Some Slider Functions

Hans Peter Wolf

June 12, 2018

Contents

1 Slider functions for exploratory data analysis 3

2 Slider functions – overview and examples 4
   2.1 The functions of the paper ................................. 4
   2.2 slider.hist ................................................. 5
   2.3 slider.density ............................................. 6
   2.4 slider.brush.pairs ......................................... 7
   2.5 slider.brush.plot.xy ...................................... 8
   2.6 slider.split.plot.ts ...................................... 9
   2.7 slider.zoom.plot.ts ..................................... 10
   2.8 slider.smooth.plot.ts ................................... 11
   2.9 slider.lowess.plot .................................... 12
   2.10 slider.bootstrap.lm.plot .............................. 13

3 General issues of the implementation 14
   3.1 The structure of slider functions ......................... 14
      3.1.1 A very simple example ............................. 14
      3.1.2 The main arguments of slider ....................... 15
      3.1.3 Buttons and slider variables ....................... 16
   3.2 General Rules ............................................. 18
      3.2.1 Names of the slider functions ...................... 18
      3.2.2 Formal arguments for data sets .................... 18
      3.2.3 Passing additional arguments ..................... 20
   3.3 Density Presenter ......................................... 22

4 Histograms and Density Traces 24
   4.1 Class number of histograms – slider.hist ............... 24
      4.1.1 Test ................................................. 25
4.2 Width and kernel of a density trace – \texttt{slider.density} \hspace{1cm} 25
4.2.1 Test \hspace{2cm} 26
4.2.2 Help Page \hspace{2cm} 26

5 Brushing Functions \hspace{1cm} 28
5.1 A draftsman’s display with Brushing – \texttt{slider.brush.pairs} \hspace{1cm} 28
5.1.1 Test \hspace{2cm} 29
5.2 Scatter plot brushing – \texttt{slider.brush.plot.xy} \hspace{1cm} 29
5.2.1 Test \hspace{2cm} 30
5.2.2 Help Page \hspace{2cm} 30

6 Time Series Plots \hspace{1cm} 31
6.1 Splitted time series – \texttt{slider.split.plot.ts} \hspace{1cm} 31
6.1.1 Test \hspace{2cm} 33
6.1.2 Help Page \hspace{2cm} 33
6.2 Zooming in Time Series – \texttt{slider.zoom.plot.ts} \hspace{1cm} 34
6.2.1 Test \hspace{2cm} 35
6.2.2 Help Page \hspace{2cm} 35
6.3 Smoothing Time Series – \texttt{slider.smooth.plot.ts} \hspace{1cm} 36
6.3.1 Test \hspace{2cm} 37
6.3.2 Help Page \hspace{2cm} 37

7 Regression \hspace{1cm} 38
7.1 Smoothing by lowess – \texttt{slider.lowess.plot} \hspace{1cm} 38
7.1.1 Test \hspace{2cm} 38
7.1.2 Help Page \hspace{2cm} 38
7.2 Sensitivity of Regression Line – \texttt{slider.bootstrap.lm.plot} \hspace{1cm} 39
7.2.1 Test \hspace{2cm} 40
7.2.2 Help Page \hspace{2cm} 40

8 Appendix \hspace{1cm} 41
8.1 Definition of \texttt{slider} \hspace{2cm} 41
1 Slider functions for exploratory data analysis

In this paper we discuss a set of slider functions for constructing dynamical statistical plots. Some of the characteristics of these plots are controlled by sliders and you can modify the plots by the sliders. The sliders will be generated within Tcl/Tk widgets. Moving a slider results in changing the associated parameter and the graphics output is updated. For example, `slider.density()` plots a density trace and opens a Tcl/Tk widget with one slider that is linked to the width parameter of `density()`. After shifting the slider and releasing the mouse button the density trace is plotted again taking into account the new window width.

The next section contains a list of functions presented in this paper. Then you get some examples so you can see the function in action. Some special tricks are used to customize the behaviour of the slider functions. These programming pearls are discussed in section three. In the remaining sections you find the definitions of the slider function. We hope that the functions are useful for you in the way they are defined. However, you are invited to take a closer look at the code. Sometimes it will be very easy to enhance the proposals and you quickly get new slider functions.
2 Slider functions – overview and examples

2.1 The functions of the paper

The following list shows the functions available in this package:

```r
fns <- readLines("sliderfns.rev")
fns <- fns[grep("slider.*function", fns)]
sort(unique(sub("<-.*$", ",", fns)))
```

```
[1] "slider"          "slider.bootstr.lm.plot"
[3] "slider.brush.pairs" "slider.brush.plot.xy"
[5] "slider.density"   "slider.hist"
[7] "slider.lowess.plot" "slider.present.density"
[9] "slider.show.density" "slider.show.norm"
[13] "slider.split.plot.ts"  "slider.trimmed.mean"
[15] "slider.trimmed.mean.plus" "slider.trimmed.mean.simple"
[17] "slider.xyz"         "slider.zoom.plot.ts"
```
The next subsections contain a sequence of screen shots showing the some of the slider functions in action.

2.2 slider.hist

slider.hist computes histograms. The number of classes is controlled by a slider. As a special feature slider.hist has a panel argument. This argument allows you to deliver a panel function which draws some additional plotting elements. In the following example panel is used to add a normal density curve as well as a rug.

\[
\text{show slider.hist} \equiv \\
\text{define slider.hist}
\]

\[
\text{slider.hist(rivers,xlab="x",col="green",probability=TRUE,}
\text{panel=function(x){}
\text{ xx<-seq(min(x),max(x),length=100)
\text{ yy<-dnorm(xx,mean(x),sd(x))}
\text{ lines(xx,yy); rug(x); print(summary(yy))
\text{}}
\text{})}
\]

After moving the slider to position 17 the control widget looks like this:

![Slider control widget](image)

After releasing the mouse button the graphics device shows the following plot:

![Histogram plot](image)

You see that the fit of the normal distribution is not very good. Perhaps an exponential distribution is a quite better model. Can you build a new call of slider.hist with a suitable exponential density curve? Hint: You have to exchange the line of code that computes the normal density values:

```
remove yy<-dnorm(xx,mean(x),sd(x))
write yy<-dexp(xx,1/mean(x)).
```
2.3 slider.density

slider.density computes density traces. The parameter width and the type of the kernel function are controlled by sliders. Like slider.hist the function slider.density has a panel argument to add further graphical elements. In the example slider.density a normal density curve and a rug are added by the panel function.

```r
(show slider.density 3) ≡
(define slider.density 34)
slider.density(log(rivers),xlab="rivers",col="red",
  panel=function(x){
    xx<-seq(min(x),max(x),length=100)
    yy<-dnorm(xx,mean(x),sd(x))
    lines(xx,yy); rug(x); print(summary(yy))
  }
)
```

If you want to select a width of 3 percent of the range of the data and a rectangular kernel the slider control widget must have the following appearance. With these settings the graph will be very shaky.

The parameter settings "width=20", "kernel=epanechnikov" result in a much smoother plot.
2.4 `slider.brush.pairs`

`pairs()` constructs a draftsman’s display – a set of two dimensional scatterplots. This function is really useful for data sets with 3 to 10 variables because dependencies between a pair of variables can be seen at once.

However, sometimes it is very interesting to explore two dimensional dependencies of a subset of the data points that are defined by a third variable. This idea leads to the function `slider.brush.pairs` which computes a pairs plot and allows you to define an interval for the values of one variable. After changing the settings all data points of the visible interval are recolored using the color "red".

\[
\text{slider.brush.pairs(iris[-4])}
\]
The function `slider.brush.plot.xy` is a small version of `slider.brush.pairs()`. It is suitable for three dimensional data sets. `slider.brush.plot.xy()` computes a two dimensional scatterplot and chooses the color of the points conditional to a third variable. Therefore, you can start the exploration with `slider.brush.pairs()` and if you want to concentrate on one of the scatterplots — e.g. for the purpose of presentation — you can call `slider.brush.plot.xy()`.

```
(show of slider.brush.plot.xy 5) ≡
(define slider.brush.plot.xy 40)
slider.brush.plot.xy(iris[,1:3])
```
Time series often have different structures in the distinct intervals. Then it may be interesting to inspect various subsets of the data. For comparing different intervals (in time) you can split the whole time series into pieces and fit linear regression models to the various regions. The function `slider.split.plot.ts` has been designed for doing this job. By sliders you are able to define a point on the time axis as well as the length of a time interval (strip). Then a time series plot of your data is computed and it is partitioned into stripes in a way that one limit between two strip is identical with the selected point. Furthermore, regression lines of fitted linear models are drawn within each of the intervals. Alternatively `slider.split.plot.ts` allows you to draw the main characteristics of the subsets of points. A mouse click on the button *five- num summary* of the control widget will add graphical five number summaries of the data values of each region.

```r
show slider.split.plot.ts 6
define slider.split.plot.ts 43
slider.split.plot.ts(as.vector(sunspots)[1:100])
```
2.7 slider.zoom.plot.ts

Some time series are often very long. For inspection such a series you want to
be able to view the data of selected intervals. slider.zoom.plot.ts enables
you to choose two windows of the whole range of the data and the function
computes two time series plots of the two sections. With this function it is easy
to compare the structures of two different regions of a time series.

```r
(show slider.zoom.plot.ts 7) ≡
(define slider.zoom.plot.ts 46)
slider.zoom.plot.ts(as.vector(sunspots),2)
```
2.8 slider.smooth.plot.ts

The function `slider.smooth.plot.ts` allows you to smooth a time series by running medians or some other operations which has been proposed by Tukey. The implementation is based on `smooth()` and the operation is applied by clicking one of the buttons of the control widget.

```r
call slider.smooth.plot.ts(8) ≡
define slider.smooth.plot.ts(50)
slider.smooth.plot.ts(rnorm(100))
```
2.9 slider.lowess.plot

`lowess()` is an R function for analyzing the trend of points in a scatterplot. The smoothness of the computed line depends on a span parameter. Because of the algorithm is time consuming, `lowess` has a parameter to fix the number of iterations to be done. The function `slider.lowess.plot()` computes a scatterplot of a data set and lets you select these two parameters by an control widget.

(show slider.lowess.plot 9) ≡
(define slider.lowess.plot 53)
slider.lowess.plot(cars)
2.10 slider.bootstrap.lm.plot

Sometimes you have fitted a model curve on the base of a data set and you wonder how sensitive the result is against changes of the data. The function `slider.bootstrap.lm.plot()` draws a scatterplot containing the curve of a fitted model as well as some curves that have been computed on samples of the original data points. The user is allowed to select the degree of the polynom to be fitted, the number of bootstrap samples and the random seed.

```
10 (show slider.bootstrap.lm.plot 10) ≡
(define slider.bootstrap.lm.plot 56)
daten<-iris[,2:3]
slider.bootstrap.lm.plot(daten)
```
3 General issues of the implementation

3.1 The structure of slider functions

The last section discusses some slider functions and the reader has got examples of their application. Now we will explore the structure of the slider functions. All of them call the function `slider` to create and to control Tcl/TK widgets. The widgets consist of sliders and buttons that are linked to R functions describing the actions to be performed.

To demonstrate the usage of `slider` within the slider functions we discuss some examples now. As a very simple one let us build a slider function that prints the trimmed mean of a data set depending on the trim fraction which is controlled by a slider.

3.1.1 A very simple example

In this paragraph we define the function `slider.trimmed.mean.simple`. It shows the usage of the function `slider()` for doing the jobs

- create a control widget with one slider
- define the effect or the action to be done in case of a slider movement
- link the slider with the action.

Before going into details activate the following code chunk, please. It will define the function "slider.trimmed.mean.simple" and it will call the new function with the data set "rivers[1:50]":

```r
slider.trimmed.mean.simple<-function(x)
{
  # general initialisations
  if(missing(x)) return("Error: no data found!")
  # function to update result
  compute.trimmed.mean<-function(...){
    # get slider
    alpha <- slider(no=1)
    # print result
    cat(paste("trim fraction",alpha,":\",mean(x,trim=alpha),"\n"))
  }
  # definition of slider widget
  slider(compute.trimmed.mean,"trim fraction",0,0.5,0.01,0)
  # initial computation
  compute.trimmed.mean()
}
slider.trimmed.mean.simple(rivers[1:50])
```

Move the slider named "trim fraction" and observe what happens! Now let’s take a look at the definition of `slider.trimmed.mean.simple`. The function consists of four instructions:

- The first statement is an input check – business as usual.
• The second one defines the effect of a movement of the slider in form of an action function although the slider hasn’t been defined yet.

• Then the function slider is called to build the control widget and the action function is passed by the first parameter.

• At last the action function is called and a trimmed mean will be printed.

You see slider.trimmed.mean.simple() mainly consists of the definition of the action function compute.trimmed.mean() and a call of slider().

compute.trimmed.mean() fulfills two tasks: fetching the actual value of the slider variable (trim fraction alpha) and printing the trimmed mean. slider(no=1) reads the value of slider number "no=1" and then the value is assigned to the local variable alpha. The computation of the trimmed mean by mean() and its printing follows.

3.1.2 The main arguments of slider

A slider of a slider control widget is declared by six parameters. The meaning of them is found in following list.

1. sl.functions: an effect function which is called after moving the slider

2. sl.names: a text for labeling the slider

3. sl.mins: the minimum of the range of the slider

4. sl.maxs: the maximum of the range of the slider

5. sl.deltas: the increment of the scale of the slider

6. sl.defaults: the default value of the slider

These parameters are arguments of the function slider. The following example has the same structure as slider.trimmed.mean.simple(). However, the action function computes a plot and in the call of slider() the names of the formal parameters are used.
trim fraction:", round(alpha*100), ",\%\)"
}
# definition of slider widget
slider(sl.functions=refresh,
   sl.names=  "trim fraction",
   sl.mins=  0,
   sl.maxs=  0.5,
   sl.deltas=  0.01,
   sl.defaults=  0
)
# initial plot
refresh()
}
slider.trimmed.mean(rivers[1:50])

It may be astonishing that the argument names are in plural form. This indicates
that more than one slider can be defined in the same way. The following section
will show you an example with two sliders.

3.1.3 Buttons and slider variables

The next version of the running example — slider.trimmed.mean.plus() —
demonstrates how two sliders are implemented. To keeps things simple a second
slider is defined. It allows the user to change the color of the plot. In addition
three buttons are generated to switch between three graphical representations
of the data: dotplot, histogram or boxplot. The code is a little bit longer.
However, we are shure that it isn’t very difficult to understand it.

13 define slider.trimmed.mean plus 11)
slider.trimmed.mean.plus<-function(x)
{
   # general initialisations
   if(missing(x)) return("Error: no data found!"
slider(obj.name="ptype",obj.value="hist")
   # function to update plot
   refresh<-function(...){
      # get settings
      alpha <- slider(no=1)


mycol <- slider(no=2)
ptype <- slider(obj.name="ptype")
# computation
tm <- mean(x,trim=alpha); m <- mean(x)
# construct high level plot
if(ptype=="dotplot") plot(as.data.frame(x),col=mycol)
if(ptype=="hist") hist(x,col=mycol)
if(ptype=="boxplot") boxplot(x,col=mycol,horizontal=TRUE)
# perform some lower level jobs
abline(v=tm,col="red")
text(tm,1.1,adj=1,"trimmed mean",col="red")
abline(v=m,col="black")
text(m,0.9,adj=0,"mean")
title(paste("trimmed mean: ", format(tm,signif=4), 
"\ntrim fraction: ",round(alpha*100),"\%")
}
# three functions to react on buttons
b1<-function(...){slider(obj.name="ptype",obj.value="dotplot");refresh()}

b2<-function(...){slider(obj.name="ptype",obj.value="hist"); refresh()}

b3<-function(...){slider(obj.name="ptype",obj.value="boxplot");refresh()}
# definition of slider widget
slider(sl.functions= refresh,
   sl.names= c("trim fraction","color"),
   sl.mins= c(0,1),
   sl.maxs= c(0.5,13),
   sl.deltas= c(0.01,1),
   sl.defaults= c(0,1),
   but.functions=c(b1,b2,b3),
   but.names= c("dotplot","histogram","boxplot")
)

# initial plot
refresh()

}
slider.trimmed.mean.plus(rivers[1:50])
In this example vectors of length two are assigned to the arguments sl.names,
sl.mins, sl.maxs, sl.deltas, and sl.defaults. The first elements of the
vectors describe the settings of the first slider whereas the second ones fix the
parameters of the other slider.

The buttons are implemented by a vector of three names (but.names) and by a
vector of three effect functions (but.functions). To handle the state of the se-
lected plot type a slider variable (ptype) is created. During updating the plot the
actual value of ptype is found by the statement slider(obj.name="ptype").
After changing ptype it is stored by statements like slider(obj.name="ptype",
obj.value="dotplot"). In this example the effect functions of the buttons only
consist of a statement to store the plot type and to update the plot by calling
the effect function refresh().

In summary there is a simple structure for implementing sliders and buttons.
Slider variables can be used to communicate states, values and other information.

3.2 General Rules

In the last section the reader learned the syntax of the function slider and he
will be able to write slider function of his own. To achieve a similar structure
of the slider functions introduced in this paper we propose some general rules
now.

3.2.1 Names of the slider functions

The names of the slider functions of this paper always start with the character
string "slider" and end with a string indicating the job to be done, e.g. a
statistical procedure or function like hist, pairs or smooth.

3.2.2 Formal arguments for data sets

One dimensional data sets are passed to a slider* function by the argument x.
If the data set is of two dimensions there are two types of calls: Either argument
x is a matrix or a data frame with two columns or the arguments x and y are
used for the first and the second variable. If x is a vector and y not available
then x is used as the second dimension and seq(x) as the first one. Three
dimensional data sets are managed in an analogous way: A (n × 3)-matrix has
to be assigned to x or the arguments x, y and z must be used. These rules
enable us to write some general statements for checking and for preparation of
some local variables containing the data.

Checking the one dimensional case is simple. Besides x.name stores how the x
argument has been assigned by the user.

14 (check if x is a vector 14) ≡  ⊂ 31, 34, 38, 66
x.name<-deparse(substitute(x))
if(missing(x)||length(x)<2) return("Error: x must be a vector")

To check a two dimensional data input the two cases described above has to be handled. After checking the data are assigned to two local variables x and y.

(check if x is a matrix or x and y are vectors 15) ≡ ⊂ 53, 56

x.name<-deparse(substitute(x))
y.name<-deparse(substitute(y))
if(length(x)<2) return("Error: x is of length 0 or 1")
if(!is.null(y)){
  if(length(y)<2) return("Error: y must be a vector")
  if(length(x)!=length(y))
    return("Error: x and y must have the same length")
  x<-cbind(x,y)
}
if(!is.matrix(x) && !is.data.frame(x)){
x<-cbind(seq(x),x)
y.name<-x.name; x.name<-"index"
}
if(is.null(y.name)){x.name<-colnames(x)[1]; y.name<-colnames(x)[2]}
y<-x[2]; x<-x[,1]

In the case of three dimensions the arguments y and z are dealt like y in the two dimensional case.

(check if x, y,z are vectors or x is matrix with >2 columns 16) ≡ ⊂ 40

x.name<-deparse(substitute(x))
y.name<-deparse(substitute(y))
z.name<-deparse(substitute(z))
if(length(x)<2) return("Error: x is of length 0 or 1")
if(!is.null(y)){
  if(length(y)<2) return("Error: y must be a vector")
  if(length(x)!=length(y))
    return("Error: x and y must have the same length")
  x<-cbind(x,y)
}
if(!is.null(z)){
  if(length(z)<2) return("Error: z must be a vector")
  if(length(x)!=length(z))
    return("Error: x and z must have the same length")
  x<-cbind(x,z)
}
if(!is.matrix(x)&& !is.data.frame(x) && ncol(x)<3)
  return("Error: not enough variables")
if("NULL"==y.name){x.name<-colnames(x)[1]; y.name<-colnames(x)[2]}
if("NULL"==z.name){x.name<-colnames(x)[1]; z.name<-colnames(x)[3]}
z<-x[3]; y<-x[,2]; x<-x[,1]

Time series are different from a one dimensional data set because data points are connected with times. Therefore, we need a code chunk for checking time series.

(check if x is a time series 17) ≡ ⊂ 43, 46, 50

x.name<-deparse(substitute(x))
if(missing(x)||length(x)<2) return("Error: x must be a vector")
y<-x; if(is.ts(x) { x<-time(x } else { x<-seq(x}
3.2.3 Passing additional arguments

Remark: This section should be skipped during the first reading of the text!

You know that high level plotting function have the formal parameter "...". This argument allows the user to pass additional settings to the graphics device. At once the question arises: Can we manage to pass such arguments to plotting functions that are called by a slider function? For example: a histogram is computed in the refresh function and the user wants to produce green bars.

There are different answers to tackle this problem. At first you can consider which of the graphics parameters are relevant to be set by the user. Then we can introduce these parameter in the header of a slider function. But the list of parameters will be very long because the user may be want to modify attributes of scale, point size, type, color, etc.

A second way is to define a "..." argument for passing additional parameters.

slider.xyz<-function(x,...)

But there will be some difficulties to be managed if you use the "..." construction. How do we handle the "..." argument, how do we set default values for some parameter and how can we remove parameters that must not be changed at all? In the initialization part of the slider.* functions it is possible to modify the list of the additional arguments. To demonstrate this step we show how you can remove a breaks argument and then append a "main" argument. If main is not available it is set to x.name. (Cleary there is no graphics parameter with name breaks but users used to set breaks in their calls of hist may try to deliver a breaks argument.)

args<-list(...) # remove "breaks"
if(!any("main"==names(args))) # set default "main"
  args<-c(args,list(main=x.name))

Now the argument list args has to be passed to the function hist within the action or refresh function. Writing hist(x,args) doesn't work because the second argument of hist is the break argument and not a slot for further arguments.

The second idea will be to use the "..." of hist: hist(x,...=args). But the "..." argument of hist is passed to the functions title and axis and so we don't get green bars at all. Our answer to this problem is to describe the call by its elements: the function hist should be called with the argument list consisting of the additional arguments stored in args and other arguments specifying the data set and the breaks, for example. Then the function do.call is called with these elements. Keep in mind that do.call needs the arguments as an alist.

To summarize the following construction will work and is used in the function slider.hist:

slider.hist<-function(x,...)
{
  #initialisation statements NA
  args<-list(...)
Before we close this section we will present a further proposal to deal with the arguments. In the game there are four sets of information:

- the set of default settings coded in the slider function
- the set of additional parameters defined by the user
- the set of forbidden arguments that must not be used
- the set of important settings that must not be changed by the user

We can save these sets on suitable list objects:

```r
defaults<-list(main=x.name)
args<-list(...)
set.of.forbidden.args<-list(probability=TRUE)
important<-alist(bty="n",sub="graphics computed by slider")
```

Now the sets have to be combined. At first we remove the forbidden elements from the argument list of the user. Then we concatenate the important ones, the user arguments and the default settings. Finally, we remove elements whose names occur on the list earlier.

```r
args<-args[!names(args) %in% set.of.forbidden.args]
defaults<-c(important,args,defaults)
args<-defaults[unique(names(defaults))]
```

The refresh function has a very simple structure now.

```r
refresh<-function(...){
xrange<-range(x); num<-slider(no=1)
breaks<-seq(xrange[1],xrange[2],length=num+1)
do.call("hist",c(alist(x=x,breaks=breaks),args))
panel(x)
}
```
In the following we have not implemented these ideas completely. Sometimes we pass ... arguments without further checks.

3.3 Density Presenter

In this paragraph we discuss some tools for plotting densities of distributions. The parameters of the distribution are controlled by sliders. These functions allow the user to study how the shapes of the densities depend on the parameter settings. It is a very easy exercise to find an implementation for a special distribution, e.g., the normal distribution.

```r
slider.show.normal.density <- function(name,...) {
  x<-seq(-10,10,length=200)
  args<-list(...)
  refresh<-function(...){
    par1<-slider(no=1); par2<-slider(no=2)
    f.x<-dnorm(x,par1,par2)
    main=paste("normal distribution\n", "E(X) =",format(par1,digits=3), ", sd(X) =",format(par2,digits=3))
    # plot(x,f.x,type="l",main=main)
    do.call("plot",c(alist(x=x,y=f.x,type="l",main=main),args))
  }
  slider(refresh,
    c("expectation","standard deviation"),
    c(-20,.001),c(20,20),c(.1,.1),c(0.1,1)
  )
  refresh()
  cat("use slider to select parameters!\n")
}
```

Here comes a check.

```r
slider.show.normal.density(col="red",bty="n")
```

A more interesting idea is to write a function that operates as a general presenter. The function `slider.show.density()` shows the shapes of the densities of various distributions.

After the user has collected some experiences she or he will like to write an improved version. Before you start programming your own presenter analyze the following function:

```r
slider.show.density <- function( 
  distribution="norm", 
  mins=c(-100,-100,-50,.001), 
  maxs=c(100,100,50,100), 
  deltas=c(1,1,1,1), 
  defaults=c(-5,5,0,1),type="l",...) 
{
  args<-list(...)
  refresh<-function(...){
    lim1<-slider(no=1); lim2<-slider(no=2)
```
par1<-slider(no=3); par2<-slider(no=4)
x<-seq(min(lim1,lim2),max(lim1,lim2),length=200)
ddist<-paste("d",distribution,sep="")
f.x<-do.call(ddist,alist(x=x,par1,par2))
main=paste("distribution: ",distribution,"\n",
"parameter 1: ",format(par1,digits=3),
"parameter 2: ",format(par2,digits=3))
# plot(x,f.x,type=type,main=main)
do.call("plot",c(alist(x=x,y=f.x,type=type,main=main),args))
}
slider(refresh,
c("x limit (min)","x limit (max)",
"parameter 1","parameter 2")[1:length(mins)],
mins,maxs,deltas,defaults)
refresh()
cat("use sliders to select parameters!\n")
)

Here are a few examples to show that the idea works:

• gamma distribution

26 \textit{show the density of the gamma distribution 26} \equiv \subset 30
slider.show.density("gamma",mins=c(0,0,.01,.01),
maxs=c(100,100,50,50),
deltas=c(.1,.1,.01,.01),
defaults=c(0,10,4,1),bty="n")

• beta distribution

27 \textit{show the density of the beta distribution 27} \equiv \subset 30
slider.show.density("beta",mins=c(0,0,0,0),
maxs=c(1,1,10,10),
deltas=c(.1,.1,.01,.01),
defaults=c(0,1,1,1),col="blue")

• binomial distribution

28 \textit{show the density of the binomial distribution 28} \equiv \subset 30
slider.show.density("binom",mins=c(0,0,1,0),
maxs=c(100,100,100,1),
deltas=c(1,1,1,.01),
defaults=c(0,20,5,0.5),type="h")

• Poisson distribution

29 \textit{show the density of the Poisson distribution 29} \equiv \subset 30
slider.show.density("pois",mins=c(0,0,.01),
maxs=c(100,100,50),
deltas=c(1,1,.1),
defaults=c(0,20,5),type="h")

The different presenter functions could be collected in central function.
slider.present.density<-function()
{
  (define slider.show.density 25)
  f0<- (define slider.show.normal.density 23)
  f1<- function(...){ (show the density of the gamma distribution 26) }
  f2<- function(...){ (show the density of the beta distribution 27) }
  f3<- function(...){ (show the density of the binomial distribution 28) }
  f4<- function(...){ (show the density of the Poisson distribution 29) }

  slider(but.functions=c(f0,f1,f2,f3,f4),
        but.names=c("show NORMAL","show GAMMA","show BETA",
                   "show BINOMIAL","show POISSON"))
}

slider.present.density()

Now the reader is prepared to take a further look at the implementations of the slider functions of this paper.

4 Histograms and Density Traces

4.1 Class number of histograms – slider.hist

In this section we propose a function which allows you to study the effect of the number of classes on the shape of a histogram. For this purpose a widget with a slider controlling the number of classes is opened and a histogram is computed. By moving the slider the user changes the number of classes and the histogram is redrawn. The implementation uses two special features: 1.) Sometimes we want to add further graphical elements like a density curve to the histogram. For this job slider.hist has a slot to deliver a panel function. This panel function will be called after updating the histogram. 2.) Additional graphics parameter can be transfered by the "..." argument of slider.hist to the call of hist which constructs the histogram.

Remarks: The default number of breaks is computed by hist. If a breaks argument is found in the call of slider.hist it will be removed.

slider.hist<-function(x,panel=rug,...)
{
  (check if x is a vector)
  args<-list(...)
  args<-(args[names(args)!="breaks"]
  ClassNumber<-length(hist(x,plot=FALSE)$breaks)
  if(!any("main"==names(args)))args<-c(args,list(main=x.name))
  refresh<-function(...){
    xrange<-(range(x)); num<-(slider(no=1)
    breaks<-(seq(xrange[1],xrange[2],length=num+1)
    do.call("hist",c(alist(x=x,breaks=breaks),args))
    panel(x)
  }
  slider(refresh,"ClassNumber",1,100,1,ClassNumber); refresh()
  "use slider to select number of classes"
}
4.1.1 Test

Here is a check to test whether we can change the color.

\[
\langle \text{check } 32 \rangle \equiv \\
\langle \text{define slider.hist } 31 \rangle \\
\text{slider.hist(log(islands),col="green")}
\]

To test the panel feature we add a normal density curve to the histogram.

\[
\langle \text{check } 32 \rangle + \equiv \\
\langle \text{define slider.hist } 31 \rangle \\
\text{slider.hist(rivers,xlab="RIVERS",col="red",probability=TRUE,} \\
\text{pa=function(x){} \\
\text{xx<-seq(min(x),max(x),length=100) \\
\text{yy<-dnorm(xx,mean(x),sd(x))} \\
\text{lines(xx,yy); rug(x); print(summary(yy))} \\
\text{}})
\]

4.2 Width and kernel of a density trace – slider.density

The main parameters of density traces are width and kernel type. In the function \text{slider.density()} both of them are selected by sliders. The alternative idea to implement the selection of the kernel function by buttons would have required seven buttons. Therefore, a slider solution seems to be a little bit smarter. To be able to add further graphical elements a panel function has been included.

Technical Remarks: 1.) The kernel type is stored in the slider variable \text{kno}. 2.) In \text{slider.density()} we have two different effect functions where the second one \text{set.kernel()} calls the first slider function (\text{refresh()}).

\[
\langle \text{define slider.density } 34 \rangle \equiv \\
\text{slider.density<-function(x,panel=rug,...)} \\
\{
\langle \text{check if } x \text{ is a vector } 14 \rangle \\
\text{args<-list(...)} \\
\text{if(!any("main"==names(args))) args<-c(args,list(main=x.name))} \\
\text{kernel<-c("gaussian", "epanechnikov","rectangular",} \\
\text{ "triangular","biweight", "cosine", "optcosine")} \\
\text{slider(obj.name="kno",obj.value=1)} \\
\text{refresh<-function(...){} \\
\text{width<-slider(no=1)*diff(range(x))/100} \\
\text{kno<-slider(obj.name="kno"); kernel<-kernel[kno]} \\
\text{xy<-density(x,width=width,kernel=kernel)} \\
\text{do.call("plot",c(alist(x=xy),args))} \\
\text{title(paste("\n\nwidth =",signif(width,4),", kernel =",kernel))} \\
\text{panel(x)} \\
\} \\
\text{set.kernel<-function(...){} \\
\text{kernel<-slider(no=2)} \\
\text{slider(obj.name="kno",obj.value=kernel)} \\
\text{refresh()}
\]

25
```r
bw.default <- diff(range(x))/density(x)$bw
t <- slider(c(refresh, set.kernel),
             c("width (% of range)", "kernel"),
             c(.1, 1), c(100, 7), c(.1, 1), c(bw.default, 1))
# tkwm.minsize(nt, "300", "110") # set width, height to prevent to small sizes
refresh()
cat("use slider to select width of window and to select kernel:
")
print(cbind("no"=1:7,"kernel"=kernel))
```

### 4.2.1 Test

At first we test `slider.density` without a panel function.

```r
(check of slider.density 35) ≡
(define slider.density 34)
slider.density(rivers, xlab="RIVERS", col="red")
```

Now we test a call of `slider.density` with a panel function to plot a fitted normal density curve.

```r
(check of slider.density 35)+ ≡
(define slider.density 34)
slider.density(log(rivers), xlab="rivers", col="red",
             panel=function(x){
                 xx <- seq(min(x), max(x), length=100)
                 yy <- dnorm(xx, mean(x), sd(x))
                 lines(xx, yy); rug(x); print(summary(yy))
             })
)
```

### 4.2.2 Help Page

```r
(define help of slider.hist 37) ≡
\name{slider.hist}
\title{interactive histogram and density traces}
\alias{slider.hist}
\alias{slider.density}
\description{
  The functions \code{slider.hist} and \code{slider.density} compute histograms and density traces whereas some parameter are controlled by sliders.
  
  \code{slider.hist} computes a histogram; the number of classes is defined by a slider.
  
  \code{slider.density} computes a density trace; width and type of the kernel are defined by sliders.
}
\usage{

slider.hist(x, panel, ...)  
slider.density(x, panel, ...)  
}

\arguments{
\item{x}{ data set to be used for plotting } 
\item{panel}{ function constructing additional graphical elements to the plot} 
\item{\dots}{ additional (graphics) parameters which are passed to 
the invoked high level plotting function } 
}

\details{
\code{slider.hist} draws a histogram of the data set \code{x} by 
calling \code{hist} and opens a Tcl/Tk widget with one slider. 
The slider defines the number of classes of the histogram. Changing the 
slider results in redrawing of the plot. For further 
details see the help page of \code{hist}. \code{rug} is used as the 
default panel function. 

\code{slider.density} draws a density trace of the data set \code{x} 
by \code{plot(density(...))} and opens a Tcl/Tk-widget with two 
sliders. The first slider defines the width of the density trace 
and the second one the kernel function: 
\code{"1-gaussian", "2-epanechnikov", "3-rectangular", 
"4-triangular", "5-biweight", "6-cosine", "7-optcosine"} 
Changing one of the sliders results in a redrawing of the plot. 
For further details see the help page of \code{density}. 
\code{rug} is used as the default panel function. 
}

\value{
 a message about the usage 
}

\references{ -- } 
\author{ Hans Peter Wolf } 
\seealso{ \code{\link{hist}}, \code{slider} 
}
\examples{
\dontrun{
## This example cannot be run by examples() but should be work in an interactive R session 
slider.hist(log(islands)) 
} 
\dontrun{
## This example cannot be run by examples() but should be work in an interactive R session 
slider.density(rivers,xlab="rivers",col="red") 
} 
\dontrun{
## This example cannot be run by examples() but should be work in an interactive R session 
slider.density(log(rivers),xlab="rivers",col="red", 
panel=function(x){ 
 xx<-seq(min(x),max(x),length=100) 
 yy<-dnorm(xx,mean(x),sd(x)) 
 lines(xx,yy) 
 rug(xx) 
 print(summary(yy)) 
 } 
) 
} 

27
5 Brushing Functions

5.1 A draftsman’s display with Brushing – slider.brush.pairs

A draftsman’s display is a nice graphics to show two dimensional dependencies of the variables of a multivariate data set. Brushing allows you to mark a subset of the data points. The subset of the points is defined by the condition that the coordinate of a selected variable has to lie within a fixed interval. By the function slider.brush.pairs() the user can select a variable (dimension) as well as an interval for recoloring the points satisfying a condition with color "red". The function pairs() is not used in the refresh function avoiding computational overhead. So the complexity of code is mostly caused by constructing the plot.

```r
slider.brush.pairs <- function(x, ...)
{
  args <- list(...)
  # preparation of data
  m <- dim(x)[2]; for (j in 1:m) x[, j] <- as.numeric(x[, j])
  mins <- apply(x, 2, min); maxs <- apply(x, 2, max)
  delta <- (maxs - mins) / 100
  # initial plot
  varnames <- paste("var ", 1:m, " : ", colnames(x), sep = "")
  dev.new(); par(mfrow = c(m, m), oma = c(0, 0, 0, 0), mai = c(0, 0, 0, 0), ...)
  usr.array <- array(0, c(m, m, 4)); axes <- FALSE
  for (i in 1:m){
    for (j in 1:m){
      # plot(x[,j],x[,i],axes=axes,type="p")
      do.call("plot", c(alist(x=x[,j], y=x[,i], type="p", axes=axes, xlab="", ylab=""), args))
      usr.array[i, j, ] <- usr <- par()$usr
      if (i == j) text(usr[1], usr[4], varnames[i], adj = c(0, 1), cex = 5)
      rect(usr[1], usr[3], usr[2], usr[4])
    }
  }
  # update function
  refresh <- function(...){
    vmin <- slider(no = 1)/100; vmax <- vmin + slider(no = 2)/100
    vno <- slider(no = 3)
    vmin <- mins[vno] * (1 - vmin) + maxs[vno] * (vmin)
    vmax <- mins[vno] * (1 - vmax) + maxs[vno] * (vmax)
    ind <- vmin <= x[, vno] & x[, vno] <= vmax
    for (i in 1:m){
      for (j in 1:m){
        par(mfg = c(i, j), usr = usr.array[i, j, ])
        points(x[, j], x[, i], col = 0, cex = 2, pch = 19)
        points(x[ind, j], x[ind, i], col = "red", pch = 1)
        points(x[!ind, j], x[!ind, i], col = "blue", pch = 19)
      }
    }
  }
```

38 \( \text{define slider.brush.pairs 38} \equiv \subset 4, 39, 60 \)
5.1.1 Test

We will use the famous iris data to test the brushing function.

5.2 Scatter plot brushing – slider.brush.plot.xy

The function slider.brush.plot.xy() computes an xy-plot and recolors a data point "red" if the value of its third variable is in the fixed interval.
rect(pos[2],pos[4],pos[1]*.4+pos[2]*.6,pos[3]*.1+pos[4]*.9,
col="white",border=NA)

} slider(refresh,
c("minimum of z","interval width of z"),
c(z.min,0),c(z.max+delta,(z.max-z.min)*delta),
c(delta,delta),c(z.min-delta,(z.max-z.min)/2),
reset.function=reset
)

refresh()
cat("use sliders to select interval for inking points\n")

5.2.1 Test

⟨ test of slider.brush.plot.xy 41 ⟩ ≡
⟨ define slider.brush.plot.xy 40 ⟩
# data<-matrix(rnorm(900), 300,3)
# slider.brush.plot.xy(data[,1],data[,2],sqrt(data[,1]^2+data[,2]^2),
# main="hallo")
# slider.brush.plot.xy(iris[,1],iris[,2],iris[,3])
slider.brush.plot.xy(iris[,1:3],cex=1.5)

5.2.2 Help Page

⟨ define help of slider.brush.pairs and of slider.brush.plot.xy 42 ⟩ ≡
\name{slider.brush}
\title{interactive brushing functions}
\alias{slider.brush.pairs}
\alias{slider.brush.plot.xy}
\description{
These functions compute a pairs plot or a simple xy-plot and
open a slider control widget for brushing.

\code{slider.brush.pairs} computes a pairs plot; the user defines an
interval for one of the variables and in effect all data points
in this interval will be recolored.

\code{slider.brush.plot.xy} computes an xy-plot; the user defines a
interval for a third variable \code{z} and all points
\code{(x,y)} will be recolored red if the \code{z} value is in the inter-
val.
}
\usage{
slider.brush.pairs(x, ...)
slider.brush.plot.xy(x, y, z, ...)
}
\arguments{
\item{\dots}{ new settings for global graphics parameters }
\item{x}{ matrix or data frame or vector }
\item{y}{ vector of y values if \code{x} is not a matrix }
\item{z}{ vector of z values if \code{x} is not a matrix }

\details{
\code{slider.brush.pairs} draws a pairs plot of the data set \code{x}. The first slider defines the lower limit of the interval and the second its width. By the third slider a variable is selected. All data points for which the selected variable is in the interval are recolored red.

\code{slider.brush.plot.xy} draws an xy-plot of the data set \code{x}. The first slider defines the lower limit of the interval of z values and the second one its width. All data points for which the variable z is in the interval are recolored red.
}

\value{
 a message about the usage
}


\author{ Hans Peter Wolf }

\seealso{ \code{\link{pairs}}, \code{\link{plot}} }

\examples{
\dontrun{
## This example cannot be run by examples() but should be work in an interactive R session
 slider.brush.pairs(iris)
}
\dontrun{
## This example cannot be run by examples() but should be work in an interactive R session
 slider.brush.plot.xy(iris[,1:3])
}
}

\keyword{ iplot }

6 Time Series Plots

6.1 Splitted time series – slider.split.plot.ts

Often there is a periodical structure in time series of a fixed period. Then you will like to have a tool to separate and compare the sections defined by the season. The function \code{slider.split.plot.ts} lets you select the length of a saison and one of the limits between two saisons. Then in the time series plot \code{fivenum} summary statistics or a regression lines are added to each of the sections.

\footnote{\texttt{(define slider.split.plot.ts 43) } \equiv \subset 6, 44, 60
\texttt{ slider.split.plot.ts<-function(x,type="l",...)}
\texttt{ }}
check if \( x \) is a time series

\[
\text{args} <- \text{list}(\ldots)
\]
\[n <- \text{length}(x) ; \; \text{xmin} <- \text{min}(x) ; \; \text{xmax} <- \text{max}(x)\]
\[\text{xdelta} <- \text{xmax} - \text{xmin}\]
\[
\text{slider(obj.name="summary.type",obj.value="linear")}
\]
\[
\text{refresh} <- \text{function}(\ldots)\{
\# \text{initialization}
\text{summary.type} <- \text{slider(obj.name="summary.type")}
\text{width} <- \text{slider(no=1)}
\text{limit} <- \text{slider(no=2)}
\text{n.sec} <- 1
\text{limit} <- \text{limit} - \text{width} * \text{ceiling}((\text{limit} - \text{xmin}) / \text{width})
\# \text{plot:}
\} \text{do.call("plot",c(alist(x,y,type=type),args))}
\text{limit} <- \text{limit} - \text{width} * \text{n.sec} ; \; j <- 0
\# \text{abline(v=limits,lwd=0.5,lty=3)}
\text{while(limit<xmax)} \{ \; j <- j+1
\text{limit} <- \text{limit} + \text{width} / \text{n.sec} ; \; \text{limit2} <- \text{limit} + \text{width}
\text{ind} <- \text{limit} < x & x <= \text{limit2}
\text{xx} <- x[ind] ; \; \text{yy} <- y[ind] ; \; \text{if}(\text{length(xx)}<2) \text{next}
\text{abline(v=limit,lwd=0.5,lty=3,col=1)}
\text{if(summary.type=="linear")}{
\text{coef} <- \text{lm(yy} \sim \text{xx}$\text{)coeff$
\text{segments(limit, coef[1]+coef[2]*limit,}
\text{limit2,coef[1]+coef[2]*limit2,col=j)}
\}
\text{if(summary.type=="five.number")}{
\text{five} <- \text{fivenum(yy)}
\text{xx} <- 0.5*(\text{limit} + \text{limit2})
\text{points(xx,five[3],pch=19,cex=1,col="red")}
\text{segments(xx,five[1],xx,five[2],lwd=3,col="red")}
\text{segments(xx,five[4],xx,five[5],lwd=3,col="red")}
\}
\}
\text{abline(v=slider(no=2),lwd=0.5,lty=1,col=1)}
\}
\text{f1} <- \text{function}(\ldots)\{
\text{slider(obj.name="summary.type",obj.value="linear")}
\text{refresh()}
\}
\text{f2} <- \text{function}(\ldots)\{
\text{slider(obj.name="summary.type",obj.value="five.number")}
\text{refresh()}
\}
\text{slider(refresh,c("width of window","limit"),}
\text{c(xdelta/length(x)*3,xmin),c(xdelta,xmax),}
\text{c(xdelta/1000,xdelta/1000),c(xdelta/4,xmin),}
\text{but.functions=c(f1,f2),}
\text{but.names=c("linear model","fivenum summary")}
\}
\text{refresh()}
cat("select window and summary type and look at time series!\n")
}
6.1.1 Test

The test call uses the time series co2.

\[
\begin{align*}
\text{test of slider.split.plot.ts} & \equiv \\
\text{define slider.split.plot.ts} & \equiv \\
\text{slider.split.plot.ts(co2,col="red",main="co2")}
\end{align*}
\]

6.1.2 Help Page

\[
\begin{align*}
\text{define help of slider.split.plot.ts} & \equiv \\
\text{name}{\text{slider.split.plot.ts}} & \equiv \\
\text{title}{\text{interactive splitting of time series}} & \equiv \\
\text{alias}{\text{slider.split.plot.ts}} & \equiv \\
\text{description}{
\begin{align*}
\text{code}{\text{slider.split.plot.ts}} & \text{ plots linear fitted lines or summary statistics in sections of a time series.} \\
\text{The sections are controlled by sliders.}
\end{align*}
}\}
\text{usage}{
\begin{align*}
\text{slider.split.plot.ts(x, type="1", ...)}
\end{align*}
}\}
\text{arguments}{
\begin{align*}
\text{item}{\text{x}} & \text{ time series or vector} \\
\text{item}{\text{type}} & \text{ plotting type: } \text{code}{\text{type}} \text{ will be forwarded to function } \text{code}{\text{plot}} \text{ } \\
\text{item}{\text{\dots}} & \text{ additional graphics parameters }
\end{align*}
}\}
\text{details}{
\begin{align*}
\text{code}{\text{slider.split.plot.ts}} & \text{ draws a time series plot and let you define sections of the series by fixing a limit on the time scale as well as a window width.} \\
\text{The whole range of the series is partitioned in pieces of the same length in a way that the fixed limit will be one of the section limits.} \\
\text{Then linear models are fitted and plotted in the sections.} \\
\text{Alternatively -- by pressing the button } \text{code}{\text{fivenum summary}} \text{ -- summary statistics are drawn instead of the model lines.}
\end{align*}
}\}
\text{value}{
\begin{align*}
\text{a message about the usage}
\end{align*}
}\}
\text{author}{\text{Hans Peter Wolf}}
\text{seealso}{\text{code}{\text{\link{plot}}}}
\text{examples}{
\begin{align*}
\text{dontrun}{
\begin{align*}
\text{## This example cannot be run by examples() but should be work in an interactive R session}
\text{slider.split.plot.ts(as.vector(sunspots)[1:100])}
\end{align*}
\end{align*}
}\}
\]
6.2 Zooming in Time Series – slider.zoom.plot.ts

Some times time series happen to be very long — for example sunspots has a length of 2820. For exploring a section of a series it is helpful to have a zooming function. The function slider.zoom.plot.ts allows you to define a time window interactively and it computes a time series plot of the selected region. As a second feature the function extracts the data of two windows and computes two time series plots one below the other. With this feature you are able to compare different regions of the series by eye.

```
\keyword{ slider.zoom.plot.ts }  
\keyword{ define slider.zoom.plot.ts 46 } \equiv \subset 7, 47, 48, 60 
slider.zoom.plot.ts<-function(x,n.windows=1,...)
{
  \check{ x is a time series } 17
  args<-list(...)
tmin<-1; tmax<-length(x)
refresh<-function(...){
  \# initialization
  width <- slider(no=1); tstart1 <- slider(no=2); tend1 <- width+tstart1
  if(n.windows>1){
    tstart2 <- slider(no=3); tend2 <- width+tstart2
  }
  \# plot
  par(mfrow=c(2+(n.windows>1),1),mai=c(.5,.5,.1,0))
  do.call("plot",c(alist(x,y,type="l",bty="n"),args))
  abline(v=c(x[tstart1],x[tend1]),col="red")
  lines(x[tstart1:tend1],y[tstart1:tend1],col="red",lty=2)
  if(n.windows>1){
    abline(v=c(x[tstart2],x[tend2]),col="blue")
    lines(x[tstart2:tend2],y[tstart2:tend2],col="blue",lty=3)
  }
  usr<-par()$usr
  ind<-tstart1:tend1
  plot(x[ind],y[ind],type="b",col="red",bty="n", # ylim=usr[3:4],
  xlim=c(x[tstart1],x[tstart1]+width*diff(x[1:2])))
  if(n.windows>1){
    ind<-tstart2:tend2
    plot(x[ind],y[ind],type="b",col="blue",bty="n", # ylim=usr[3:4],
  xlim=c(x[tstart2],x[tstart2]+width*diff(x[1:2])))
  }
  par(mfrow=c(1,1))
}

if(n.windows<2){
  slider(refresh,c("width of window","start of window"),
  c(1,1),c(tmax,tmax),c(1,1),c(ceiling(tmax/4),1))
}else{
  slider(refresh,
  c("width of window","start window 1","start window 2"),
  c(1,1),c(tmax,tmax),c(1,1),c(ceiling(tmax/4),1,ceiling(tmax/2)))
}
```
refresh()
cat("select window and look at time series!\n")
}

6.2.1 Test

The first test will be done with co2.

\begin{verbatim}
(test of slider.zoom.plot.ts 47) ≡
(define slider.zoom.plot.ts 46)
slider.zoom.plot.ts(co2,2,main="co2")
\end{verbatim}

For comparing two sections of a series take a look at sunspots.

\begin{verbatim}
(test of slider.zoom.plot.ts 47)+ ≡
(define slider.zoom.plot.ts 46)
slider.zoom.plot.ts(sunspots,2)
\end{verbatim}

6.2.2 Help Page

\begin{verbatim}
(define help of slider.zoom.plot.ts 49) ≡
\name{slider.zoom.plot.ts}
\title{interactive zooming of time series}
\alias{slider.zoom.plot.ts}
\description{
This function shows one or two sections of a time series. The window(s) is
(are) controlled by sliders.
}
\usage{
slider.zoom.plot.ts(x, n.windows, ...)
}
\arguments{
\item{x}{ time series }
\item{n.windows}{ \code{if(n.windows>1} two sections are defined }
\item{\dots}{ additional graphical parameters }
}
\details{
\code{slider.zoom.plot.ts} plots the original time series and it lets you
select one or two sections of the series by fixing the width(s) and the
starting point(s) of the region(s). Then the section(s) of the series is (are)
plotted separately one below the other.

The first slider defines the width of the section(s).
The second (third) one sets the start of the first (second) section.
}
\value{
 a message about the usage
}
\author{ Hans Peter Wolf }
\seealso{ \code{\link{plot}} }
\examples{
\dontrun{

\end{verbatim}

35
### This example cannot be run by examples() but should be work in an interactive R session

```r
slider.zoom.plot.ts(co2, 2)
```
cat("select type of smoothing a la Tukey and look the result!\n")
}

6.3.1 Test
For testing we use some random numbers.

6.3.2 Help Page

\name{slider.smooth.plot.ts}
\title{interactive Tukey smoothing}
\alias{slider.smooth.plot.ts}
\description{
\code{slider.smooth.plot.ts} computes smooth curves
of a time series plot by Tukey's smoothers.
The kind of smoothing is controlled by a Tcl/Tk widget.
}
\usage{
slider.smooth.plot.ts(x, 
}
\arguments{
 \item{x}{ time series }
 \item{\dots}{ additional graphical parameters }
}
\details{
\code{slider.smooth.plot.ts} draws the time series \code{x}.
The user selects a filter of the set 
\code{c("3RS3R", "3RSS", "3RSR", "3R", "3", "S")}
step by step and the resulting curve is added to the plot.
The selection is performed by pressing a button of the control
widget of \code{slider.smooth.plot.ts}.
The button \code{reset} restarts the smoothing process.
}
\value{
a message about the usage
}
\references{ Tukey, J. W. (1977). Exploratory Data Analysis, Reading
 Massachusetts: Addison-Wesley.
}
\author{ Hans Peter Wolf }
\seealso{ \code{\link{plot}}, \code{\link{smooth}} }
\examples{
\dontrun{
## This example cannot be run by examples() but should be work in an interactive R session
slider.smooth.plot.ts(rnorm(100))
}
7 Regression

7.1 Smoothing by lowess – slider.lowess.plot

The result of the lowess smoother heavily depends on the setting of the parameter \( f \). This parameter controls the smoother span and has to be fixed by the user. The function `slider.lowess.plot` allows the user to select the span and to observe the corresponding line. Furthermore, the number of iterations are set interactively.

```r
slider.lowess.plot<-function(x,y=NULL,...)
{
  # slider function to draw lowess smoother, pwolf 080525
  args<-list(...)
  refresh<-function(...){
    f<-slider(no=1)
    iter<-slider(no=2)
    xy<-lowess(x,y,f=f,iter=iter)
    # plot(x,y,bty="n")
    do.call("plot",c(alist(x,y,bty="n"),args))
    lines(xy)
    title(paste("\n\nlowess: f =",signif(f,4),", iterations =",iter))
    lines(xy)
  }
  slider(refresh,
    c("smoother span","iterations"),
    c(.01,1),c(1,7),c(.01,1),c(2/3,3)
  )
  refresh()
  cat("use slider to select smoother span!\n")
}
```

7.1.1 Test

```r
slider.lowess.plot(cars,col="red")
```

7.1.2 Help Page

```r
\name{slider.lowess.plot}
```
\title{interactive lowess smoothing}
\alias{slider.lowess.plot}
\description{
\code{slider.lowess.plot} computes an xy-plot of the data and
adds LOWESS lines. The smoother span and the number of iterations are selected by sliders.
}
\usage{
slider.lowess.plot(x, y, ...)
}
\arguments{
\item{x}{ data set to be used for plotting or vector of x values }
\item{y}{ vector of y values in case \code{x} is not a matrix }
\item{\dots}{ additional (graphics) parameter settings }
}
\details{
\code{slider.lowess.plot} computes a scatterplot of the data.
Then a LOWESS smoother line is added to the plot.
For more details about the lowess parameters \code{f} and \code{iter}
take a look at the help page of \code{lowess}.
The parameters are set by moving sliders of the control widget. The first slider defines the smoother span \code{f}
and the second one the number of iterations.
}
\value{
\ a message about the usage
}
\references{ for references see help file of \code{lowess} }
\author{ Hans Peter Wolf }
\seealso{ \code{\link{lowess}}, \code{slider} }
\examples{
\dontrun{
## This example cannot be run by examples() but should be work in an interactive R session
slider.lowess.plot(cars)
}
}
\keyword{ iplot }

\section{7.2 Sensitivity of Regression Line – slider.bootstrap.lm.plot}

To understand the sensitivity of a regression curve the function \code{slider.bootstrap.lm.plot}
shows the user a lot of bootstrap regression lines computed on samples of the data points. The number of repetitions, the random seed and the polynomial degree is controlled by sliders.

\begin{verbatim}
slider.bootstrap.lm.plot<-function(x,y=NULL,...)
{
  (check if x is a matrix or x and y are vectors 15)
  args<-list(...) # define
  n<-length(x)
  ind<-order(x); x.orig<-x<-x[ind]; y.orig<-y<-y[ind]
\end{verbatim}
xx<-seq(min(x),max(x),length=100)
# plot(x,y,...)
do.call("plot",c(alist(x,y,bty="n"),args))
abline(lm(y~x),lwd=5)
refresh<-function(...){
  # plot(x,y,...);
do.call("plot",c(alist(x,y,bty="n"),args))
  abline(coefyx<-lm(y~x)$coef, lwd=3)
polytype<-slider(no=1)
  form<-paste(paste(sep="","I(x^",1:polytype,"")",collapse="*"))
  form<-as.formula(paste("y ~",form)); coef<-lm(form)$coef
  yy<-outer(xx,0:polytype,""%%coef; lines(xx,yy,lwd=2)
  B<-slider(no=2); zz<-slider(no=3); set.seed(zz)
  result<-matrix(0,1+polytype,B)
  for(i in 1:B){
    index<-sample(1:n,n,replace=TRUE)
    x<-x.orig[index]; y<-y.orig[index]
    coef<-lm(form)$coef
    yy<-outer(xx,0:polytype,""%%%coef
    lines(xx,yy,lwd=2,col=i,lty=2)
    result[,i]<-coef
  }
  abline(coefyx, lwd=5)
  result<-t(result);
colnames(result)<-c("intercept",paste(sep="","beta: x^",1:polytype))
  print(summary(result))
}
slider(refresh,c("polynomial degree","number repetitions","random seed"),
c(1,1,1),c(5,50,100),c(1,1,1),c(1,10,1))

7.2.1 Test

(test of slider..bootstrap.lm.plot 57) ≡
(define slider.bootstrap.lm.plot 56)
daten<-iris[,2:3]
slider.bootstrap.lm.plot(daten)

7.2.2 Help Page

(define help of slider.bootstrap.lm.plot 58) ≡
\name{slider.bootstrap.lm.plot}
\title{interactive bootstraping for lm}
\alias{slider.bootstrap.lm.plot}
\description{
  \code{slider.bootstrap.lm.plot} computes a scatterplot and
  adds regression curves of samples of the data points.
The number of samples and the degree of the model are
controlled by sliders.

\usage{slider.bootstrap.lm.plot(x, y, ...)}

\arguments{
  \item{x}{ two column matrix or vector of x values if y is used }
  \item{y}{ y values if x is not a matrix }
  \item{\dots}{ additional graphics parameters }
}

\details{\code{slider.bootstrap.lm.plot} draws a scatterplot of the data points and fits a linear model to the data set. Regression curves of samples of the data are then added to the plot. Within a Tcl/Tk control widget the degree of the model, the repetitions and the start of the random seed are set. After modification of a parameter the plot is updated.}

\value{a message about the usage}

\references{~~}

\author{Hans Peter Wolf}

\seealso{\code{\link{plot}}}
if (!missing(obj.name)) {
    if (!missing(obj.value))
        assign(obj.name, obj.value, env = slider.env)
    else obj.value <- get(obj.name, env = slider.env)
    return(obj.value)
}
if (missing(title))
    title <- "slider control widget"
if (missing(sl.names)) {
    sl.defaults <- sl.names <- NULL
}
if (missing(sl.functions))
    sl.functions <- function(...) {
}
# require(tcltk) # now in the depends section of the package
nt <- tktoplevel()
tkm.title(nt, title)
tkm.geometry(nt, "*0+15")
assign("tktop.slider", nt, env = slider.env)
"relax"
for (i in seq(sl.names)) {
"relax"
    eval(parse(text = paste("assign('slider' , i , "", tclVar(sl.defaults[i]), env=slider.env)",
    sep = "")))
    tkpack(fr <- tkframe(nt))
    lab <- tklabel(fr, text = sl.names[i], width = "25")
    sc <- tkscale(fr, from = sl.mins[i], to = sl.maxs[i],
    showvalue = TRUE, resolution = sl.deltas[i], orient = "horiz")
    tkpack(lab, sc, side = "right")
    assign("sc", sc, env = slider.env)
    eval(parse(text = paste("tkconfigure(sc,variable=slider",
    i, ")", sep = "")), env = slider.env)
    sl.fun <- if (length(sl.functions) > 1)
        sl.functions[[i]]
    else sl.functions
    if (!is.function(sl.fun))
        sl.fun <- eval(parse(text = paste("function(...){",
    sl.fun, "}")))
    tkbind(sc, "<ButtonRelease>", sl.fun)
} assign("slider.values.old", sl.defaults, env = slider.env)
tkpack(f.but <- tkframe(nt), fill = "x")
tkpack(tkbutton(f.but, text = "Exit", command = function() tkdestroy(nt)),
    side = "right")
if (!missing(reset.function)) {
    if (!is.function(reset.function))
        reset.function <- eval(parse(text = paste("function(...){",
        reset.function, "}")))
    tkpack(tkbutton(f.but, text = "Reset", command = function() {
        for (i in seq(sl.names)) eval(parse(text = paste("tclvalue(slider",
        i, ")""
    )))
        reset.function()
    })), side = "right")
}
if (missing(but.names))
    but.names <- NULL
for (i in seq(but.names)) {
    but.fun <- if (length(but.functions) > 1)
        but.functions[[i]]
    else but.functions
    if (!is.function(but.fun))
        but.fun <- eval(parse(text = c("function(...){",
        but.fun, "}")))
    tkpack(tkbutton(f.but, text = but.names[i], command = but.fun),
        side = "left")
}
invisible(nt)
}