Implementing interactive computing in an object-oriented environment

Frederic Udina <udina@upf.es>*

*Departament d'Economia i Empresa, Universitat Pompeu Fabra. Ramon Trias Fargas, 25. 08005 Barcelona, SPAIN. Author's work was supported by Spanish DGES grant PB96-0300.

Abstract

Statistical computing when input/output is driven by a Graphical User Interface is considered. A proposal is made for automatic control of computational flow to ensure that only strictly required computations are actually carried on. The computational flow is modeled by a directed graph for implementation in any object-oriented programming language with symbolic manipulation capabilities. A complete implementation example is presented to compute and display frequency based piecewise linear density estimators such as histograms or frequency polygons.

JEL clasification: C10; C13; C43.

Keywords: Histogram; Frequency polygon; Bin width; Object oriented programming; Interactive graphics; Statistical computing.

1 Introduction

Controlling computation flow in classical programs is not a difficult task: some conditional or *case* statements would do the job in most cases. When user interaction is needed, the program prompts for it, then it waits for an answer, processes the response and produces output.

If we consider statistical computing in a graphical user interface (GUI) environment, things are very different. The user can decide at any moment to change any of the quantities involved in the computation. (S)he can also decide on the desired output: what lines, graphs or windows are to be shown or not. In some cases, the user can decide to change a setting while an animation is running. It can be difficult to know in advance which of the intermediate results are really needed and exactly when they will be needed. Some results can be required by the system when refreshing a window, far beyond the programmer's control. Some other results may not be needed at all because the window displaying them is closed or not visible at that moment. The algorithm might work efficiently even when only some intermediate quantity is required, say, by another program that uses it. In general, the whole or some parts of the computation can be expensive and the user wants the fastest possible response to mouse or keyboard actions.

Our goal is thus to discuss a mechanism such that:

- Flow of the computation is automatically driven, so the programmer need not write repetitive parts of the program to control what quantities must be computed or are up to date at a given moment.
- The user has freedom to modify the values under his or her control at any moment.
- Only needed quantities are computed and then stored to avoid recomputation until they must change.

The flow of the computation can be usually described by a directed graph. An arrow going from some quantity to another means that any change in the origin implies that the destination must be updated. Figure 1 show a simple example. The graph in the figure shows that if input **I1** changes, both its descendants **A1** and **A3** need to be updated, but if only output **O2** is to be displayed, then only **A3** must actually be computed. This way changes go forward. But computation goes backward: the actual computation always start when some output is required. Thus the flow goes backwards through the tree looking for the needed intermediate results which may or may not have changed. Note that not every arrow needs to be actually followed backwards: some user controlled flag, like **A6**, can decide whether **A4** or **A5** is needed to update **O2**.

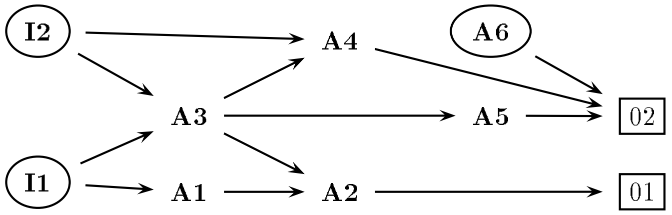

Figure 1: A directed graph to represent the flow of computation. Changes go forward, computation goes backward.

We can consider three types of quantities involved in the computation: input, intermediate and output quantities. Let SI be the set of parameters directly modifiable by the user. This means that at any given moment the user has access to some keyboard command, menu, dialog window, slider, or other interface mechanism which can give a new value to one or several of these *input* parameters. Let SO be the set of *output* variables or devices of the computation process. They can be numbers or arrays representing lines or other parts of a graph. Finally, let SA be the set of all other variables involved in the computation and not included in the previous sets. In the graphs, we show input variables in ovals and output variables in rectangles.

Let G be the graph that describes the flow of the computation, i.e., the set of all pairs (a, b) where a and b are variables in $\mathsf{SI} \cup \mathsf{SA} \cup \mathsf{SO}$ and b depends on a in the sense defined above. Let us write G_0 for the set of variables a such that a pair (a, b) exist in G. Usually, $G_0 = \mathsf{SI} \cup \mathsf{SA}$. Initially, no allowance is made for cycles in the graph.

2 Implementation

Although any computation can be implemented using almost any programming language, it is worth using an object-oriented programming language with symbolic manipulation capabilities.

We use XLISP-STAT, a Lisp dialect with statistical and graphical additions (TIERNEY[1990]). Like in any other object-oriented environment, XLISP-STAT objects have *slots* and *methods* (in terms of non object-oriented programming these would be variables and functions, but slots and methods are owned by the object, and so they can differ from one object to another). Any one of the quantities, say S_i, described above will correspond both to a *slot* and a *method* . The slot can contain either the value for S_i or a symbol like `needs-updating`, flagging that some of the quantities S_i depends on have changed and so S_i needs to be recomputed to be up to date. Note that Lisp variables (and slots) are not typed and thus they can contain either any kind of value or a symbol. The method corresponding to S_i, if called with a value as argument, sets the slot to that value. If called with no arguments, it returns a valid value for S_i: if S_i needs to be updated, the method does that.

The rules to be followed in programming the computation are:

- Carefully define graph G to drive the computation flow, and translate it to a `dependency-tree`: this is a list formed by items in the form $(a, b_1, \ldots, b_k)$ where b_i are all quantities that directly depend on a, and there is one and only one such item for every quantity $a \in G_0$.

- A specific method `:propagate-changes` is used to mark all the slots that depend on the one being changed with the specific symbol `needs-updating`.

- Changes to a slot are always done through the corresponding method. This method should call `:propagate-changes`.

- The same method, when called with no arguments, returns the value for the slot unless it is `needs-updating`, in which case it is recomputed, stored and returned.

This way, all the variables of interest are contained in slots of an object, and they will always be accessed by means of an accessor method similar to the one in Figure 2.

When the method `:var-name` is called without arguments, it checks the current value of the slot `var-name`. If the special symbol `needs-updating` is found there, some routine to recompute the variable value is run and the result is stored and returned. Otherwise, just the value found is returned.

```
(defmeth some-object :var-name
                     (&optional (value nil value-set))
  (if value-set ;then store it in 'var-name slot
      (progn
         ;;set all dependants to 'needs-updating
         (send self :propagate-changes 'var-name)
         ;; set the given value
         (slot-value 'var-name value))
      ;;not value-set. If needed, it must be recomputed
      (when (eq 'needs-updating (slot-value 'var-name))
            ;; set the slot value to the result
            ;; of the updating computation
            (slot-value 'var-name
                        (call-to-compute-it))))
  ;;in any case, we return the slot contents
  (slot-value 'var-name))
```

Figure 2: The standard method to access a slot and propagate the changes. Only the third line from the bottom must be tailored for each variable.

If the method is called with an argument, this value is stored in the slot. As a side effect, all the variables that depend on the variable being changed are marked with `needs-updating` by a `:propagate-changes` method such as the one in Figure 3. In this method we assume that the dependence tree is not very big and that the propagation mechanism is fast. Otherwise, some time can be saved by checking if the slot is already marked as `needs-updating` and skipping its subtree in that case.

A `dependence-tree` as described before is needed to propagate changes. For the example in Figure 1, the tree will be installed in its own slot by a call as shown in Figure 4

User input and graphical output

User input via a graphical interface will arrive through the keyboard, a dialog window or any other GUI mechanism, at any time, requesting a change to a quantity or parameter. To introduce the change into the computation flow a call such as

```
(defmeth some-object :propagate-changes (symb)
"Will mark as changed all the slots that depend on SYMB
 following the dependence tree"
  (let ((seq (find symb (slot-value 'dependence-tree)
                        :key #'first)))
      (when seq ; if found, seq is the list of items
            (mapcar #'(lambda
                        (slo)
                        (slot-value slo 'needs-updating)
                        (send self :propagate-changes slo))
                     (cdr seq))))) ; and mark it all
```

Figure 3: The method to propagate changes.

```
(send some-object :slot-value 'dependence-tree
  '((i1 a1 a3)
    (i2 a3 a4)
    (a1 a2)
    (a2 o1)
    (a3 a4 a5)
    (a5 o2)
    (a4 o2)
    (a6 o2)))
```

Figure 4: The call to install the dependence tree in its slot.

`(send my-obj :i1 a-new-value)`

is used. This way all the dependent variables will be marked as out of date. In most cases this will be followed by a call to a redrawing method, usually `:redraw-content`. When something (an output routine or some part of the program answering to a `:redraw-content` call) asks for `o1` by using

`(send my-obj :o1),`

only `a1`, `a2` and `a3` will be recomputed, while `a4` and `a5` will remain with the `'needs-updating` label until `o2` is requested.

Note that some of the output variables can be boolean flags that are false (or `needs-updating`) if the window is out of date and true if everything has already been updated. In such a case, simply asking for the value of the slot can return true if everything is OK, and nothing would be recomputed, but if it's not true, it will be `needs-updating` and all the information needed to draw the window will be recomputed.

Finally, the program needs to provide for reaction to requests from the window system. If the window gets uncovered, for example, the window manager will send a `:redraw-content` request. This is the same request the program should send to be sure that all graphic output is up to date. The `:redraw-content` method of the involved window(s) should then be modified in the following way: before calling the standard method, check if the window must be updated or not. If yes, call the appropiate slots to recompute it and produce the effective graphical output.

3 A complete example

As an example, we analyze `fde.lsp`, a lisp program that implements rich histograms as XLISP-STAT objects.[1]

After loading the code file with

`(load "fde")`

a rich histogram window is created by

`(make-histogram data)`

where *data* is a list of numbers. The call `(fde-demo)` can be used for demo purposes. It will create a `fde` object with data from SIMONOFF[1997] giving the duration of the Old Faithful geyser eruptions in August 1978

[1] The full code and some examples of using it is available in `http://libiya.upf.es/soft`

and 1979 (originally collected by Sandy Weisberg). The interaction with the window is usually done through the window menu (see Figure 5) or via keyboard shortcuts (see Figure 6).

Mouse mode	Either 'Show coordinates' or 'Zoom in'. To zoom-out again use 'Rescale plot'.
Rescale plot	Use a new scale in the plot so all lines are visible.
Options	Standard XLISP-STAT Options menu.
Show info	Show some information about the data set and the histogram cells.
Stability Index	Compute the Simonoff-Udina stability index.
Stability index plot	Compute the index for a range of bin widths and show a plot. Indices are shown for the edge frequency polygon and the regular histogram.
Assess the Index value	Assess the stability of the histogram through a bootstrap-like method.
Bin control	Open a dialog with two sliders to control bin width and anchor shift.
What to show	Open a dialog to choose what estimators are to be shown in the window. Also data, simple box-plot and density shadow are available.
Animate	Perform animation of the anchor shifting effect.

Figure 5: Menu items available in a FDE window.

Several frequency based estimators are supported: histograms, hollow histograms, frequency polygon (see SCOTT[1992]), edge frequency polygon (JONES *et al.*[1998]), piecewise linear estimator (see BEIRLANT, BERLINET AND GYÖRFI[1998]) are all briefly described in Section 4. Using a menu item, the user decides which estimators (s)he wants to see and the choice is stored in `what-to-show`. Then, when the window is to be redrawn, some `bin-counts` are needed and computed from the data, using the default

`+/-`	Increase/Decrease the bin width
`l/r`	Move the anchor to the left/right
`a`	Adjust the axis scales to the current state
`d`	Show/hide data points

Figure 6: Keyboard shortcuts in a FDE window.

`bin-width` and `anchor-point`. The user can then change these values using a slider accessible from a menu item again, or using the keyboard. The effect of changing the anchor point (see SIMONOFF AND UDINA[1997] and Section 4) can be easily visualized through an animation (available from the menu). Note that while the animation is running, all the input parameters can be changed using the keyboard or any menu item. The animation is implemented easily using the `:do-idle` feature of XLISP-STAT graphic windows (see source file). This means that while the animation is running, the user has access to the menu or the keyboard shortcuts for tuning up any of the animation parameters.

All quantities and flags involved are represented as instance slots in the main object, a `fde-proto` descendant, and are listed in Figure 7. The dependence tree is stored in a shared slot because all instances use the same tree. The graphics window is actually a separated object. The reason for this is to make it easier for a single object to perform all computations, while the window object is created only if display is needed. More detail can be found in the source listing.

The computation flow is shown in a directed graph in Figure 8. The input quantities are shown in an oval frame, the output ones in a rectangular frame. Label `WuD` is a flag which shows if the window is up to date or otherwise needs updating. When a variable is changed, the change is transmitted following the arrows in the graph. When an output is required, computation flow goes backward through the graph, searching just for the required values to complete the computation. Note that not every arrow is followed backwards: slot `what-to-show` is looked for to know what lines really need to be computed.

In the actual implementation a couple of macros are used to simplify method writing and reading. The macro

`defmeth-fde-proto-accessor-changes`

(see Figure 9) produces a defmeth call very similar to the one shown in Fig-

`data`
`data-summary`
`scale-estimate`
`x-range`
	The data and some statistics to describe it, the range to be displayed.

`bin-width` `bw-ends`	Bin width and the minimum and maximum value for it.

`bin-edges`
`bin-counts`
`long-bin-counts`
`half-bin-counts`
	The bin edges list and several ways to count the data in each bin.

`anchor-base` `anchor-shift`	To determine the anchor, the position of the first edge.

`boxplot-lines`
`...`
`edgpoly-lines`
	Several kinds of lines or graphical output to be displayed, depending on user choice.

`stability-index`	An index to measure shape change for the histogram.
`density` `density-lines`	A density to be matched to or compared with.
`what-to-show` `y-scale`	A list describing what elements must appear in the window and a selector for the vertical scale to use.
`window-up-to-date`	A flag showing if the window contents are up to date.

Figure 7: The quantities involved in histogram computation.

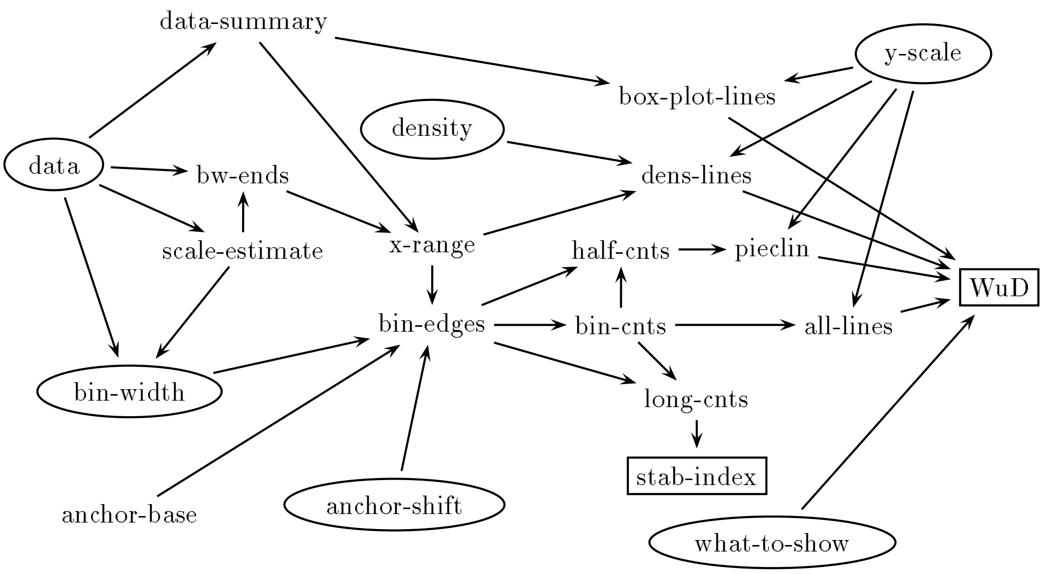

Figure 8: A directed graph to represent the flow of computation. Changes go forward, computation goes backward. Input quantities are shown in oval frames, the output ones in rectangular frames. `WuD` stands for the `Window up to date` flag.

ure 2. There is a second macro in the source file used to produce accessors

```
(defmacro defmeth-fde-proto-accessor-changes
          (keyword help-string computation)
  (let ((sym (find-symbol (symbol-name keyword))))
    `(defmeth fde-proto
       ,keyword
       (&optional (value nil vset))
       ,help-string
       (if vset (progn
                  (send self :mark-changed ',sym)
                  (setf (slot-value ',sym) value))
         (when (eq 'must-compute (slot-value ',sym))
           (setf (slot-value ',sym) ,computation)))
       (slot-value ',sym))))
```

Figure 9: The macro used to produce standard accessors. The relevant argument is `computation`, a list of lisp expressions to be evaluated to compute the value of the quantity corresponding to symbol `symb`.

that admit an optional keyword argument `:draw` defaulting to nil. If it is non-nil, the change in the slot will force redrawing the object's window (provided it is open).

3.1 Using `fde` objects from Lisp

Any Lisp program can use `fde` objects in a very efficient way, given how they are implemented. As an example, a sequence call such as the following would give the desired results computing just the required quantities but nothing else.

```
(require "fde")
(def myhist (make-histogram my-data :show nil))
(def stabind (send myhist :stability-index))
(def mylins (send myhist :edgpoly-lines))
(undef 'myhist)
```

After running this code, the variable `stabind` will contain the stability index of the given data histogram (for the default bin-width) and frequency polygon. `mylins` will contain the segments to build the edge frequency estimator. Use of the last line is recommended to send used memory space which is no longer needed to the garbage collector.

The construction function `make-histogram` has mandatory argument `data`. It can be followed by any of the following keyword arguments in the usual keyword/value pair form.

`:title`
`:bin-width`
`:anchor-base`
`:anchor-shift`
`:x-range`
`:y-scale`
`:density`
`:what-to-show`
	These optional keyword arguments have the same meaning as the corresponding methods, see next listing.

`:show`	`t` if a window must be created to show the estimators. Default is `t`.
`:debug`	`t` if debugging information is to be printed. Default is `nil`.

Here follows the list of the `fde-proto` methods that could be of interest for a Lisp program. Some of these methods have the optional keyword argument `:draw`. It defaults to `nil`. If a non-nil value is given, window redrawing is forced.

`:isnew (...)`

Creator for `fde` object. Creation by using the function `make-histogram` is recommended. Argumentss are similar to those described above for that function.

`:title (&optional title)`

Gets or sets a title string for the object and the window.

`:data (&optional data & key (draw nil))`

Gets current data or set new data.

`:bin-width (&optional value & key (draw nil))`

Gets or sets bin width. If `:draw` is not nil, force redrawing.

`:data-summary`

Returns a list with the data five numbers (min, quartiles and max), data size, mean and standard deviation.

`:scale-estimate`

Computes a robust estimate for the scale following JANSSEN *et al.*[1995]. It's useful to replace the standard deviation in multimodal densities, for example.

`:x-range (&optional interval & key (draw nil))`

Sets or gets the estimation range. If an interval is given, it must be a two numbers list, covering the whole data range.

`:bw-ends (&optional interval)`

Sets or gets the interval where the bin width is confined. If an interval is given, it must be a two numbers list.

`:anchor-base`

Sets or gets the reference for the anchor: the position of the first bin when `anchor-shift` is zero. If it is `'data-based`, the minimum of the data will be used. It can also be a number, the known minimum value possible for data values.

`:anchor-shift (&optional value &key (draw nil))`

Sets or gets a value in [0,1) that is the fraction of the bin width to be subtracted from the anchor-base to put the first bin edge there.

`:stability-index`

Computes the stability index for the histogram as defined in SIMONOFF AND UDINA[1997] (see Section 4).

`:assess-index-value`

Run a bootstrap-like simulation to assess the current stability index. See details in Section 4.

`:bin-edges`

Sets or gets a list with the bin edges position. Bin edges must be equally spaced.

`:bin-counts`

Returns a list with the counts for the bins.

`:bin-frequencies`

Returns a list with the bin's relative frequencies.

`:density (&optional (value nil vset) (numargs 1))`

Gets or sets a density function to be drawn in the window. The function can have 1, 2, or 3 arguments, first is x value, second is data mean, third is data standard deviation. If argument is the symbol `'normal`, an adjusted normal density is installed.

`:y-scale (&optional value &key (draw nil))`

Gets or sets the vertical axis scale. Can be any of `'density`, `'frequency` or `'count`.

`:histo-lines :hohisto-lines :piecelin-lines`
`:fpoly-lines :edgpoly-lines`

These methods return a list of pairs (x, y), the points to be joined to draw lines representing respectively a regular histogram, a hollow histogram, a piecewise linear estimator, a frequency polygon or an edge frequency polygon. The result can be put as argument in the `:add-lines` method of a `graph-proto`, except for `:piece-lines`. In this case a list of lines is returned.

`:density-lines :boxplot-lines`

These methods return a list of (x, y) pairs, the points to be joined to draw lines representing a normal density fitted to data, or a simple box-plot.

`:what-to-show (&optional list-of-estimators)`

Sets or gets a list of the estimators to be shown in the window. `list-of-estimators` must be a sublist of `(:histo-lines :hohisto-lines :fpoly-lines :edgpoly-lines :piecelin-lines :data :boxplot-lines :density-lines)`.

`:add-to-show (what &key (remove nil) (draw t))`

Adds to (or removes from when `:remove` is not-nil) the given symbols to/from the slot `what-to-show`, see above.

`:have-window`

Creates a window to display the estimators, if it does not already exist.

`:to-window (&rest args)`

With no args, returns the window object where estimators are being displayed. It is a `graph-proto` descendant. Supplied args will be sent to it.

`:redraw-window`

The window will be redrawn after updating all the slots that contain `needs-updating`.

4 Frequency based estimators, brief summary

Let $\{x_1, \ldots, x_N\}$ be a set of i.i.d. data values with common density function $f(x)$. A fixed bin width histogram is determined by an anchor point b_1 ($b_1 \leq x_{(1)}$ and a bin-width h, ($h \geq 0$). The bin edges are then

$$b_j = b_1 + (j-1) * h, \qquad j = 1, \ldots, K$$

where K is such that $b_K > x_{(N)}$. The histogram estimate of the underlying density within a given bin is

$$\hat{f}(x) = \frac{n_j}{Nh}, \qquad x \in [b_j, b_{j+1}),$$

where n_j is the number of observations falling in the j^{th} bin $[b_j, b_{j+1})$. When needed, we consider $n_0 = n_{K+i} \equiv 0, i > 0$ as the counts of the adjacent empty bins.

The simplest improvement to the histogram is the frequency polygon, the linear interpolant of histogram heights at the bin centers. The frequency polygon has the form

$$\hat{f}_{FP}(x) = (Nh)^{-1}\left[\frac{n_i + n_{i+1}}{2} + \left(\frac{n_{i+1} - n_i}{h}\right)(x - b_{i+1})\right],$$

$$x \in [b_{i+1} - h/2, b_{i+1} + h/2], i = 0, \ldots, K.$$

An alternative to the frequency polygon, the edge frequency polygon, was introduced by JONES *et al.*[1998], and has the form

$$\hat{f}_{EF}(x) = (2Nh)^{-1}\left[\frac{n_{i+1} + 2n_i + n_{i-1}}{2} + \left(\frac{n_{i+1} - n_{i-1}}{h}\right)(x - b_{i+1} + h/2)\right],$$

$$x \in [b_i, b_{i+1}], \quad i = 1, \ldots, K.$$

This estimate is the linear interpolant of the averages of two adjacent bin heights at right bin edges, and will therefore be called the average frequency polygon here.

A more complicated estimator, the linearly binned frequency polygon, replaces the cell counts n_i in (5.1) with linear bin counts

$$\ell_i = \sum_{i=1}^{n} (1 - h^{-1}|x_j - b_i - h/2|)_+$$

where + subscript denotes positive part (JONES AND LOTWICK[1983], JONES[1989]). This can be seen as each data point splitting its unit mass between the two nearest bin centers, in inverse proportion to the distances to them. The estimator is a discretized kernel estimator with triangular kernel function JONES[1989], and can achieve 5.8% lower optimal AMISE than $\hat{f}_{FP}$.

To avoid the difficulty of extending these estimators to higher dimensions, BEIRLANT, BERLINET AND GYÖRFI[1998] introduced the piecewise linear estimator: it is obtained by dividing each bin in two halves, and joining the frequencies of each half at the half-bin centers by a line, the line covering the whole bin. Despite its good theoretical properties, it produces nasty results for small samples: estimates are not necessarily positive or continuous.

Bin width

The bin width h acts as a smoothing parameter, as it controls the degree of smoothness of the estimate, with larger values of h resulting in histograms with a smoother appearance. All density estimators include some form of smoothing parameter, and a good deal of research has focused on choosing it for different estimators, often based on an assessment of accuracy using the integrated squared error of the estimator,

$$ISE = \int_{-\infty}^{\infty} [\hat{f}(x) - f(x)]^2 dx,$$

and its expected value, mean integrated squared error ($MISE$).

The frequency polygon is superior to the histogram in terms of $MISE$, achieving the rate $AMISE = O(N^{-4/5})$ (taking $h = O(N^{-1/5})$, rather than the optimal $O(N^{-1/3})$ asymptotic rate for histograms), and this improved accuracy carries over to small samples (SIMONOFF AND HURVICH[1993]), but its appearance (in terms of modes, bumps and dips) is identical to that of the histogram, and it therefore has the identical anchor stability properties for given h. The edge frequency polygon can achieve 11.5% smaller optimal $AMISE$.

Automatic (data based) choice of bin width has been studied by many authors. We refer to WAND[1997] for a complete study of plug-in type selectors. In `fde` we only implement the basic *normal reference* bin as defined by SCOTT[1979]:

$$h_{NR} = 3.5\hat{\sigma}^{-1/3}$$

where we use as $\hat{\sigma}$ the scale estimate defined by JANSSEN *et al.*[1995].

Anchor position and shifting

Asymptotic analysis shows that anchor position of a histogram has a lower order asymptotic effect on $MISE$ compared with the bin width, and can therefore be ignored. Despite this, from a practical point of view, shifting the bin edges by changing the anchor position can have an important effect on the appearance of the resultant histogram for finite samples, changing the uni/multimodality or symmetry aspect, for example. Many authors have focused on this as one of the biggest drawbacks of using the histogram (see, e.g., FISHER[1989]; HÄRDLE[1991], Section 1.4; HÄRDLE AND SCOTT[1992]; IZENMAN[1991]; SAMIUDDIN, JONES AND EL-SAYYAD[1993]; SCOTT[1992], Section 4.3; SILVERMAN[1986], Section 2.2).

By *shifting the anchor* or the bin edges we mean replacing the fixed initial point b_1 by $a_b - s * h$ where a_b is the anchor base (which can be any number $a_b \leq x_{(1)}$, often taken as $a_b = x_{(1)}$) and $s \in [0, 1)$ is the shift factor.

The literature contains little systematic examination of anchor position effects. In a Monte Carlo simulation study, SIMONOFF[1995] found that the ISE (which can be termed quantitative accuracy) of the histogram estimate is insensitive to anchor position, unless a discontinuity (or near discontinuity) of the density is crossed by a bin (that is, if the discontinuity occurs inside a bin rather than at a bin edge; see also SCOTT[1992], pp. 65–66). On the other hand, the appearance of histograms (as quantified by the number of observed modes, a measure of qualitative accuracy) can be very sensitive to anchor position. SCOTT[1992] (p. 111), in the context of the frequency polygon, noted that the anchor position can be thought of as a nuisance parameter, and suggested choosing it for a given bin width to make the resultant estimate as smooth as possible.

Stability index

SIMONOFF AND UDINA[1997] agree that the anchor position is a nuisance parameter. Rather than pick a particular (arbitrary) value for that parameter, however, they propose constructing a measure to assess how sensitive the appearance of the histogram is to *any* possible choice. That is, the measure is a function of the data and h, and not of any particular anchor choice. This is the *stability index* implemented in `fde`.

The stability index ranges in $[0, 1]$ and can be computed for any of the bin frequencies based estimators described so far.

If the stability index is high, i.e. close to 1, the appearance of the histogram does not change very much for a given h as the anchor position changes, and thus the analyst is free to choose whatever anchor (s)he wishes, without worrying about the effect of that choice. However, if the stability index is low, i.e. close to 0, the appearance is sensitive to anchor position, the impressions gained from a histogram using any particular choice cannot be trusted since a different choice could give a very different impression.

It is important to note that the stability measure is **not** a measure of the accuracy of the histogram as an estimate of the true density f, but rather of the consistency of the representation of that density as anchor position is changed. That is, a stable bin width is not necessarily one that gives an accurate impression of the true density, but rather one where the impressions do not change very much with anchor position. Thus, a data analyst would use the index as a secondary tool, after first choosing the bin width to provide an accurate impression of the true density (based on ISE, or some other measure). If the chosen bin width is stable, an anchor position is chosen, and the estimate is constructed.

If the bin width is unstable, however, any choice of anchor position is dangerous. Instead, a different, more stable, bin width should be chosen. If the new bin width is close to the original one, it is likely that the histogram is as accurate as one based on the original choice, and little is lost; but if there are no stable choices near the original choice, it is likely that no histogram will be satisfactory, and a different density estimator should be used.

In `fde` we provide a menu item to compute the index for a range of bin widths. This gives the analyst a useful tool for inspecting the graphical output and to choose a bin width that is both reasonable from the data analysis point of view and has a good (high) index value.

As a general rule, stability of the considered estimators (for a given bin width) is higher for the edge frequency estimator, less for the frequency polygon and the histogram (which have the same appearance and thus the same stability index) and least for the piecewise linear estimate.

We refer to SIMONOFF AND UDINA[1997] for the definition of the stability index. It consists in measuring the shape of each of the many histograms obtained by shifting the anchor position and then comparing the (dis)similarity of the numbers obtained by a Gini type index (see MARSHALL AND OLKIN[1979]).

References

BEIRLANT, J.; BERLINET, A.; GYÖRFI, L. (1998) On Piecewise Linear Density Estimators. *Statistica Nederlandica. to appear*.

FISHER, N.I. (1989) Smoothing a sample of circular data. *J. Structural Geology*. **11**, 775-778.

HÄRDLE W. (1991) *Smoothing techniques with Implementation in S*. Springer–Verlag, New York.

HÄRDLE, W.; SCOTT, D. W. (1992) Smoothing by Weighted Averaging of Rounded Points. *Computational Statistics*. **7**, 97–128.

IZENMAN, A. J. (1991) Recent Development in Nonparametric Density Estimation. *Journal of the American Statistical Association*. **86**, 205-224.

JANSSEN, P.; MARRON, J.S.; VERAVERBEKE, N.; SARLE, W. (1995) Scale measures for bandwidth selection. *Journal of Nonparametric Statistics*. **5**, 359-380.

JONES, M. C. (1989) Discretized and interpolated Kernel Density Estimates. *Journal of the American Statistical Association*. **84**, 733-740.

JONES, M. C.; LOTWICK, H.W. (1983) On the errors involved in computing the empirical characteristic function. *Journal of Statistical Computation and Simulation*. **17**, 133-149.

JONES, M. C.; SAMIUDDIN, M.; AL-HARBEY, A.H.; MAATOUK, T.A.H. (1998) The Edge Frequency Polygon. *Biometrika*. **85**, 235-9.

MARSHALL, A.W.; OLKIN, I. (1979) *Inequalities: theory of majorization and its applications*. Academic Press, New York.

SAMIUDDIN, M.; JONES, M.C.; EL-SAYYAD, G.M. (1993) On bin–based density estimation. *Journal of Statistical Computation and Simulation*. **47**, 241–252.

SCOTT, D. W. (1979) On optimal and data-based histograms. *Biometrika*. **66**, 605–610.

SCOTT, D. W. (1992) *Multivariate Density Estimation: theory, practice and visualization*. John Wiley, New York.

SILVERMAN, B. W. (1986) *Density Estimation for Statistics and Data Analysis*. Chapman and Hall, London, 1986.

SIMONOFF, J. S. (1995) The anchor position of histograms and frequency polygons: quantitative and qualitative smoothing. *Communications in Statistics – Simulation and Computation*. **24**, 691–710.

SIMONOFF, J. S. (1997) *Smoothing methods in Statistics*. Springer-Verlag, New York.

SIMONOFF, J. S.; HURVICH, C. M. (1993) A Study of the Effectiveness of simple density estimation methods. *Computational Statistics*. **8**, 259–278.

SIMONOFF, J. S.; UDINA, F. (1997) Measuring the stability of histogram appearance when the anchor position is changed. *Computational Statistics and Data Analysis*. **23**, 335–353.

TIERNEY, L. (1990) *LISP-STAT: An Object Oriented Environment for Statistical Computing and Dynamic Graphics*. John Wiley and Sons, New York.

WAND, M. P. (1997) Data-based Choice of Histogram Bin Width. *The American Statistician*. **51**, 59–64.