axesdata=PlotFunc(ftrans,{-math.pi/2,math.pi/2},{-math.pi/2,math.pi/2},false,1,"auto",true,1)
PlotData({root},{ftrans(root)},{},axesdata,true,2)
```

Example Program : **XNRGh**

The Figure shows the results obtained by running this program. Note that , in this particular example, function value **f(math.tan(root))** is exactly zero.



Results obtained by the example program **XNRGh**

Remarks

NewtonR is able to find one (and only one) root of the function **f**. If there are more than one roots, the root returned depends on the derivative of the transformed function and the initial guess. In other words, **NewtonR** does not necessarily return the root that is close to the initial guess.

If the argument dftransdx is omitted, **NewtonR** will approximate the derivative numerically. Although this numerical approximation is not trivial, it does not go too far, and may be inaccurate, especially at x close to zero. In general, numerical differentiation is kown to be very unsafe, and should be avoided. Consequently, although **dftransdx** is an optinal argument, you should supply it whenever this is possible.

The initial guess may be crucial for **Newton-Raphson** method. If the initial guess is bad, the algorithm may fail to converge. As a general rule, select an initial guess for which the derivative of the transformed is not close to zero. If the method diverges, check the definition of the function **f** and **dftransdx**, or try a different initial guess. Note that the most common error when using **NewtonR** is to define **dftransdx** incorrectly; this will cause show convergence to wrong results.

You may need to reduce the desired accuracy only in special cases. As in **Bisect**, the optional argument **maxit** is rarely needed, if the desired accuracy cannot be achieved with default number of iterations, it is more than likely that the desired accuracy is too high, or there is an error is the definition of **f** and **dftransdx**.

FILENAME: **NewtonR**.

DEPENDENCIES: **Epsilon**.

The Driver program code and it,s requirements code:

LNA/NewtonR

```
require("LNAutils/Epsilon")



function NewtonR(f,guess,...)

-- Optional arguments:

local dfdx="auto"

local eps=Epsilon

local maxit=100
```

```
local show=false
if arg["n"]>=1 then
 dfdx=arg[1]
 if arg["n"]>=2 then
  if arg[2]~="auto" then
  eps=arg[2] end
  if arg["n"]>=3 then
   maxit=arg[3]
   if arg["n"]>=4 then
    show=arg[4]
   end
  end
 end
end
-- Numerical derivative:
if dfdx=="auto" then
 local tiny=Epsilon^(1/3)
 local function dfdxnum(x)
 local h=x*tiny
 return (f(x+h)-f(x-h))/(2*h)
 end
 dfdx=dfdxnum
end
```

```
-- Local variables:

local root=guess

local preroot=-1E99

local froot,dfroot

-- Find root:

if show then print(" i  discrepancy") end

for i=1,maxit do

 froot=f(root)

 if show then printf("%2i  %+e\n",i,froot) end

 if (math.abs(root-preroot)<eps) or (froot==0) then

 return root end

 dfroot=dfdx(root)

 if dfroot==0 then

  print("Warning: NewtonR: No more iterations are possible. The derivative is equal to zero, but the accuracy criterion is not satisfied.")

  return root

 end

 preroot=root

 root=root-froot/dfroot

end

print("Warning: NewtonR: Maximum number of iterations reached, but the accuracy criterion is not satisfied.")

return root

end
```

```
export{NewtonR=NewtonR}
```

LNAutils/Epsilon

```
Epsilon=1.12E-16
export{Epsilon=Epsilon}
```

The Example program code and it,s requirements code:

LNAutils/XBrentGh

```
require("LNAplot/PlotFunc","LNAplot/PlotData","LNA/NewtonR")


local function f(x) return math.exp(-x^3)-x+1.5 end



local function ftrans(x) return math.atan(f(math.tan(x))) end


local function dftransdx(x) return (((1/math.cos(x))^2*(-1-3*math.exp(-
(math.tan(x))^3)*(math.tan(x))^2))/(1+(1.5+math.exp(-(math.tan(x))^3)-math.tan(x))^2)) end


local root,error,axesdata


root,error=NewtonR(ftrans,0.7,dftransdx)

print("root:",math.tan(root))

print("f(root):",f(math.tan(root)))
```

```
axesdata=PlotFunc(ftrans,{-math.pi/2,math.pi/2},{-math.pi/2,math.pi/2},false,1,"auto",true,1)

PlotData({root},{ftrans(root)},{},axesdata,true,2)
```

LNAutils/PlotFunc

```
require ("table","draw","LNAplot/PlotUtil")


function PlotFunc(f,xv,yv,...)

--Optional arguments:

local wait=true

local lwidth={1}

local tics={"auto","auto"}

local grid=true

local lblpos={0,0}

local lblsize={9,9}

local c="auto"

local discont={}

--

local scale

local lws,funcs,x,xp,X,Xp,Y,Yp,disconts

local id={}

if arg["n"]>=1 then
```

```
wait=arg[1]
if arg["n"]>=2 then
 lwidth=arg[2]
 if type(lwidth)=="number" then
  lwidth={lwidth}
 end
 if arg["n"]>=3 then
  tics=arg[3]
  if type(tics)~="table" then
   tics={tics,tics}
  end
  if arg["n"]>=4 then
   grid=arg[4]
   if arg["n"]>=5 then
    lblpos=arg[5]
    if type(lblpos)=="number" then
     lblpos={lblpos,lblpos}
    end
    if arg["n"]>=6 then
     lblsize=arg[6]
     if type(lblsize)=="number" then
      lblsize={lblsize,lblsize}
     end
```

```
    if arg["n"]>=7 then
     c=arg[7]
     if arg["n"]>=8 then
      discont=arg[8]
     end
    end
   end
  end
 end
end
lws=#lwidth
if type(f)=="function" then
 f={f}
end
funcs=#f
for i=lws+1,funcs do
 lwidth[i]=lwidth[lws]
end
discont,disconts=Discontinuities(funcs,discont)
for i=1,funcs do
 id[i]=1
```

```
 table.sort(discont[i])
end
draw.onbuffer()
if xv[1]~=nil then
 scale=SetScale(xv,yv)
 PlotAxes(xv,yv,scale,tics,grid,lblpos,lblsize,c)
else
 xv=yv[1]
 scale=yv[3]
 yv=yv[2]
end
Xp=0;xp=xv[1]
Y={};Yp={}
for i=1,funcs do
 Yp[i]=(yv[2]-f[i](xv[1]))*scale[2]
end
for X=1,X_pixels do
 x=xv[1]+X/scale[1]
 for i=1,funcs do
  Y[i]=(yv[2]-f[i](x))*scale[2]
  if id[i]>disconts[i] or x<discont[i][id[i]] or xp>discont[i][id[i]] then
   draw.line(Xp,Yp[i],X,Y[i],1,lwidth[i])
  elseif xp<=discont[i][id[i]] and x>=discont[i][id[i]] then
```

```
  id[i]=id[i]+1
 end
 Yp[i]=Y[i]
end
xp=x;Xp=X
end
showgraph()
draw.update()
if wait then waitkey() end
showconsole()
return {xv,yv,scale}
end


export{PlotFunc=PlotFunc}
```

LNAutils/PlotUtil

```
require ("string","table","draw","LNAutils/OrderMag")


X_max=158
Y_max=213
X_center=79
Y_center=106
```

```
X_pixels=159

Y_pixels=214


local function SetScale(xv,yv)

local xscale,yscale

xscale=X_pixels/(xv[2]-xv[1])

yscale=Y_pixels/(yv[2]-yv[1])

return {xscale,yscale}

end


local function AutoXtics(range)

--tics=math.max(math.floor(range/7),1)

local tics=OrderMag(range)

if range<4*tics then

 tics=tics/2

end

return tics

end


local function AutoYtics(range)

--tics=math.max(math.floor((yv[2]-yv[1])/9),1)

local tics=OrderMag(range)

if range<4*tics then
```

```
 tics=tics/2
end
return tics
end


function Discontinuities(funcs,discont)
local D=table.copy(discont)
local id={}
local disconts={}
for i=1,funcs do
 if D[i]==nil then
  D[i]={}
 elseif type(D[i])=="number" then
  D[i]={D[i]}
 end
 disconts[i]=#D[i]
end
return D,disconts
end


function PlotAxes(xv,yv,scale,tics,grid,lblpos,lblsize,c)
local Xc,Yc,X,Y,Xl,Yl,lblxp,lblyp
local lblcut=1E-8
```

```
--Auto tics selection:
if tics[1]=="auto" then
 tics[1]=AutoXtics(xv[2]-xv[1])
end
if tics[2]=="auto" then
 tics[2]=AutoYtics(yv[2]-yv[1])
end
--
if c=="auto" then
 c={0,0}
end
--[[ AXES ]]--
Xc=(c[1]-xv[1])*scale[1]
Yc=(yv[2]-c[2])*scale[2]
draw.line(0,Yc,X_max,Yc)
draw.line(Xc,0,Xc,Y_max)
--[[ TICS & GRID ]]--
if lblpos[1]==0 then
 Yl=Y_pixels-lblsize[1]
else
 Yl=Yc+1
end
if lblpos[2]==0 then
```

```
 Xl=2
else
 Xl=Xc+2
end
for xp=c[1]-math.floor((c[1]-xv[1])/tics[1])*tics[1],math.floor((xv[2]-c[1])/tics[1])*tics[1],tics[1]
do
 X=(xp-xv[1])*scale[1]
 draw.line(X,Yc-2,X,Yc+2)
 if grid then
  for Y=0,Y_pixels,4 do
   draw.pixel(X,Y)
  end
 end
 if lblsize[1]>0 and X>0 and X<X_pixels then
  lblxp=xp
  if math.abs(xp)<=lblcut then
   lblxp=0
  end
  draw.text(X+2,Yl,lblxp,1,lblsize[1])
 end
end
for yp=c[2]-math.floor((c[2]-yv[1])/tics[2])*tics[2],math.floor((yv[2]-c[2])/tics[2])*tics[2],tics[2]
do
 Y=(yv[2]-yp)*scale[2]
```

```
 draw.line(Xc-2,Y,Xc+2,Y)
 if grid then
  for X=0,X_pixels,4 do
   draw.pixel(X,Y)
  end
 end
 if lblsize[2]>0 and Y>0 and Y<Y_pixels then
  lblyp=yp
  if math.abs(yp)<=lblcut then
   lblyp=0
  end
  draw.text(Xl,Y-lblsize[2],lblyp,1,lblsize[2])
 end
end
end


function PlotPoint(X,Y,pointtype,pointsize)
--Non-filled circle:
if pointtype==0 then
 draw.point(X,Y,1,pointsize)
elseif pointtype==1 then
 draw.circle(X,Y,pointsize,1,1,-1)
 draw.point(X,Y)
```

```
--Crossed circle:
elseif pointtype==2 then
 draw.circle(X,Y,pointsize,1,1,-1)
 draw.line(X-pointsize,Y,X+pointsize,Y)
 draw.line(X,Y-pointsize,X,Y+pointsize)
--Filled circle:
elseif pointtype==3 then
 draw.circle(X,Y,pointsize,1,1,1)
--Non-filled rectangle:
elseif pointtype==4 then
 draw.rect(X-pointsize,Y-pointsize,X+pointsize,Y+pointsize,1,1,-1)
 draw.point(X,Y)
--Crossed rectangle:
elseif pointtype==5 then
 draw.rect(X-pointsize,Y-pointsize,X+pointsize,Y+pointsize,1,1,-1)
 draw.line(X-pointsize,Y,X+pointsize,Y)
 draw.line(X,Y-pointsize,X,Y+pointsize)
--Filled rectangle:
elseif pointtype==6 then
 draw.rect(X-pointsize,Y-pointsize,X+pointsize,Y+pointsize,1,1,1)
end
end
```

```
local function DrawLabel()

return

end


export{X_max=X_max,Y_max=Y_max,X_center=X_center,Y_center=Y_center,X_pixels=X_pixels,
Y_pixels=Y_pixels,SetScale=SetScale,Discontinuities=Discontinuities,PlotAxes=PlotAxes,PlotPoin
t=PlotPoint}
```

LNAutils/OrderMag

```
require("string")


function OrderMag(x)

local xexp=string.lower(string.format("%e",x))

local e=string.find(xexp,"e")

local n=string.len(xexp)

return 10^string.sub(xexp,e+1,n)

end


export{OrderMag=OrderMag}
```

LNAplot/PlotData

```
require ("table","draw","LNAplot/PlotUtil")
```

```
function PlotData(x,y,xv,yv,...)
--Optional arguments:
local wait=true
local ptype={1}
local psize={3}
local lwidth={0}
local tics={"auto","auto"}
local grid=true
local lblpos={0,0}
local lblsize={9,9}
local c="auto"
--
local points=#x
local scale
local sets,pws,lws,X,Xp,Y,Yp
if arg["n"]>=1 then
 wait=arg[1]
 if arg["n"]>=2 then
  ptype=arg[2]
  if type(ptype)=="number" then
   ptype={ptype}
  end
```

```
if arg["n"]>=3 then
 psize=arg[3]
 if type(psize)=="number" then
  psize={psize}
 end
 if arg["n"]>=4 then
  lwidth=arg[4]
  if type(lwidth)=="number" then
   lwidth={lwidth}
  end
  if arg["n"]>=5 then
   tics=arg[5]
   if type(tics)~="table" then
    tics={tics,tics}
   end
   if arg["n"]>=6 then
    grid=arg[6]
    if arg["n"]>=7 then
     lblpos=arg[7]
     if type(lblpos)=="number" then
      lblpos={lblpos,lblpos}
     end
     if arg["n"]>=8 then
```

```
        lblsize=arg[8]
        if type(lblsize)=="number" then
         lblsize={lblsize,lblsize}
        end
        if arg["n"]>=9 then
         c=arg[9]
        end
       end
      end
     end
    end
   end
  end
 end
end
if type(y[1])=="number" then y={y} end
sets=#y
pws=#ptype
for i=pws+1,sets do
 ptype[i]=ptype[pws]
end
pws=#psize
for i=pws+1,sets do
```

```
 psize[i]=psize[pws]
end
lws=#lwidth
for i=lws+1,sets do
 lwidth[i]=lwidth[lws]
end
draw.onbuffer()
if xv[1]~=nil then
 scale=SetScale(xv,yv)
 PlotAxes(xv,yv,scale,tics,grid,lblpos,lblsize,c)
else
 xv=yv[1]
 scale=yv[3]
 yv=yv[2]
end
Xp=(x[1]-xv[1])*scale[1]
Y={};Yp={}
for j=1,sets do
 Yp[j]=(yv[2]-y[j][1])*scale[2]
 PlotPoint(Xp,Yp[j],ptype[j],psize[j])
end
for i=2,points do
 X=(x[i]-xv[1])*scale[1]
```

```
 for j=1,sets do
  Y[j]=(yv[2]-y[j][i])*scale[2]
  if psize[j]>0 then
   PlotPoint(X,Y[j],ptype[j],psize[j])
  end
  if lwidth[j]>0 then
   draw.line(Xp,Yp[j],X,Y[j],1,lwidth[j])
   Yp[j]=Y[j]
  end
 end
 Xp=X
end
showgraph()
draw.update()
if wait then waitkey() end
showconsole()
return {xv,yv,scale}
end


export{PlotData=PlotData}
```