GUI Model

Matthew Bartenshlag, Peter Santiago

April 16, 2009

1 Defmodel Code

Matt will provide an explaination here at some point once the code is finalized. See Figures 1, 2, 3, and 4 for code.

```
;; Inputs is of the form
;; k => name provided by user for input
;; v => the code generated to make an instance
;; ex: (:scale pony 0 10 1) creates an entry
       ;; where pony=>(make-instance 'scale ...)
(defparameter *inputs* (make-hash-table))
;; The parameters passed to the action function.
(defparameter *params* nil)
;; Resets the model
(defun reset-inputs ()
  (setf *inputs* (make-hash-table))
  (setf *params* '()))
;; Converts a keyworded form to the
;;appropriate LTK make-instance.
;; Side-effect: Adds the instance to *inputs*
(defun convert-to-instance (keyword)
  (let ((key  (first  keyword)) ; eg: :scale
        (name (second keyword)) ;
        (vals (cddr   keyword)) ; all additional parameters
        (special (eql 'special (first (last keyword))))
        tmp)
    (cond
      ((equal key :scale)
       (progn
         (setf tmp
               `(make-instance 'scale
                               :master main-label
                               :label ,(format nil "~A" name)
                               :from ,(first vals)
                               :to ,(second vals)
                               :digits 5
                               :resolution ,(or (third vals) 1
                               :troughcolor ,(if special :red  black)
                               :orientation 'horizontal))
         (setf (gethash name *inputs*) tmp))))
    tmp))
(defmacro defmodel (name inputs model importance &body action)
  (reset-inputs)
  (let (
        letz   ; all LTK forms
        packer
        ;;generated from letz, a sequence of (pack NAME)
        ui-name
        model-name
        function-name
        function-parameters
        importance-name
        )
    ; Generates two symbols: NAME-MODEL and NAME-PROCESSOR
    (setf model-name (intern (format nil "~A-MODEL"
                                         name)))
    (setf function-name (intern (format nil "~A-PROCESSOR"
                                              name)))
    (setf ui-name (intern (format nil "~A-UI" name)))
    (setf importance-name (intern (format nil "~A-IMPORTANCE"
                                                  name)))
```

Figure 1: CODE: defmodelcode1

```
    ; Two glorious effects:
    ;  a. Generate a parameter listing for the processor
       ;;function from the inputs.
    ;  b. Convert the :scale syntax to LTK syntax
    (dolist (i inputs)
      (setf function-parameters
            (append function-parameters (list (second i))))
      (setf *params* (append *params*
                             (list `(value ,(second i)))))
      (convert-to-instance i))
    (setf letz
          (append letz
                  (list `(display-label
                          (make-instance 'labelframe
                                         :text "Display")))))
(setf letz (append letz
                   (list `(main-label
                           (make-instance 'labelframe
                                          :text "Inputs")))))

(setf letz (append letz
                   (list `(action-label
                           (make-instance 'labelframe
                                          :text "Action")))))

(setf letz (append letz
                   (list `(kanvas (make-canvas display-label
                                               :width 384 :height 865)))))

    (maphash #'(lambda (k v)
                 (setf letz (append letz (list (list k v)))))
             *inputs*)
(setf letz (append letz (list `(img0 (make-image)))))
(setf letz (append letz (list `(img1 (make-image)))))
(setf letz (append letz (list `(img2 (make-image)))))
```

Figure 2: CODE: defmodelcode2

```
(setf letz
      (append letz
              (list
               `(action2
                 (make-instance 'button
                                :master action-label
                                :text "Generate 3D"
                                :command #'(lambda ()
(progn
  (funcall #',function-name ,@*params*)
  (sleep 1)
  (image-load img0 "0.gif")
  (image-load img1 "1.gif")
  (image-load img2 "2.gif")
  (create-image kanvas 0 0 :image img0)
  (create-image kanvas 0 289 :image img1)
  (create-image kanvas 0 578 :image img2))))))))
(setf letz
      (append letz
              (list
               `(action3
                 (make-instance 'button
                                :master action-label
                                :text "Importance"
                                :command #'(lambda ()

(progn
  (funcall #',importance-name ,@function-parameters))))))))
(setf packer
      (mapcar
       #'(lambda (l)
           (cond
             ((or (eql 'display-label l)
                  (eql 'kanvas l)) `(pack ,l :side :right))
             ((or (eql 'img0 l)
                  (eql 'img1 l) (eql 'img2 l)) nil)
             (t `(pack ,l)))) (mapcar #'first letz)))
```

Figure 3: CODE: defmodelcode3

```
 (eval `(defun ,ui-name ()
        (with-ltk ()
          (let* ,letz
            ,@packer))))
    (eval `(defun ,model-name ,function-parameters
            ,@model))
    (eval `(defun ,importance-name ,function-parameters
            ,@importance))
    `(defun ,function-name ,function-parameters
        (let* ((path (make-pathname :name "tmp.dat"))
               (output-stream (open path :direction :output
                                    :if-exists :supersede)))
          (with-gnuplot ('linux)
            (set-grid 'on)
            (format-gnuplot "set xticks 1")
            (format-gnuplot "set yticks 1")
            ,@action)))))

(defparameter *p-values*
  #(.25 .22 .14 .14 .069 .064 .033 .033 .022 .02 .013))
(defparameter *s-values* #(.728 .166 .065 .018 .018 .006))
(defparameter *DL* #\Tab)


;(defun update-scale-color ()
;  (configure 'lambda troughcolor :blue))

(defmacro set-labels (x y z)
  `(progn
     (format-gnuplot "set xlabel '~A'" ,x)
     (format-gnuplot "set ylabel '~A'" ,y)
     (format-gnuplot "set zlabel '~A'" ,z)))

(defmacro generate-thumbs (viewx viewy dx dy)
  `(progn
     (format-gnuplot "set terminal gif")
     (format-gnuplot "set size .6,.6")
     (dotimes (i 3)
       (format-gnuplot
        (format nil "set view ~A,~A"
                (+ ,viewx (* (1+ i) ,dx))
                (+ ,viewy (* (1+ i) ,dy))))
       (format-gnuplot
        (format nil "set output \"~A.gif\"" i))
       (format-gnuplot "splot 'tmp.dat' with linespoints"))
       (format-gnuplot "set terminal wxt")
       (format-gnuplot "set size 1,1")
       (format-gnuplot "set output")))

(defun calculate-t (l j)
  (/ (* (exp (- l)) (expt l j)) (n! j)))

(defun n! (n)
  (if (or (= n 0) (= n 1)) 1 (* n (n! (- n 1)))))
```

Figure 4: CODE: defmodelcode4

1.1 Using Real-World Data

Real-world data can be used in our model to inform the user (or the underlying model programming) of slider choices which are represented in our knowledge base. Slider choices which are more likely than others to occur in the real world have a changed slider color when such a choice has been made. Knowledge of real-world data is maintained through simple frequency counts – this allows for very simply and quick calculations to be performed.

1.2 Using Model-Generated Data

Model-generated data are created and interpreted when the user presses the 'Important' button. The data are interpreted using a support-based Bayesian Ranking.

1. Several thousand samples of data to be passed to the model are generated.

2. These samples are run through the model, and their results are generated.

2. The ranges for attributes of the samples are binned using an equal frequency binning algorithm.

3. These samples are then classified into the *best* (currently 10% highest) and *rest*.

4. Each range for each attribute is then ranked according to the following algorithm:

$$P(best|E) * support(best|E) = \frac{like(best|E)^2}{like(best|E) + like(rest|E)} \tag{1}$$

where

$$like(best|E) = freq(E|best) * P(best) \tag{2}$$

$$like(rest|E) = freq(E|rest) * P(rest) \tag{3}$$

```
(defmodel evett
    ((:scale lamb 1 10)
     (:scale fvalue .01 .25 .01 special)
     (:scale n 1 5)
     (:scale lr 25 225 25))

    ;This becomes the "model" accessible by evett-model.
    ((min lr (+ (/ (* (elt *p-values* 0)
                      (calculate-t lamb (+ 1 n)))
                   (* (elt *p-values* 1)
                      (elt *s-values*  n) fvalue))
                (calculate-t  lamb 0))))

    ;Importance isn't implemented for this model. Yet?
    ((configure lamb :troughcolor
                (intern (format nil "gray~A" 30)))
     (configure n :troughcolor :green))

  ;This is the business that generates the 3D image. It can't be removed.
  (format output-stream "#LAMBDA ~A N ~A LR~%" *DL* *DL*)
  (dotimes (x (round lamb))
    (dotimes (y (round n))
      (format output-stream "~A ~A ~A ~A ~A~%"
              (+ 1 x) *DL* (1+ y) *DL*
              (evett-model (1+ x) fvalue (1+ y) lr))))

  (close output-stream)
  (format-gnuplot "set dgrid3d")
  (set-labels "LAMBDA" "N" "LR")
  (generate-thumbs 30 40 40 30)
  (format-gnuplot "splot 'tmp.dat' with linespoints"))
```

Figure 5: CODE: `defmodelevett`

2 Evett Model

I'll have to get matt to add a description here someday. See Figure 5 for code. A sample gui image is located in Figure 6.

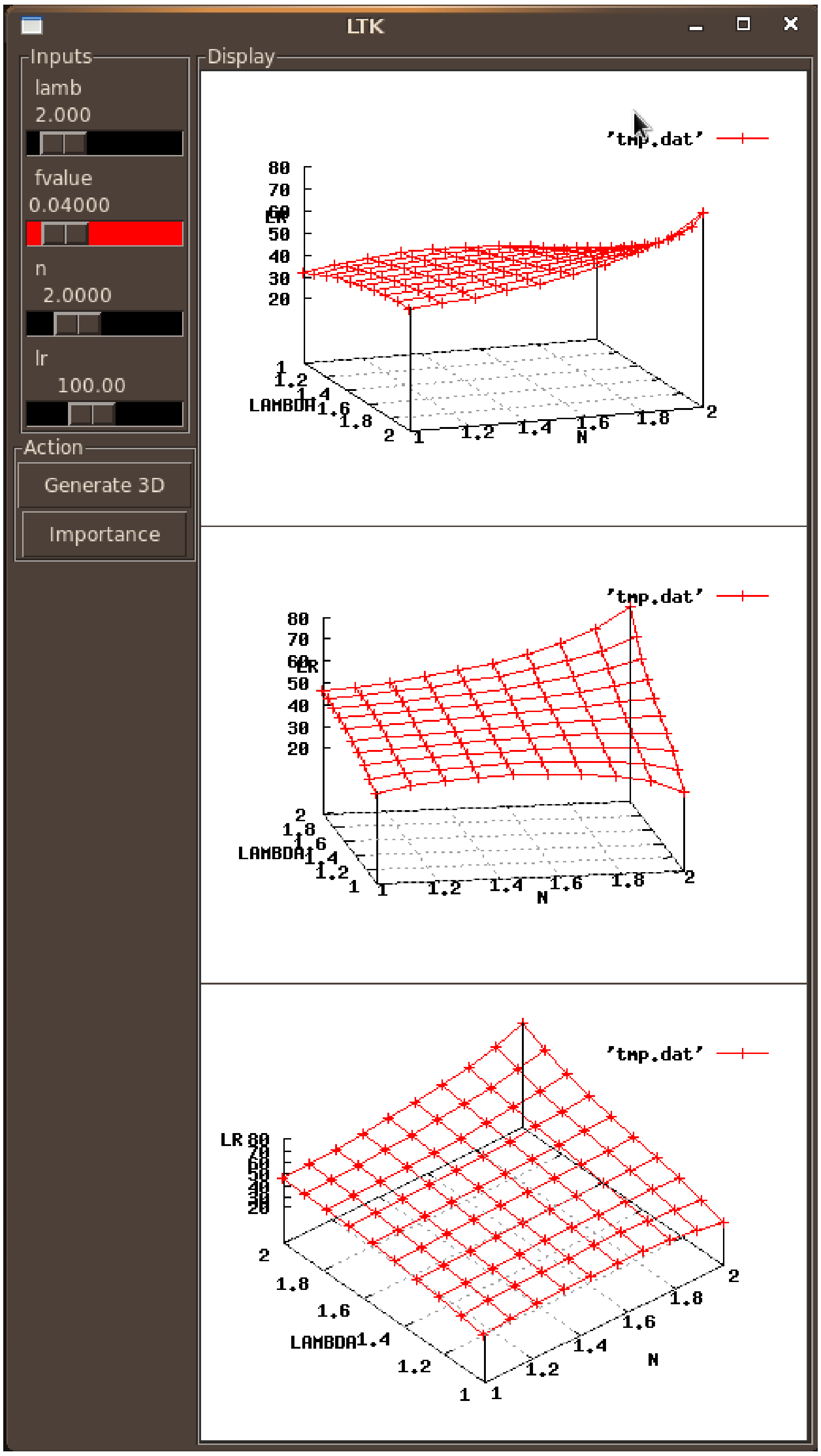


Figure 6: Evett GUI

3 Seheult Model

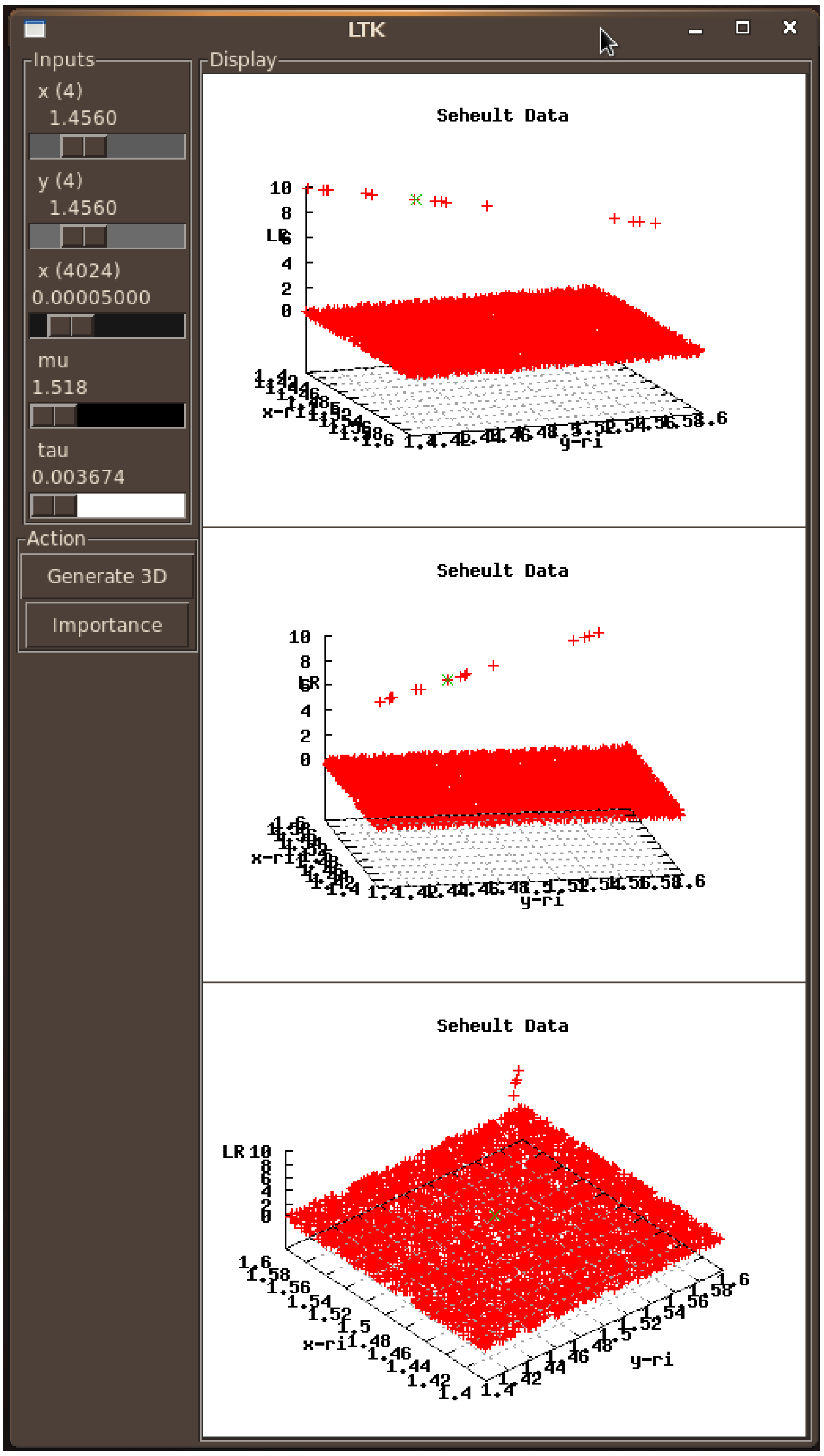
The final section of the defmodel pertaining to the generation of three-dimensional graphs has been heavily modified. This model implements the Seheult algorithm as described in (insert section reference once this is combined with the full document). See Figures 7, 8 for code. A sample gui image is located in Figure 9.

```
(defmodel sh
             ((:scale x 1.4 1.6 .001)
              (:scale y 1.4 1.6 .001)
              (:scale sd .00001 .00025 .00001)
              (:scale mu 1.5181932 15.181932 1.5181932)
              (:scale tau 0.0036737733 0.036737733 0.0036737733))
    ((format t "Wut~%")
     (let* ((l (/ sd tau))
         (ll (* l l))
         (u (/ (- x y) (* sd (sqrt 2))))
         (z (* .5 (+ x y)))
         (v (/ (- z mu) (* tau (expt (+ 1 (* .5 ll)) .5))))
         (factor (* (/ (+ 1 ll) (* l (expt (+ 2 ll) .5)))
                    (exp (* (/ -1 (* 2 (+ 1 ll)))
                            (- (* u u) (* v v)))))))
    factor))
    ((format t "Start!~%")
     (let ((s (demo-seheult)) str (xf 0))
       (labels ((between (left right val) (and (<= left val) (<= val right))))

         (dotimes (i (length *x-freqs*) xf)
           (when (<= (value x) (car (nth i *x-freqs*)))
               (unless (< (1- i) 0)
                 (configure x :label (format nil "x (~A)" (second (nth (1- i) *
                 (return))))
         (format t "Frequency of x=~A is ~A~%" (value x) xf)
         (dolist (NANER (gethash 'x s))
           (when (between (first NANER) (second NANER) (value x))
             (configure x :troughcolor (intern (format nil "gray~A" (round (* 1
             (format t "Value: ~A~%" (round (* 100 (first (last NANER)))))))
         (dotimes (i (length *y-freqs*) xf)
           (when (<= (value y) (car (nth i *y-freqs*)))
               (unless (< (1- i) 0)
                 (configure y :label (format nil "y (~A)" (second (nth (1- i) *
                 (return))))
         (dolist (NANER (gethash 'y s))
           (if (between (first NANER) (second NANER) (value y))
               (configure y :troughcolor (intern (format nil "gray~A" (round (*
         (dotimes (i (length *sd-freqs*) xf)
           (when (<= (value sd) (car (nth i *sd-freqs*)))
             (unless (< (1- i) 0)
               (configure sd :label (format nil "x (~A)" (second (nth (1- i) *s
               (return))))
         (dolist (NANER (gethash 'sd s))
           (if (between (first NANER) (second NANER) (value sd))
               (configure sd :troughcolor (intern (format nil "gray~A" (round (
         (dolist (NANER (gethash 'mu s))
           (if (between (first NANER) (second NANER) (value mu))
               (configure mu :troughcolor (intern (format nil "gray~A" (round (
         (dolist (NANER (gethash 'tau s))
           (if
            (between (first NANER) (second NANER) (value tau))
            (configure
             tau :troughcolor
             (intern
              (format nil "gray~A"
               (round (* 100 (first (last NANER))))))))))
         (format t "END~%"))))
(format t "Woot~%")
  (new-random-seheult 1.4 1.6 1 1)
  (let (tempval)
    (handler-case
        (progn
          (setf tempval (sh-model x y sd mu tau)))

      (floating-point-overflow ()
        (setf tempval 10)))
    (if (> tempval 10)
        (setf tempval 10))
    (with-open-file (out "randomdata.dat"
                         :direction :output
                         :if-exists :append)
      (format out "~%~%~a ~a ~a" x y tempval))
    tempval)
  (format-gnuplot "set title 'Seheult Data'")
  (format-gnuplot "unset key")
  (set-labels "x-ri" "y-ri" "LR")
  (new-generate-thumbs 30 40 40 30)
  (format-gnuplot "splot 'randomdata.dat' index 0,
'randomdata.dat' index 1"))
```

Figure 8: CODE: `defmodelseheult2`



x (4)
1.4560

y (4)
1.4560

x (4024)
0.00005000

mu
1.518

tau
0.003674

Figure 9: Seheult GUI

4 Grove Model

The final section of the defmodel pertaining to the generation of three-dimensional graphs has been heavily modified. This model implements the Grove algorithm as described in (insert section reference once this is combined with the full document). See Figures 10, 11 for code. A sample gui image is located in Figure 12.

```
(defmodel grove
             ((:scale x 1.4 1.6 .001)
              (:scale y 1.4 1.6 .001)
              (:scale sd .00001 .00025 .00001)
              (:scale mu 1.5181932 15.181932 1.5181932)
              (:scale tau 0.0036737733 0.036737733 0.0036737733))
    ((format t "Wut~%")
     (let* ((u (/ tau sd))
         (v (/ (* (- x y)(- x y) ) (* -4 sd sd) ))
         (w (/ (* (- y mu)(- y mu) ) (* 2 tau tau)))
         (lr (* u  (exp ( + v w )))))
         lr))
    ((format t "Start!~%")
     (let ((s (demo-grove)) str (xf 0))
       (labels ((between (left right val) (and (<= left val) (<= val right))))
         (dotimes (i (length *x-freqs*) xf)
           (when (<= (value x) (car (nth i *x-freqs*)))
               (unless (< (1- i) 0)
                 (configure x :label (format nil "x (~A)" (second (nth (1- i) *
                 (return))))
         (format t "Frequency of x=~A is ~A~%" (value x) xf)
         (dolist (NANER (gethash 'x s))
           (when (between (first NANER) (second NANER) (value x))
             (configure x :troughcolor (intern (format nil "gray~A" (round (* 1
             (format t "Value: ~A~%" (round (* 100 (first (last NANER)))))))
         (dotimes (i (length *y-freqs*) xf)
           (when (<= (value y) (car (nth i *y-freqs*)))
               (unless (< (1- i) 0)
                 (configure y :label (format nil "y (~A)" (second (nth (1- i) *
                 (return))))
         (dolist (NANER (gethash 'y s))
           (if (between (first NANER) (second NANER) (value y))
               (configure y :troughcolor (intern (format nil "gray~A" (round (*
         (dotimes (i (length *sd-freqs*) xf)
           (when (<= (value sd) (car (nth i *sd-freqs*)))
             (unless (< (1- i) 0)
               (configure sd :label (format nil "x (~A)" (second (nth (1- i) *s
               (return))))
         (dolist (NANER (gethash 'sd s))
           (if (between (first NANER) (second NANER) (value sd))
               (configure sd :troughcolor (intern (format nil "gray~A" (round (
         (dolist (NANER (gethash 'mu s))
           (if (between (first NANER) (second NANER) (value mu))
               (configure mu :troughcolor (intern (format nil "gray~A" (round (
         (dolist (NANER (gethash 'tau s))
           (if (between (first NANER) (second NANER) (value tau))
               (configure tau :troughcolor (intern (format nil "gray~A" (round
         (format t "END~%"))))
```

Figure 10: CODE: `defmodelgrove1`

```
(format t "Woot~%")
  (new-random-grove 1.4 1.6 1 1)
  (let (tempval)
    (handler-case
        (progn
          (setf tempval (grove-model x y sd mu tau)))

      (floating-point-overflow ()
        (setf tempval 10)))
    (if (> tempval 10)
        (setf tempval 10))
    (with-open-file (out "randomdata.dat"
                         :direction :output
                         :if-exists :append)
      (format out "~%~%~a ~a ~a" x y tempval))
    tempval)
  (format-gnuplot "unset key")
  (format-gnuplot "set title 'Grove Data'")
  (set-labels "x-ri" "y-ri" "LR")
  (new-generate-thumbs 30 40 40 30)
  (format-gnuplot "splot 'randomdata.dat' index 0,
'randomdata.dat' index 1"))
```

Figure 11: CODE: `defmodelgrove2`

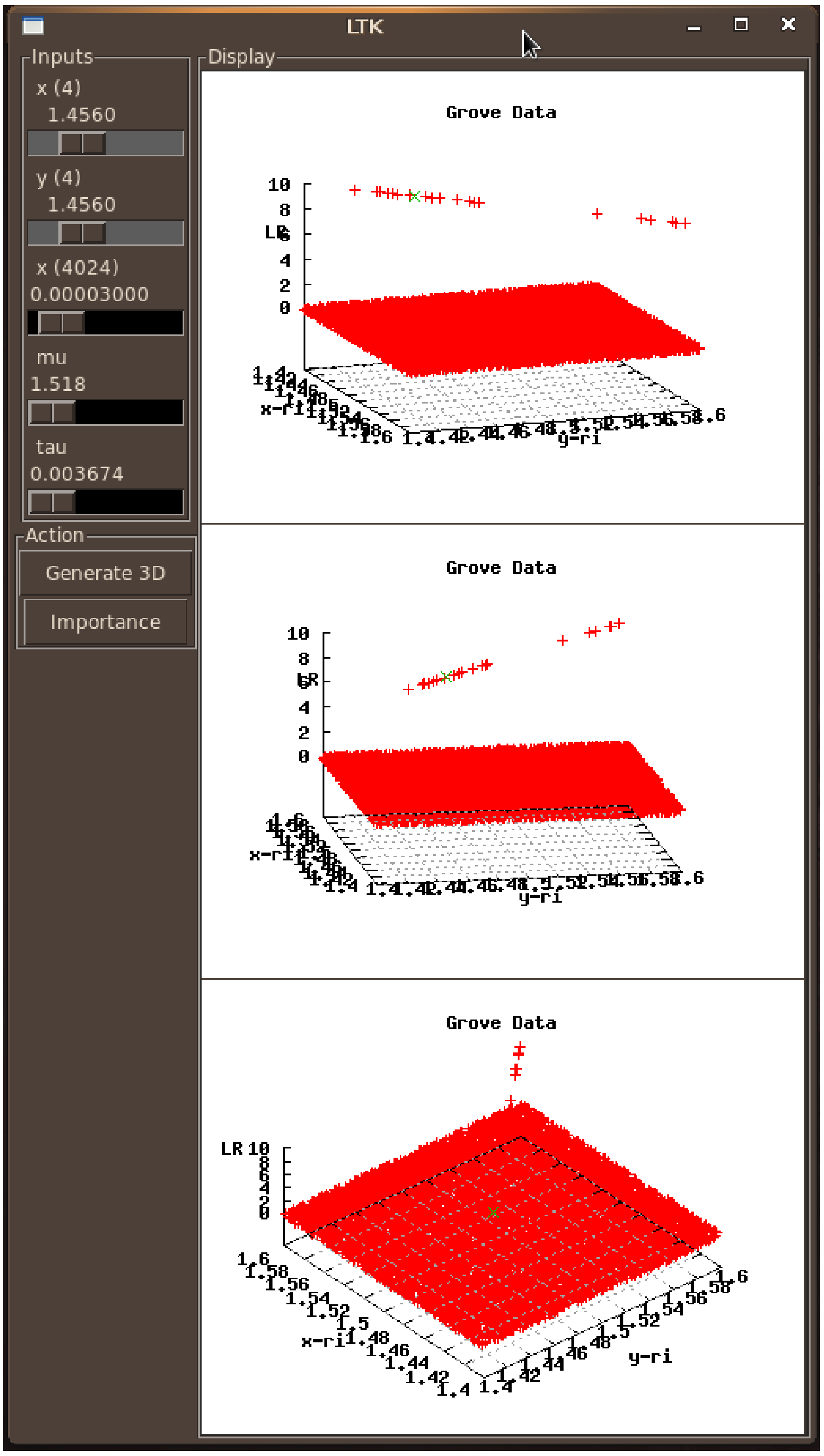


Figure 12: Grove GUI