Some Dynamic Graphics Examples

Some Background

Dynamic graphs usually have some of the following characteristics:

- high level of interaction
- motion
- more than two dimensional data
- require high performance graphics display hardware

Some of the basic techniques that appear to be useful are

- animation
  - controlled interactively
  - controlled by a program
- direct interaction and manipulation
  - linking and brushing
- rotation

Many ideas are described in the papers in the book edited by Cleveland and McGill (1988).

Research on the effective use of dynamic graphics is just beginning.

Some of the questions to keep in mind are

- How should dynamic graphs be used?
- What should you plot?
- How do you interpret dynamic images?
- What techniques are useful, and for which problems?

Even in simple two-dimensional graphics there are difficult open issues.

Answering these questions will take time.

We can make a start by learning how to try out some of the more promising dynamic graphical ideas, and variations on these ideas.

Some Examples of Animation

The basic idea in animation is to display a sequence of views rapidly enough and smoothly enough to

- create an illusion of motion
- allow the change in related elements to be tracked visually

The particular image is determined by the values of one or more parameters.

- For a small number of parameters, (1 or 2), the values can be controlled directly, e.g. using scroll bars.
- For more parameters, automated methods of exploring the parameter space are useful.

Dynamic Box-Cox Plots

Fowlkes's implementation of dynamic transformations in the late 1960's was one of the earliest examples of dynamic statistical graphics.

Start by defining a function to compute the transformation and scale the data to the unit interval:

```
(defun bc (x p)
  (let* ((bcx (if (< (abs p) .0001)
                  (log x)
                  (/ (^ x p) p)))
         (min (min bcx))
         (max (max bcx)))
    (/ (- bcx min) (- max min))))
```

Next, construct an ordered sample and a set of approximate expected normal order statistics:

```
(setf x (sort-data precipitation))
(setf nq (normal-quant (/ (iseq 1 30) 31)))
```

We can construct an initial plot as

```
(setf p (plot-points nq (bc x 1)))
```

To change the plot content to reflect a square-root transformation, we can use the `:clear` and `:add-points` messages:

```
(send p :clear)
(send p :add-points nq (bc x 0.5))
```

To achieve a smooth transition, you can give the `:clear` message a keyword argument `:draw` with value `nil`:

```
(send p :clear :draw nil)
(send p :add-points nq (bc x 0.0))
```

These steps can be built into a `:change-power` method:

```
(defmeth p :change-power (pow)
  (send self :clear :draw nil)
  (send self :add-points nq (bc x pow)))
```

To view a range of powers, you can use a `dolist` loop

```
(dolist (pow (rseq -1 2 20))
  (send p :change-power pow))
```

or you can use one of two *modeless* dialogs to get finer control over the animation:

```
(sequence-slider-dialog (rseq -1 2 21) :action
  #'(lambda (pow) (send p :change-power pow)))
```

or

```
(interval-slider-dialog '(-1 2)
  :points 20
  :action
  #'(lambda (pow) (send p :change-power pow)))
```

To avoid losing color, symbol or state information in the points, we can change the coordinates of the points in the plot instead of replacing the points.

To preserve the original data ordering, we need to base the x values on ranks instead of ordering the data

```
(let ((ranks (rank precipitation)))
  (setf nq (normal-quant (/ (+ ranks 1) 31))))
```

and use these to construct our plot:

```
(setf p (plot-points nq (bc precipitation 1)))
```

To change the coordinates, we need a list of the indices of all points.

To avoid generating this list every time, we can store it in a slot:

```
(send p :add-slot 'indices (iseq 30))
(defmeth p :indices () (slot-value 'indices))
```

The method for changing the power now becomes

```
(defmeth p :change-power (pow)
  (send self :point-coordinate
        1
        (send self :indices)
        (bc precipitation pow))
  (send self  :redraw-content))
```

and the slider is again set up by

```
(interval-slider-dialog '(-1 2)
  :points 20
  :action
  #'(lambda (pow) (send p :change-power pow)))
```

The same idea can be used with a regression model.

To transform the dependent variable and scale the residuals, define the functions

```
(defun bcr (x p)
  (if (< (abs p) 0.0001)
      (log x)
      (/ (^ x p) p)))
```

and

```
(defun sc (x)
  (let ((min (min x))
        (max (max x)))
    (/ (- x min) (- max min))))
```

Let's use the stack loss data as an illustration.

A set of approximate expected normal order statistics and a set of indices are set up as

```
(setf nqr
      (let ((n (length loss)))
        (normal-quant (/ (iseq 1 n) (+ 1 n)))))

(setf idx (iseq (length loss)))
```

The regression model and initial plot are set up by

```
(setf m (regression-model (list air conc temp)
                          loss))

(setf p (let ((r (send m :residuals)))
          (plot-points (select nqr (rank r))
                       (sc r))))
```

The new `:change-power` method transforms the dependent variable in the regression model and then changes the plot using the new residuals:

```
(defmeth p :change-power (pow)
  (send m :y (bcr loss pow))
  (let* ((r (send m :residuals))
         (r-nqr (select nqr (rank r))))
    (send self :point-coordinate 0 idx r-nqr)
    (send self :point-coordinate 1 idx (sc r))
    (send self :redraw-content)))
```

Again, a slider for controlling the animation is set up by

```
(interval-slider-dialog '(-1 2)
  :action
  #'(lambda (pow) (send p :change-power pow)))
```

Since the point indices in the plot correspond to the indices in the regression, it is possible to track the positions of the residuals for groups of observations as the power is changed.

Density Estimation

Choosing a bandwidth for a kernel density estimator is a difficult problem.

It may help to have an animation that shows the effect of changes in the bandwidth on the estimate.

An initial plot can be set up as

```
(setf w (plot-lines (kernel-dens precipitation
                                 :width 1)))
```

The initial bandwidth can be stored in a slot

```
(send w :add-slot 'kernel-width 1)
```

and the accessor method for the slot can adjust the plot when the width is changed:

```
(defmeth w :kernel-width (&optional width)
  (when width
        (setf (slot-value 'kernel-width) width)
        (send self :set-lines))
  (slot-value 'kernel-width))
```

It may also be useful to store the data in a slot and provide an accessor method:

```
(send w :add-slot 'kernel-data precipitation)

(defmeth w :kernel-data ()
  (slot-value 'kernel-data))
```

The method for changing the plot is

```
(defmeth w :set-lines ()
  (let ((width (send self :kernel-width))
        (data (send self :kernel-data)))
    (send self :clear-lines :draw nil)
    (send self :add-lines
          (kernel-dens data :width width))))
```

and a slider for controlling the width is given by

```
(interval-slider-dialog '(.25 1.5)
  :action
  #'(lambda (s) (send w :kernel-width s)))
```

Assessing Variability by Animation

One way to get a feeling for the uncertainty in a density estimate is to re-sample the data with replacement, i.e. to look at bootstrap samples, and see how the estimate changes.

We can do this by defining a `:do-idle` method:

```
(defmeth w :do-idle ()
  (setf (slot-value 'kernel-data)
        (sample precipitation 30 :replace t))
  (send self :set-lines))
```

We can turn the animation on and off with the `:idle-on` message, but it is better to use a menu item.

We have already used such an item once; since we may need one again, it is probably worth constructing a prototype:

```
(defproto run-item-proto
          '(graph) () menu-item-proto)

(send run-item-proto :key #\R)

(defmeth run-item-proto :isnew (graph)
  (call-next-method "Run")
  (setf (slot-value 'graph) graph))

(defmeth run-item-proto :update ()
  (send self :mark
        (send (slot-value 'graph) :idle-on)))

(defmeth run-item-proto :do-action ()
  (let ((graph (slot-value 'graph)))
    (send graph :idle-on
          (not (send graph :idle-on)))))
```

We can now add a run item to the menu by

```
(send (send w :menu) :append-items
      (send run-item-proto :new w))
```

A problem with this version is that the x values at which the density estimate is evaluated change with the sample.

To avoid this, we can add another slot and accessor method,

```
(send w :add-slot
      'xvals
      (rseq (min precipitation)
            (max precipitation)
            30))

(defmeth w :xvals () (slot-value 'xvals))
```

and change the `:set-lines` method to use these values:

```
(defmeth w :set-lines ()
  (let ((width (send self :kernel-width))
        (xvals (send self :xvals)))
    (send self :clear-lines :draw nil)
    (send self :add-lines
          (kernel-dens (send self :kernel-data)
                       :width width
                       :xvals xvals))))
```

Another possible problem is that the curve may jump around too much to make it easy to watch.

We can reduce this jumping by changing one observation at a time instead of changing all at once:

```
(send w :slot-value
      'kernel-data (copy-list precipitation))

(defmeth w :do-idle ()
  (let ((d (slot-value 'kernel-data))
        (i (random 30))
        (j (random 30)))
    (setf (select d i)
          (select precipitation j))
    (send self :set-lines)))
```

A similar idea may be useful for estimated survival curves.

Suppose `s` is a survival distribution object.

A plot of the Fleming-Harrington estimator is

```
(setf p
      (let ((d (send s :num-deaths))
            (r (send s :num-at-risk))
            (udt (send s :death-times)))
        (plot-lines
         (make-steps
          udt
          (exp (- (cumsum (/ d r))))))))
```

We can put a copy of the estimated hazard increments in a slot:

```
(send p :add-slot
      'del-hazard
      (copy-list (/ (send s :num-deaths)
                    (send s :num-at-risk))))

(defmeth p :del-hazard ()
  (slot-value 'del-hazard))
```

For certain prior distributions, a Bayesian analysis produces approximately a gamma posterior distribution for the hazard increments.

We can use this distribution to animate the plot:

```
(defmeth p :do-idle ()
  (let* ((udt (send s :death-times))
         (d (send s :num-deaths))
         (r (send s :num-at-risk))
         (dh (send self :del-hazard))
         (n (length dh))
         (i (random n))
         (di (select d i))
         (ri (select r i)))
    (setf (select dh i)
          (/ (first (gamma-rand 1 di)) ri))
    (send self :clear-lines :draw nil)
    (send self :add-lines
          (make-steps udt
                      (exp (- (cumsum dh)))))))
```

A run item lets us control the animation:

```
(send (send p :menu) :append-items
      (send run-item-proto :new p))
```

Another way to think of this animation is as a simulation from an approximation to the sampling distribution.

Modifying Mouse Responses

Modifying Selection and Brushing

We have already seen several examples of using the standard selecting and brushing modes.

The series of messages used to implement these modes lets us modify what happens when the brush is moved or points are selected.

The simplest way to build on the standard modes is to modify the `:adjust-screen` method.

This message is sent each time the set of selected or highlighted points is changed with the mouse.

As an example, if we are using selection or brushing to examine a bivariate conditional distribution, it may be useful to augment the bivariate plot with a smoother.

Suppose we have a histogram of **hardness** and a scatterplot of **abrasion-loss** against **tensile-strength** for the abrasion loss data

```
(setf h (histogram hardness))

(setf p (plot-points tensile-strength
                     abrasion-loss))
```

The method

```
(defmeth p :adjust-screen ()
  (call-next-method)
  (let ((i (union
            (send self :points-selected)
            (send self :points-hilited))))
    (send self :clear-lines :draw nil)
    (if (< 1 (length i))
        (let ((x (select tensile-strength i))
              (y (select abrasion-loss i)))
          (send self :add-lines
                (kernel-smooth x y)))
        (send self :redraw-content))))
```

adds a kernel smooth to the plot if at least two points are selected or hilited.

To have control over the smoother bandwidth, we can add a slot and accessor method, modify the `:adjust-screen` method, and add a slider:

```
(send p :add-slot 'kernel-width 50)

(defmeth p :kernel-width (&optional width)
  (when width
        (setf (slot-value 'kernel-width) width)
        (send self :adjust-screen))
  (slot-value 'kernel-width))

(defmeth p :adjust-screen ()
  (call-next-method)
  (let ((i (union
            (send self :points-selected)
            (send self :points-hilited))))
    (send self :clear-lines :draw nil)
    (if (< 1 (length i))
        (let ((x (select tensile-strength i))
              (y (select abrasion-loss i))
              (w (send self :kernel-width)))
          (send self :add-lines
                (kernel-smooth x y :width w)))
        (send self :redraw-content))))

(interval-slider-dialog '(20 100)
  :action #'(lambda (w) (send p :kernel-width w)))
```

A similar idea is useful for multivariate response data (e.g repeated measures or time series).

Suppose `p` is a plot of covariates and `q` is a plot of m responses, contained in the list `resp`, against their indices.

Then the method

```
(defmeth p :adjust-screen ()
  (call-next-method)
  (let ((i (union
            (send self :points-selected)
            (send self :points-hilited))))
      (send q :clear-lines :draw nil)
      (if i
          (flet ((ms (x) (mean (select x i))))
            (let ((y (mapcar #'ms resp))
                  (j (iseq 1 (length resp))))
              (send q :add-lines j y)))
          (send q :redraw-content))))
```

shows in `q` the average response profile for the highlighted and selected observations.

Another example was recently posted to `statlib` by Neely Atkinson, M. D. Anderson Cancer Center, Houston.

Here is a simplified version:

Suppose we have some censored survival data and a number of covariates.

To examine the relationship between the covariates and the times

- put the covariates in a plot, for example a scatterplot matrix
- in a second plot show the Kaplan-Meier estimate of the distribution of the survival times for points selected in the covariate plot.

To speed up performance, first define a faster Kaplan-Meier estimator that assumes the data are properly ordered:

```
(defun km (x s)
  (let ((n (length x)))
    (accumulate #'* (- 1 (/ s (iseq n 1))))))
```

Next, set up the covariate plot and a plot of the Kaplan-Meier estimator for all survival times:

```
(setf p (scatterplot-matrix (list x y z)))
(setf q (plot-lines
         (make-steps times (km times status))))
```

Finally, write an `:adjust-screen` method for `p` that changes the Kaplan-Meier estimator in `q` to use only the currently selected and highlighted points in `p`:

```
(defmeth p :adjust-screen ()
  (call-next-method)
  (let ((s (union
            (send self :points-selected)
            (send self :points-hilited))))
    (send q :clear-lines :draw nil)
    (if s
        (let* ((s (sort-data s))
               (tm (select times s))
               (st (select status s)))
          (send q :add-lines
                (make-steps tm (km tm st))))
        (send q :redraw-content))))
```

Using New Mouse Modes

We have already seen a few examples of useful new mouse modes.

As another example, let's make a plot for illustrating the sensitivity of least squares to points with high leverage.

Start with a plot of some simulated data:

```
(setf x (append (iseq 1 18) (list 30 40)))
(setf y (+ x (* 2 (normal-rand 20))))
(setf p (plot-points x y))
```

Next, add a new mouse mode:

```
(send p :add-mouse-mode 'point-moving
      :title "Point Moving"
      :cursor 'finger
      :click :do-point-moving)
```

The point moving method can be broken down into several pieces:

```
(defmeth p :do-point-moving (x y a b)
  (let ((p (send self :drag-point
                 x y :draw nil)))
    (if p (send self :set-regression-line))))

(defmeth p :set-regression-line ()
  (let ((coefs (send self :calculate-coefs)))
    (send self :clear-lines :draw nil)
    (send self :abline
          (select coefs 0)
          (select coefs 1))))

(defmeth p :calculate-coefs ()
  (let* ((i (iseq (send self :num-points)))
         (x (send self :point-coordinate 0 i))
         (y (send self :point-coordinate 1 i))
         (m (regression-model x y :print nil)))
    (send m :coef-estimates)))
```

If we want to change the fitting method, we only need to change the `:calculate-coefs` method.

Finally, add a regression line and put the plot into point moving mode

```
(send p :set-regression-line)

(send p :mouse-mode 'point-moving)
```

Several enhancements and variations are possible:

- The plot can be modified to use only visible points for fitting the line.
- Several fitting methods can be provided, with a dialog to choose among them.
- The same idea can be used to study the effect of outliers on smoothers.

As another example, we can set up a plot for obtaining graphical function input.

Start by setting up the plot, turning off the y axis, and adding a new mouse mode:

```
(setf p (plot-lines (rseq 0 1 50)
                    (repeat 0 50)))
(send p :y-axis nil)

(send p :add-mouse-mode 'drawing
      :title "Drawing"
      :cursor 'finger
      :click :mouse-drawing)
```

We can put the plot in our new mode with

```
(send p :mouse-mode 'drawing)
```

The drawing method is defined as

```
(defmeth p :mouse-drawing (x y m1 m2)
  (let* ((n (send self :num-lines))
         (rxy (send self :canvas-to-real x y))
         (rx (first rxy))
         (ry (second rxy))
         (old-i (max 0 (min (- n 1) (floor (* n rx)))))
         (old-y ry))
    (flet ((adjust (x y)
             (let* ((rxy (send self :canvas-to-real x y))
                    (rx (first rxy))
                    (ry (second rxy))
                    (new-i (max 0
                                (min (- n 1)
                                     (floor (* n rx)))))
                    (y ry))
               (dolist (i (iseq old-i new-i))
                 (let ((p (if (= old-i new-i)
                              1
                              (abs
                               (/ (- i old-i)
                                  (- new-i old-i))))))
                   (send self :linestart-coordinate 1 i
                         (+ (* p y) (* (- 1 p) old-y)))))
               (send self :redraw-content)
               (setf old-i new-i old-y y))))
      (adjust x y)
      (send self :while-button-down #'adjust))))
```

Some notes:

- `:canvas-to-real` converts the click coordinates to data coordinates.
- The `dolist` loop is needed to smooth out the curve if the mouse is moved rapidly.
- If the plot is to be normalized as a density, it can be adjusted before the `:redraw-content` message is sent.

A method for retrieving the lines is given by

```
(defmeth p :lines ()
  (let ((i (iseq 50)))
    (list
     (send self :linestart-coordinate 0 i)
     (send self :linestart-coordinate 1 i))))
```

Some Issues in Rotation

Statistical Issues

Rotation is a method for viewing a three dimensional set of data.

The objective of rotation is to create an illusion of 3D structure on a 2D screen.

Factors that can be used to produce or enhance such an illusion are

- stereo imaging
- perspective
- lighting
- motion parallax

Motion parallax is the principle driving rotation.

Scaling can have a significant effect on the shape of a point cloud.

Possible scaling strategies include

- fixed scaling – comparable scales
- variable scaling – non-comparable scales
- combinations

Several methods are available for enhancing the 3D illusion produced by rotation

- depth cuing
- framed box
- Rocking
- slicing

The nature of the rotation controls may also affect the illusion.

Some possible control strategies are

- rotation around screen axes: horizontal (Pitch), vertical (Yaw), out-of-screen (Roll)
- rotation around data axes: X, Y, Z
- direct manipulation (like a globe)

Rotation is still a fairly new technique; we are still learning how to use it effectively.

Some questions to keep in mind as you use rotation to explore your data sets:

- What types of phenomena are easy to detect?
- What is hard to detect?
- What enhancements aid in detection?

It may help to try rotation on some artificial structures and see if you can identify these structures.

Implementation Issues

The default scaling used by the rotating plot is `variable`.

You can change the scale type with the **Options** dialog or the `:scale-type` message.

You can implement non-standard scaling strategies by overriding the default `:adjust-to-data` method.

The standard rotating plot has controls for screen axis rotation.

You can use the `:transformation` and `:apply-transformation` methods to implement alternate control methods.

Plot overlays can be used to hold controls for these strategies.

Some Examples

By rocking the plot back and forth we can get the 3D illusion while keeping the view close to fixed.

A method for rocking the plot can be defines as

```
(defmeth spin-proto :rock-plot
         (&optional (a 0.15))
  (let* ((angle (send self :angle))
         (k (round (/ a angle))))
    (dotimes (i k)
      (send self :rotate-2 0 2 angle))
    (dotimes (i (* 2 k))
      (send self :rotate-2 0 2 (- angle)))
    (dotimes (i k)
      (send self :rotate-2 0 2 angle))))
```

A method for rotating around a specified data axis is

```
(defmeth spin-proto :data-rotate
         (axis &optional (angle pi))
  (let* ((alpha (send self :angle))
         (cols (column-list
                (send self :transformation)))
         (m (case axis
              (x (make-rotation (select cols 1)
                                (select cols 2)
                                alpha))
              (y (make-rotation (select cols 0)
                                (select cols 2)
                                alpha))
              (z (make-rotation (select cols 0)
                                (select cols 1)
                                alpha)))))
    (dotimes (i (floor (/ angle alpha)))
      (send self :apply-transformation m))))
```

To allow rotation to be done by direct manipulation, we can add a new mouse mode

```
(send spin-proto :add-mouse-mode 'hand-rotate
      :title "Hand Rotate"
      :cursor 'hand
      :click :do-hand-rotate)
```

and define the `:do-hand-rotate` method as

```
(defmeth spin-proto :do-hand-rotate (x y m1 m2)
  (flet ((calcsphere (x y)
           (let* ((norm-2 (+ (* x x) (* y y)))
                  (rad-2 (^ 1.7 2))
                  (z (if (< norm-2 rad-2) (sqrt (- rad-2 norm-2)) 0)))
             (if (< norm-2 rad-2)
                 (list x y z)
                 (let ((r (sqrt (max norm-2 rad-2))))
                   (list (/ x r) (/ y r) (/ z r))))))))
    (let* ((oldp (apply #'calcsphere
                        (send self :canvas-to-scaled x y)))
           (p oldp)
           (vars (send self :content-variables))
           (trans (identity-matrix (send self :num-variables))))
      (send self :idle-on nil)
      (send self :while-button-down
            #'(lambda (x y)
                (setf oldp p)
                (setf p (apply #'calcsphere
                               (send self :canvas-to-scaled x y)))
                (setf (select trans vars vars) (make-rotation oldp p))
                (when m1
                      (send self :slot-value 'rotation-type trans)
                      (send self :idle-on t))
                (send self :apply-transformation trans))))))
```

4D and Beyond

Introduction and Background

As with two and three dimensional plots, the objectives of higher dimensional graphics are, among others, to

- detect groupings or clusters
- detect lower dimensional structure
- detect other patterns

Higher dimensions introduce new difficulties:

- sparseness
- loss of intuition

A rough ranking of types of structures that can be detected might look like this:

- points (zero-dimensional structures) in 1D and 2D plots
- curves (one-dimensional structures) in 2D and 3D plots
- surfaces (two-dimensional structures) in 3D plots

Possible structures in four dimensions include

- separate point clusters
- curves
- 2D surfaces
- 3D surfaces

Useful general techniques include

- glyph augmentation
  - colors
  - symbol types
  - symbol sizes
- dimension reduction
  - projection
  - slicing, masking, conditioning

Some issues to keep in mind:

- using symbols or colors loses ordering
- projections to 2D will only become 1D if part of the structure is linear
- masking/conditioning requires large data sets

Some specific implementations:

- scatterplot matrix
- linked 2D plots
- 3D plot linked with a histogram
- plot interpolation
- grand tours and variants
- parallel coordinates

Plot Interpolation

Plot interpolation provides a way of examining the relation among 4 variables.

The idea is to take two scatterplots and rotate from one to the other.

(Interpolate using trigonometric interpolation.)

Watching the interpolation allows you to

- track groups of points from one plot to another
- identify clusters based on both location and velocity

A simple function to set up an interpolation plot for a data set with four variables:

```
(defun interp-plot (data &rest args)
  ;; Should check here that data is 4D
  (let ((w (apply #'plot-points data
                  :scale 'variable args))
        (m (matrix '(4 4) (repeat 0 16))))
    (flet ((interpolate (p)
             (let* ((a (* (/ pi 2) p))
                    (s (sin a))
                    (c (cos a)))
               (setf (select m 0 0) c)
               (setf (select m 0 2) s)
               (setf (select m 1 1) c)
               (setf (select m 1 3) s)
               (send w :transformation m))))
      (let ((s (interval-slider-dialog
                '(0 1)
                :points 25
                :action #'interpolate)))
        (send w :add-subordinate s)))
    w))
```

Some points:

- The definition allows keywords to be passed on to `plot-points`.
- The plot `w` and the matrix `m` are locked into the environment of `interpolate`.
- The slider is registered as a subordinate.

Some examples to try:

- Stack loss data
- Places rated data
- Iris data

The Grand Tour

The Grand Tour provides a way of examining "all" one- or two-dimensional projections of a higher-dimensional data set.

The projections are moved smoothly in a way that eventually brings them near every possible projection.

One way to construct a Grand Tour is to repeat the following steps:

- Choose two directions at random to define a rotation plane.
- Rotate in this plane by a specified angle a random number of times.

A simple Grand Tour function:

```
(defun tour-plot (&rest args)
  (let ((p (apply #'spin-plot args)))
    (send p :add-slot 'tour-count -1)
    (send p :add-slot 'tour-trans nil)
    (defmeth p :tour-step ()
      (when (< (slot-value 'tour-count) 0)
        (let ((vars (send self :num-variables))
              (angle (send self :angle)))
          (setf (slot-value 'tour-count)
                (random 20))
          (setf (slot-value 'tour-trans)
                (make-rotation
                 (sphere-rand vars)
                 (sphere-rand vars)
                 angle))))
      (send self :apply-transformation
            (slot-value 'tour-trans))
      (setf (slot-value 'tour-count)
            (- (slot-value 'tour-count) 1)))
    (defmeth p :do-idle ()
      (send self :tour-step))
    (send (send p :menu) :append-items
          (send run-item-proto :new p))
    p))
```

The `sphere-rand` function can be defined as

```
(defun sphere-rand (n)
  (let* ((z (normal-rand n))
         (r (sqrt (sum (^ z 2)))))
    (if (< 0 r)
        (/ z r)
        (repeat (/ (sqrt n)) n))))
```

Some examples to try the tour on:

- Diabetes data
- Iris data
- Stack loss data
- 4D structures

Some variations and additions:

- Make new rotation orthogonal to current viewing plane
- Controls for replaying parts of the tour
- Touring on only some of the variables (independent variables only)
- Constraints on the tour (Correlation Tour)
- Integration with slicing/conditioning
- Guided tours (Projection Pursuit – XGobi)

A similar idea can be used for examining functions.

As an example, suppose we want to determine if $f(x)$ looks like a multivariate standard normal density.

This is true if and only if $g(z) = f(zu)$ looks like a univariate standard normal density for any unit vector u.

By moving u through space with a series of random rotations, we can look at these univariate densities for a variety of different directions.

The normality checking plot can be implemented as a prototype:

```
(defproto ncheck-plot-proto
  '(function direction xvals
    tour-count tour-trans angle)
  ()
  scatterplot-proto)

(send ncheck-plot-proto :slot-value
      'tour-count -1)
(send ncheck-plot-proto :slot-value
      'angle 0.2)
```

The method for adjusting the image to the current state is

```
(defmeth ncheck-plot-proto :set-image ()
  (let* ((x (slot-value 'xvals))
         (f (slot-value 'function))
         (d (slot-value 'direction))
         (y (mapcar #'(lambda (x)
                        (funcall f (* x d)))
                    x)))
    (send self :clear-lines :draw nil)
    (send self :add-lines (spline x y))))
```

The method for moving the direction is similar to the tour method used earlier:

```
(defmeth ncheck-plot-proto :tour-step ()
  (when (< (slot-value 'tour-count) 0)
        (let* ((d (slot-value 'direction))
               (n (length d))
               (a (abs (slot-value 'angle))))
          (setf (slot-value 'tour-count)
                (random (floor (/ pi
                                  (* 2 a)))))
          (setf (slot-value 'tour-trans)
                (make-rotation d
                               (normal-rand n)
                               a))))
  (setf (slot-value 'direction)
        (matmult (slot-value 'tour-trans)
                 (slot-value 'direction)))
  (send self :set-image)
  (setf (slot-value 'tour-count)
        (- (slot-value 'tour-count) 1)))
```

Two additional methods:

```
(defmeth ncheck-plot-proto :do-idle ()
  (send self :tour-step))

(defmeth ncheck-plot-proto :menu-template ()
  (append
   (call-next-method)
   (list (send run-item-proto :new self))))
```

Finally, the initialization method sets up the plot:

```
(defmeth ncheck-plot-proto :isnew (f d)
  (setf (slot-value 'function) f)
  (setf (slot-value 'direction) d)
  (setf (slot-value 'xvals) (rseq -3 3 7))
  (call-next-method 2)
  (send self :range 0 -3 3)
  (send self :range 1 0 1.2)
  (send self :x-axis t nil 7)
  (send self :y-axis nil)
  (send self :set-image))
```

Some examples to try:

- independent gamma variates
- normal mixtures
- likelihood function
- posterior densities

Some possible enhancements:

- normal curve for comparison
- replay tools
- integration with transformations
- guided tour

Some References

Asimov, D., (1985), "The grand tour: a tool for viewing multidimensional data," *SIAM J. of Scient. and Statist. Comp.* 6, 128-143.

Becker, R. A. and Cleveland, W. S., (1987), "Brushing scatterplots," *Technometrics* 29, 127-142, reprinted in *Dynamic Graphics for Statistics*, W. S. Cleveland and M. E. McGill (eds.), Belmont, Ca.: Wadsworth.

Becker, R. A., Cleveland, W. S. and Weil, G., (1988), "The use of brushing and rotation for data analysis," in *Dynamic Graphics for Statistics*, W. S. Cleveland and M. E. McGill (eds.), Belmont, Ca.: Wadsworth.

Bolorforush, M., and Wegman, E. J. (1988), "On some graphical representations of multivariate data," *Computing Science and Statistics: Proceedings of the 20th Symposium on the Interface*, E. J. Wegman, D. T. Ganz, and J. J. Miller, editors, Alexandria, VA: ASA, 121-126.

Buja, A., Asimov, D., Hurley, C., and McDonald, J. A., (1988), "Elements of a viewing pipeline for data analysis," in *Dynamic Graphics for Statistics*, W. S. Cleveland and M. E. McGill (eds.), Belmont, Ca.: Wadsworth.

Donoho, A. W., Donoho, D. L. and Gasko, M., (1988), "MACSPIN: Dynamic Graphics on a Desktop Computer," n *Dynamic Graphics for Statistics*, W. S. Cleveland and M. E. McGill (eds.), Belmont, Ca.: Wadsworth.

Fisherkeller, M. A., Friedman, J. H. and Tukey, J. W., (1974), "PRIM-9: An interactive multidimensional data display and analysis system," in *Data: Its Use, Organization and Management*, 140-145, New York: ACM, reprinted in *Dynamic Graphics for Statistics*, W. S. Cleveland and M. E. McGill (eds.), Belmont, Ca.: Wadsworth.

Inselberg, A., and Dimsdale, B. (1988), "Visualizing multi-dimensional geometry with parallel coordinates," *Computing Science and Statistics: Proceedings of the 20th Symposium on the Interface*, E. J. Wegman, D. T. Ganz, and J. J. Miller, editors, Alexandria, VA: ASA, 115-120.

Comments

Some statistical issues:

- Many good ideas for viewing higher-dimensional data are available
- There is room for many more ideas
- Eventually, it will be useful to learn to quantify the effectiveness of different viewing methods in different situations.
- It is also important to explore numerical enhancements as well as combinations of different graphical strategies.

Some computational issues:

- For point clouds, image rendering is quite fast, even on stock hardware.
- This is not true of lines and surfaces.
- The effective use of high performance 3D hardware is worth exploring.
- For data sets derived from models, computing speed can still be limiting
- Pre-computation can help, but limits interaction.