A Literate Program for a Hyper-Document

B. Narasimhan
Department of Mathematics
Penn State Erie, The Behrend College
Erie, PA 16563

Draft of December 20, 1994

Abstract

We describe a literate program for a hyper-document using the `noweb` literate programming tools. The program essentially consists of a hyper-document and the associated code that implements the hyper-links. The code uses the `Lisp-Stat` environment. The hyper-document can be LaTeXed and viewed with `XHDvi`, a hyper-TeX dvi previewer. This is but a small part of a free hyper-text for introductory statistics that is being developed by a group of us.

1 Introduction

This paper describes a literate program for a hyper-document. The hyper-document is a LaTeX file with embedded hyper-links to `Lisp-Stat` code. Using a hyper-TeX viewer like `XHDvi` one can view the document and have `Lisp-Stat` automatically invoked on the hyper-links.

Before we go any further, it must be remarked that all the tools mentioned earlier are freely available. Here is a list that will get you started.

The `noweb` tools are available from all `CTAN` sites in `/web/noweb`. As a last recourse, it is also available at `ftp://bellcore.com:/pub/norman`. `Lisp-Stat` is available from `ftp://stat.umn.edu/pub/xlispstat`. `XHDvi` is available from `http://xxx.lanl.gov/hypertex/`. Several pre-compiled binaries are also available. `XHDvi` is still under beta-test, and actually contains bugs. For example, it is able to process only one hyper-link per invocation, and dumps core on the second, at least on my SGI machine. However, such problems are bound to be fixed in the near future.

After one has LaTeXed the document, one needs to invoke `XHDvi` as shown below.

```
% xhdvi -browser 'xterm -e xlispstat' hyperdoc &
```

One can also, of course, set the `.mailcap` and `.mime.types` entries, but they are not really necessary for this simple hyper-document.

Some users might be interested in using just the statistical tables. The figure 1 shows how the code works. To invoke the code, one does the following.

```
% xlispstat stbls
```

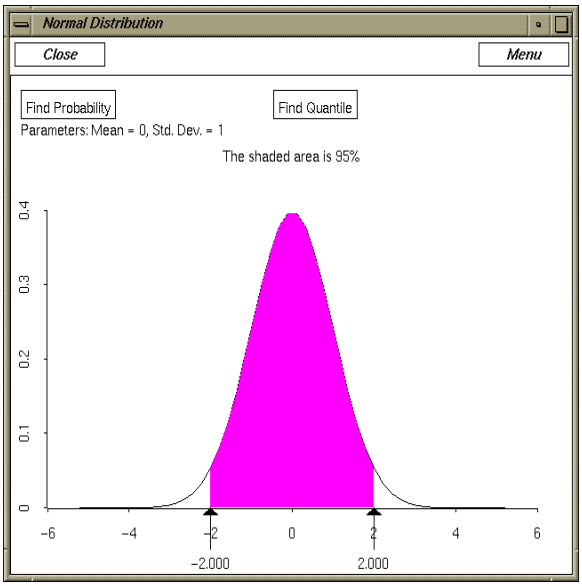

Figure 1: Statistical Tables in `Lisp-Stat`

Here is our entire document.

2a ⟨*Literate Program* 2a⟩≡
⟨*Hyper-document stuff* 2b⟩
⟨*Makefile* 51a⟩
⟨*Readme file* 51b⟩
Root chunk (not used in this document).

Let us begin with `Hyper-document stuff`. There are two main parts. The first is `Hyper-document` which contains textual matter with embedded hyper-links. The second is the `Code` that implements the actions invoked when a hyper-link is used.

2b ⟨*Hyper-document stuff* 2b⟩≡
⟨*Hyper-document* 3⟩
⟨*Code* 6⟩
This code is used in chunk 2a.

2 The Hyper-document

So, what is our hyper-document about? The document should essentially teach students how to use statistical tables for various distributions. As this literate program is already sizeable in terms of printed pages, we shall keep the document brief and save a few trees.

This hyper-document assumes students know about histograms. The material in this document is a compressed excerpt from my notes from Moore and McCabe[1] for my introductory statistics class.

3 ⟨*Hyper-document* 3⟩≡

```
\documentstyle{article}
\def\href#1#2{\special{html:<a href="#1">}{#2}\special{html:</a>}}
\begin{document}
\section{Density Functions}
Consider a data set of $1000$ numbers. We already know to pictorially
summarize the distribution of these numbers by means of histograms.
Here, for \href{file:hist.lsp}{example}, is a histogram of $1000$
numbers. We can \href{file:smhist.lsp}{approximate} the histogram by a
smooth curve that displays the shape of the distribution after ironing
out some of the raggedness. Such smoothing loses some details in the
histogram and therefore can be thought of as an idealized form of the
histogram.  We can also say that the unevenness in the histogram is a
consequence of the classes we have chosen and so the curve is really a
better description of the data.  In a relative frequency histogram,
the areas of the bars are proportional to the relative frequency of
the classes. And if we add together all the relative frequencies, we
get $1$. Therefore, it is natural to ask that our idealization of the
histogram, the smooth curve have total area $1$ underneath it. The area
under the curve between any two values on the $x$-axis is then equal
to the proportion of observations falling between the two values. This
curve is called the density curve of the distribution of the data.

There are many density curves. Let us study an important one which is
called the normal density.

\subsection{The Normal Density}
Let us first take a look at the normal density. Please click
\href{file:stbl.lsp}{here} and continue reading this document for
further instructions.

Now, if all went well, you should have a menu with the word {\bf
  Tables} on it. Press your mouse on the word {\bf Tables} and drag it
on to the {\bf Normal Distribution} menu item and release the mouse
button.

Do you see the density? It should have a bell-shape with a single
```

```
peak. Notice how the density is {\em symmetric\/} about $0$; that is,
the shape to the left of $0$ is the same as the shape to the right of
$0$. The exact density of a normal curve is described by giving
information about two quantities, the mean $\mu$, where the peak
occurs and the standard deviation $\sigma$, which specifies how widely
spread the curve is. The curve that you see now has $\mu=0$ and
$\sigma=1$ and is called the standard normal distribution. The exact
formula for a normal density curve with mean $\mu$ and standard
deviation $\sigma$ is given by
\begin{equation}
\phi(x) = \frac{1}{\sigma\sqrt{2\pi}}
\exp{-\frac{1}{2}\biggl(\frac{x-\mu}{\sigma}\biggr)^2}
\label{eq:normal-density}
\end{equation}

The normal density curve has the following property, which is often
referred to as the empirical rule.

\begin{center}
\fbox{
  \parbox[b]{4in}{
    \paragraph{The $68$-$95$-$99$ Rule}
    \begin{itemize}
    \item \href{file:68.lsp}{$68$\%} of the observations fall within
      $\sigma$ of the mean $\mu$.
    \item \href{file:95.lsp}{$95$\%} of the observations fall within
      $2\sigma$ of the mean $\mu$.
    \item \href{file:99.lsp}{$99.7$\%} of the observations fall within
      $3\sigma$ of the mean $\mu$.
    \end{itemize}
    }
  }
\end{center}
\end{document}
```

Uses `hist.lsp` 46c, `68.lsp` 47b, `95.lsp` 48, `99.lsp` 49a, `smhist.lsp` 47a, and `stbl` 46a.
This code is used in chunk 2b.

Notice how this document needs a few `Lisp-Stat` programs for the hyper-links. Section 7 deals with them.

3 The Code

Let's first get the copyright out of the way.

5 ⟨*Copyright for code* 5⟩≡

```
;;;
;;; @(#)$Header$
;;;
;;; Copyright (C) 1994 B. Narasimhan, naras@euler.bd.psu.edu
;;;
;;; This program is free software; you can redistribute it and/or modify
;;; it under the terms of the GNU General Public License as published by
;;; the Free Software Foundation; either version 2 of the License, or
;;; (at your option) any later version.
;;;
;;; This program is distributed in the hope that it will be useful,
;;; but WITHOUT ANY WARRANTY; without even the implied warranty of
;;; MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
;;; GNU General Public License for more details.
;;;
;;; You should have received a copy of the GNU General Public License
;;; along with this program; if not, write to the Free Software
;;; Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
;;;
```

Defines:
`Copyright`, never used.
`FSF`, never used.
`GNU`, never used.
This code is used in chunks 6 and 46–50.

So, what must our code do? It should provide a user-friendly interface to the built-in distribution functions in `Lisp-Stat`. `Lisp-Stat` has many of the common distribution functions; however, one has to write `Lisp` phrases to use them. Since our intended audience is introductory statistics students, we want to insulate them from `Lisp`. Our code should provide a menu via which the user can choose a particular distribution as shown in figure1. Once a distribution is chosen, it should display a graph of the density along with button controls for calculating probabilities or quantiles. Inputs from the user should be solicited via informative dialogs. Answers to queries must be displayed in the margin of the graph along with parameters identifying the distribution. Finally, the code must also be extensible—if new distributions are added, it must be easy to make them available too.

Here is the main structure of our program.

6 ⟨*Code* 6⟩≡
⟨*Copyright for code* 5⟩
⟨*Program hist.lsp* 46c⟩
⟨*Program smhist.lsp* 47a⟩
⟨*Utility functions* 7a⟩
⟨*Additional methods for built-in prototypes* 9a⟩
⟨*Implementation constants* 17a⟩
⟨*Distribution prototype definition* 10⟩
⟨*Distribution prototype methods* 11a⟩
⟨*Button overlay prototype definition* 26a⟩
⟨*Button overlay prototype methods* 26b⟩
⟨*Normal distribution* 29a⟩
⟨*T distribution* 34⟩
⟨*Chi-square distribution* 38⟩
⟨*F distribution* 42⟩
⟨*Main function* 46a⟩
⟨*Invoke main function* 46b⟩
This code is used in chunk 2b.

3.1 Utility functions

Here are some simple but useful functions. I won't even bother to explain them, since they are pretty self-documenting.

7a ⟨*Utility functions* 7a⟩≡

```
(defun probability-p (x)
" Method args: (x)
Returns true if x is a number between 0 and 1, end-points included."
  (and (>= x 0.0) (<= x 1.0)))

(defun strict-probability-p (x)
" Method args: (x)
Returns true if x is a number between 0 and 1, end-points not included."
  (and (> x 0.0) (< x 1.0)))

(defun nonzero-probability-p (x)
" Method args: (x)
Returns true if x is a number between 0 and 1, 0 not included."
  (and (> x 0.0) (<= x 1.0)))

(defun nonunit-probability-p (x)
" Method args: (x)
Returns true if x is a number between 0 and 1, 1 not included."
  (and (>= x 0.0) (< x 1.0)))
```

Defines:
`nonzero-probability-p`, never used.
`probability-p`, never used.
`strict-probability-p`, never used.
This definition is continued in chunks 7, 8, and 28.
This code is used in chunks 6 and 49b.

The next function `new-xlispstat` is useful for writing code that is compatible with older versions of `Lisp-Stat`.

7b ⟨*Utility functions* 7a⟩+≡

```
(defun new-xlispstat ()
" Method args: none
Returns true if the version of xlispstat is 3.xx or greater."
  (and (boundp 'xls-major-release) (>= xls-major-release 3)))
```

Defines:
`new-xlispstat`, used in chunk 8a.

Often, we need to get the value of a string returned by a dialog.

8a ⟨*Utility functions* 7a⟩+≡
```
(defun val-of (str)
"Method args: (str)
Returns the value of str. If empty or invalid string, returns nil."
  (if (= (length str) 0)
      nil
    (if (new-xlispstat)
        (ignore-errors (read-from-string str))
      (unwind-protect
          (read (make-string-input-stream str))))))
```
Defines:
  `val-of`, used in chunk 8.
Uses `new-xlispstat` 7b.

The function `get-values-from` takes a list of `text-items` and returns a list of values from each item in the list. If any item has invalid values, no error is signalled, but a `nil` is returned for that entry.

8b ⟨*Utility functions* 7a⟩+≡
```
(defun get-values-from (list)
" Method args: (list)
List should be list of edit-text-items. A list of values from each of
them is returned."
  (mapcar #'(lambda(x) (val-of (send x :text))) list))
```
Defines:
  `get-values-from`, used in chunks 32, 36, 40, and 44.
Uses `val-of` 8a.

The function `get-numbers-from` is similar, but will signal an error if any value is not a number.

8c ⟨*Utility functions* 7a⟩+≡
```
(defun get-numbers-from (list)
" Method args: (list)
List should be list of edit-text-items. A list of values from each of
them is returned.  If any of them is invalid, an error is signalled."
  (let ((vals (mapcar #'(lambda(x) (val-of (send x :text))) list)))
     (unless (every #'numberp vals)
        (error "Non-numeric values in some items."))
     vals))
```
Defines:
  `get-numbers-from`, used in chunks 30, 34, 38, and 42.
Uses `val-of` 8a.

The functions `get-values-from` and `get-numbers-from` are useful in dialogs.

3.2 Additional Useful Methods

The following additional methods for the built-in `Lisp-Stat` prototypes turn out be surprisingly useful in writing good looking dialogs.

9a ⟨*Additional methods for built-in prototypes* 9a⟩≡

```
(defmeth text-item-proto :width (&optional width)
"Method args: (&optional wid)
Sets or retrieves the width of a text-item."
  (if width
      (let ((sz (slot-value 'size)))
        (setf (slot-value 'size) (list width (select sz 1))))
    (select (slot-value 'size) 0)))
```

Defines:
`:width`, used in chunks 30, 32, 34, 36, 38, 40, 42, and 44.
This definition is continued in chunk 9.
This code is used in chunks 6 and 49b.

9b ⟨*Additional methods for built-in prototypes* 9a⟩+≡

```
(defmeth edit-text-item-proto :width (&optional width)
"Method args: (&optional wid)
Sets or retrieves the width of a edit-text-item."
  (if width
      (let ((sz (slot-value 'size)))
        (setf (slot-value 'size) (list width (select sz 1))))
    (select (slot-value 'size) 0)))
```

Defines:
`:width`, used in chunks 30, 32, 34, 36, 38, 40, 42, and 44.

9c ⟨*Additional methods for built-in prototypes* 9a⟩+≡

```
(defmeth interval-scroll-item-proto :width (&optional width)
"Method args: (&optional width)
Sets or retrieves the width of an interval-scroll-item."
  (if width
      (let ((sz (slot-value 'size)))
        (setf (slot-value 'size) (list width (select sz 1))))
    (select (slot-value 'size) 0)))
```

Defines:
`:width`, used in chunks 30, 32, 34, 36, 38, 40, 42, and 44.

3.3 The `dist-plot-proto` Prototype

The prototype for defining various distributions is `dist-plot-proto`. The documentation string in the code provides some help for users of the code. Let us discuss the slots in detail. (You may want to look at the actual definition of the prototype below now.) The three slots `dens`, `cdf` and `icdf` respectively hold functions that return the density, cumulative distribution, and inverse cumulative distribution functions. We shall henceforth refer to these functions are f, F and F^{-1} respectively. The slot `params` holds the parameters of the distribution. Slots `l-point` and `r-point` hold two abscissa values between which the plot is shaded using `shade-color`. The plot is shaded only when at least one of them is non-`nil`. If `l-point` is non-`nil`, then the plot is shaded to the left of `l-point`. When `r-point` is non-`nil`, the plot is shaded to the right of `r-point`. If both are non-`nil`, then the plot is shaded between the two points. The slot `num-points` will indicate the number of points used in plotting. Every time a density plot is drawn, we want to display the parameters of the distribution on the plot so as to uniquely identify the distribution. The slot `params-print-format` will hold a string that indicates the format in which the parameters are to be printed. The slot `params-display-loc` will indicate where on the plot the parameters should be displayed. The slots `answer` and `answer-display-loc` hold analagous information for the answer string. Finally, since a distribution might have infinite support, we have to impose practical limits on the maximum and minimum probabilities. The density will only be drawn between $F^{-1}(p_0)$ and $F^{-1}(p_1)$ where p_0 and p_1 are the values contained in the slots `min-probability` and `max-probability` respectively.

Note that `dist-plot-proto` inherits from `scatterplot-proto` and so we have all the methods of `scatterplot-proto` available for `dist-plot-proto`. In particular, much of the graph drawing is really the responsibility of `scatterplot-proto`.

10 ⟨*Distribution prototype definition* 10⟩≡

```
(defproto dist-plot-proto
          '(dens cdf icdf params l-point r-point shade-color
                 num-points important-abscissae
                 params-print-format params-display-loc
                 answer answer-display-loc
                 min-probability max-probability)
           () scatterplot-proto
"The distribution plot prototype. The slots dens, cdf, and icdf
respectively hold the function that calculate the density,
cumulative distribution and inverse of the cumulative distribution
functions. The slot params holds the parameters for the
distribution. L-point and r-point hold values between which the
plot must be shaded under the density with shade-color. Num-points
indicates how many points must be used to plot the density. The slot
important-abscissae is a list of points that need to be highlighted.
The strings params-print-format and answer indicate how
the parameters and answers should be printed on the plot while the
loc-slots hold the location on the plot where they should be
printed. Min-probability and max-probability indicate the
```

```
  practical support of the density.")
```

Defines:
answer, used in chunks 12, 20b, 25a, 28, 30, 32, 34, 36, 38, 40, 42, 44, and 47–49.
answer-display-loc, used in chunks 12b and 20b.
cdf, used in chunks 13b, 16–18, 29b, 32, 33, 36, 37, 40, 41, 44, 45, and 47–49.
dens, used in chunks 13a, 16–18, 23–25, 29a, 33, 37, 41, 45, and 47–49.
dist-plot-proto, used in chunks 11–17, 20–22, 25, 33, 37, 41, 45, and 47–49.
icdf, used in chunks 13c, 16–18, 29c, 30, 33, 34, 37, 38, 41, 42, 45, and 47–49.
important-abscissae, used in chunks 14a, 22a, 25b, 30, 32, 34, 36, 38, 40, 42, 44, and 47–49.
l-point, used in chunks 14c, 22b, 30, 32, 34, 36, 38, 40, 42, 44, and 47–49.
max-probability, used in chunks 15–18.
min-probability, used in chunks 15, 17, and 18a.
num-points, used in chunks 13d, 17, 18a, 25b, 33, 37, 41, 45, and 47–49.
params, used in chunks 11, 12, 15–20, 25a, 30, 32–34, 36–38, 40–42, 44, 45, and 47–49.
params-display-loc, used in chunks 12d, 19b, and 20a.
params-print-format, used in chunks 12c, 17c, 18a, 20a, 33, 37, 41, 45, and 47–49.
r-point, used in chunks 14d, 22b, 30, 32, 34, 36, 38, 40, 42, 44, and 47–49.
shade-color, used in chunks 14b, 17, 18a, 23, and 24.
This code is used in chunks 6 and 50a.

3.4 Methods for dist-plot-proto

The methods for using dist-plot-proto can be categorized as follows.

11a ⟨*Distribution prototype methods* 11a⟩≡
⟨*Distribution prototype accessor and modifier methods* 11b⟩
⟨*Other useful methods for distribution prototype* 15c⟩
⟨*Distribution prototype :isnew method* 17c⟩
⟨*Distribution prototype redrawing methods* 25a⟩
This code is used in chunks 6 and 50a.

3.4.1 Accessor and Modifier Methods.

The following methods provide access to the slots of dist-plot-proto.

11b ⟨*Distribution prototype accessor and modifier methods* 11b⟩≡

```
(defmeth dist-plot-proto :params (&optional params)
  "Method args: (&optional params)
Sets or retrieves the parameters for the distribution."
  (if params
      (setf (slot-value 'params) params)
    (slot-value 'params)))
```

Defines:
:params, used in chunks 12, 15, 16, 19b, 20a, 30, 32–34, 36–38, 40–42, 44, 45, and 47–49.
Uses dist-plot-proto 10 and params 10.
This definition is continued in chunks 12–15.
This code is used in chunk 11a.

12a ⟨*Distribution prototype accessor and modifier methods* 11b⟩+≡

```
(defmeth dist-plot-proto :answer (&optional str)
  "Method args: (&optional str)
Sets or retrieves the answer string."
  (if str
      (setf (slot-value 'answer) str)
    (slot-value 'answer)))
```

Defines:
:answer, used in chunks 12b, 20b, 30, 32, 34, 36, 38, 40, 42, 44, and 47–49.
Uses answer 10 and dist-plot-proto 10.

12b ⟨*Distribution prototype accessor and modifier methods* 11b⟩+≡

```
(defmeth dist-plot-proto :answer-display-loc (&optional loc)
  "Method args: (&optional loc)
Sets or retrieves the answer-display-loc slot."
  (if loc
     (setf (slot-value 'answer-display-loc) loc)
  (slot-value 'answer-display-loc)))
```

Defines:
:answer-display-loc, used in chunk 20b.
Uses :answer 12a, answer 10, answer-display-loc 10, and dist-plot-proto 10.

12c ⟨*Distribution prototype accessor and modifier methods* 11b⟩+≡

```
(defmeth dist-plot-proto :params-print-format (&optional str)
  "Method args: (&optional str)
Sets or retrieves the parameter-print-format string."
  (if str
      (setf (slot-value 'params-print-format) str)
    (slot-value 'params-print-format)))
```

Defines:
:params-print-format, used in chunks 20a, 33, 37, 41, 45, and 47–49.
Uses dist-plot-proto 10, :params 11b, params 10, and params-print-format 10.

12d ⟨*Distribution prototype accessor and modifier methods* 11b⟩+≡

```
(defmeth dist-plot-proto :params-display-loc (&optional loc)
  "Method args: (&optional loc)
Sets or retrieves the params-display-loc slot."
  (if loc
     (setf (slot-value 'params-display-loc) loc)
  (slot-value 'params-display-loc)))
```

Defines:
:params-display-loc, used in chunks 19b and 20a.
Uses dist-plot-proto 10, :params 11b, params 10, and params-display-loc 10.

13a ⟨*Distribution prototype accessor and modifier methods* 11b⟩+≡

```
(defmeth dist-plot-proto :dens (&optional dens)
  "Method args: (&optional dens)
Sets or retrieves the density."
  (if dens
      (setf (slot-value 'dens) dens)
    (slot-value 'dens)))
```

Defines:
:dens, used in chunks 16b, 23–25, 33, 37, 41, 45, and 47–49.
Uses dens 10 and dist-plot-proto 10.

13b ⟨*Distribution prototype accessor and modifier methods* 11b⟩+≡

```
(defmeth dist-plot-proto :cdf (&optional cdf)
  "Method args: (&optional cdf)
Sets or retrieves the CDF."
  (if cdf
      (setf (slot-value 'cdf) cdf)
    (slot-value 'cdf)))
```

Defines:
:cdf, used in chunks 16c, 32, 33, 36, 37, 40, 41, 44, 45, and 47–49.
Uses cdf 10 and dist-plot-proto 10.

13c ⟨*Distribution prototype accessor and modifier methods* 11b⟩+≡

```
(defmeth dist-plot-proto :icdf (&optional icdf)
  "Method args: (&optional icdf)
Sets or retrieves the Inverse CDF."
  (if icdf
      (setf (slot-value 'icdf) icdf)
    (slot-value 'icdf)))
```

Defines:
:icdf, used in chunks 16d, 30, 33, 34, 37, 38, 41, 42, 45, and 47–49.
Uses dist-plot-proto 10 and icdf 10.

13d ⟨*Distribution prototype accessor and modifier methods* 11b⟩+≡

```
(defmeth dist-plot-proto :num-points (&optional num-points)
  "Method args: (&optional num-points)
Sets or retrieves the slot num-points."
  (if num-points
      (setf (slot-value 'num-points) num-points)
    (slot-value 'num-points)))
```

Defines:
:num-points, used in chunks 25b, 33, 37, 41, 45, and 47–49.
Uses dist-plot-proto 10 and num-points 10.

14a ⟨*Distribution prototype accessor and modifier methods* 11b⟩+≡

```
(defmeth dist-plot-proto :important-abscissae (&optional list)
  "Method args: (&optional list)
Sets or retrieves the list of important abscissae."
  (if list
      (setf (slot-value 'important-abscissae) list)
    (slot-value 'important-abscissae)))
```

Defines:
:important-abscissae, used in chunks 22a, 30, 32, 34, 36, 38, 40, 42, 44, and 47–49.
Uses dist-plot-proto 10 and important-abscissae 10.

14b ⟨*Distribution prototype accessor and modifier methods* 11b⟩+≡

```
(defmeth dist-plot-proto :shade-color (&optional color)
  "Method args: (&optional color)
Sets or retrieves the shading color."
  (if color
      (setf (slot-value 'shade-color) color)
    (slot-value 'shade-color)))
```

Defines:
:shade-color, used in chunks 23 and 24.
Uses dist-plot-proto 10 and shade-color 10.

14c ⟨*Distribution prototype accessor and modifier methods* 11b⟩+≡

```
(defmeth dist-plot-proto :l-point (&optional (point nil supplied-p))
  "Method args: (&optional (point nil supplied-p))
Sets or retrieves the slot l-point."
  (if supplied-p
      (setf (slot-value 'l-point) point)
    (slot-value 'l-point)))
```

Defines:
:l-point, used in chunks 22b, 30, 32, 34, 36, 38, 40, 42, 44, and 47–49.
Uses dist-plot-proto 10 and l-point 10.

14d ⟨*Distribution prototype accessor and modifier methods* 11b⟩+≡

```
(defmeth dist-plot-proto :r-point (&optional (point nil supplied-p))
  "Method args: (&optional (point nil supplied-p))
Sets or retrieves the slot r-point."
  (if supplied-p
      (setf (slot-value 'r-point) point)
    (slot-value 'r-point)))
```

Defines:
:r-point, used in chunks 22b, 30, 32, 34, 36, 38, 40, 42, 44, and 47–49.
Uses dist-plot-proto 10 and r-point 10.

15a ⟨*Distribution prototype accessor and modifier methods* 11b⟩+≡

```
(defmeth dist-plot-proto :min-probability (&optional val)
  "Method args: (&optional val)
Sets or retrieves the minimum probability for which quantiles
can be calculated."
  (if val
      (setf (slot-value 'min-probability) val)
    (slot-value 'min-probability)))
```

Defines:
:min-probability, used in chunk 15c.
Uses dist-plot-proto 10 and min-probability 10.

15b ⟨*Distribution prototype accessor and modifier methods* 11b⟩+≡

```
(defmeth dist-plot-proto :max-probability (&optional val)
  "Method args: (&optional val)
Sets or retrieves the maximum probability for which quantiles
can be calculated."
  (if val
      (setf (slot-value 'max-probability) val)
    (slot-value 'max-probability)))
```

Defines:
:max-probability, used in chunk 16a.
Uses dist-plot-proto 10 and max-probability 10.

3.4.2 Other Useful Methods.

The density must be drawn on $[x_{min}, x_{max}]$, where x_{min}=$F^{-1}(p_0)$ and x_{max}=$F^{-1}(p_1)$, p_0 and p_1 being the practical limits on the minimum and maximum probabilities. So it is useful to have two methods that return x_{min} and x_{max}.

15c ⟨*Other useful methods for distribution prototype* 15c⟩≡

```
(defmeth dist-plot-proto :xmin ()
  "Method args: None
Returns the minimum value of x used in plotting."
  (apply (slot-value 'icdf) (send self :min-probability)
         (send self :params)))
```

Defines:
:xmin, used in chunk 25b.
Uses dist-plot-proto 10, :min-probability 15a, min-probability 10, :params 11b, and params 10.
This definition is continued in chunks 16 and 20–24.
This code is used in chunk 11a.

16a ⟨*Other useful methods for distribution prototype* 15c⟩+≡

```
(defmeth dist-plot-proto :xmax ()
  "Method args: None
Returns the maximum value of x used in plotting."
  (apply (slot-value 'icdf) (send self :max-probability)
         (send self :params)))
```

Defines:
:xmax, used in chunk 25b.
Uses dist-plot-proto 10, :max-probability 15b, max-probability 10, :params 11b, and params 10.

We will also need to calculate the density, cdf, and icdf at a point.

16b ⟨*Other useful methods for distribution prototype* 15c⟩+≡

```
(defmeth dist-plot-proto :dens-at (x)
  "Method args: x
Returns the value of the density at x."
  (apply (slot-value 'dens) x (send self :params)))
```

Defines:
:dens-at, used in chunks 23–25.
Uses :dens 13a, dens 10, dist-plot-proto 10, :params 11b, and params 10.

16c ⟨*Other useful methods for distribution prototype* 15c⟩+≡

```
(defmeth dist-plot-proto :cdf-at (x)
  "Method args: x
Returns the value of the cdf at x."
  (apply (slot-value 'cdf) x (send self :params)))
```

Defines:
:cdf-at, used in chunks 32, 36, 40, and 44.
Uses :cdf 13b, cdf 10, dist-plot-proto 10, :params 11b, and params 10.

16d ⟨*Other useful methods for distribution prototype* 15c⟩+≡

```
(defmeth dist-plot-proto :icdf-at (x)
  "Method args: x
Returns the value of the icdf at x."
  (apply (slot-value 'icdf) x (send self :params)))
```

Defines:
:icdf-at, used in chunks 30, 34, 38, and 42.
Uses dist-plot-proto 10, :icdf 13c, icdf 10, :params 11b, and params 10.

3.4.3 The :isnew Method.

We are now ready to write our `:isnew` method. This method determines how we create an instance of the `dist-plot-proto` prototype. The mandatory arguments to this method are of course f, F, and F^{-1} specified via keyword arguments along with the parameters of the distribution, and two functions `prob-dialog` and `quant-dialog` which handle distribution specific dialogs. See also sections 5.1, 5.2, 5.3, and 5.4 for more on these dialogs. But first, some sensible defaults for p_0 and p_1.

17a ⟨*Implementation constants* 17a⟩≡
```
(defparameter *min-probability* 1e-7)
(defparameter *max-probability* (- 1 1e-7))
```
Defines:
`*max-probability*`, used in chunk 17c.
`*min-probability*`, used in chunk 17c.
Uses `max-probability` 10 and `min-probability` 10.
This definition is continued in chunks 17b and 21.
This code is used in chunks 6 and 50a.

to be the default p_0 and p_1 respectively. Let us also make the default number of points to be plotted to be 50 and the default shade color as `magenta`.

17b ⟨*Implementation constants* 17a⟩+≡
```
(defparameter *num-points* 50)
(defparameter *shade-color*
  (if (screen-has-color)
     'magenta
    'black))
```
Defines:
`*num-points*`, used in chunk 17c.
`*shade-color*`, used in chunk 17c.
Uses `num-points` 10 and `shade-color` 10.

Note that the `:isnew` method uses another prototype `button-overlay-proto` which is described in section4. Here is the beginning of the `:isnew` method with the arguments.

17c ⟨*Distribution prototype :isnew method* 17c⟩≡
```
(defmeth dist-plot-proto :isnew (&key dens cdf icdf params
          prob-dialog quant-dialog
          (shade-color *shade-color*)
          (params-print-format "Parameters: ~g")
          (num-points *num-points*)
          (max-probability *max-probability*)
          (min-probability *min-probability*)
          (title "Probability Density"))
```
Uses `cdf` 10, `dens` 10, `dist-plot-proto` 10, `icdf` 10, `:isnew` 19b 19b, `*max-probability*` 17a, `max-probability` 10, `*min-probability*` 17a, `min-probability` 10, `*num-points*` 17b, `num-points` 10, `params` 10, `params-print-format` 10, `*shade-color*` 17b, and `shade-color` 10.
This definition is continued in chunks 18 and 19.
This code is used in chunk 11a.

Now, on to the basic tasks the `:isnew` method must perform. Clearly, we need to store the supplied arguments in the respective slots.

18a ⟨*Distribution prototype :isnew method* 17c⟩+≡

```
(setf (slot-value 'dens) dens)
(setf (slot-value 'cdf) cdf)
(setf (slot-value 'icdf) icdf)
(setf (slot-value 'shade-color) shade-color)
(setf (slot-value 'params) params)
(setf (slot-value 'params-print-format) params-print-format)
(setf (slot-value 'num-points) num-points)
(setf (slot-value 'min-probability) min-probability)
(setf (slot-value 'max-probability) max-probability)
```

Uses `cdf` 10, `dens` 10, `icdf` 10, `max-probability` 10, `min-probability` 10, `num-points` 10, `params` 10, `params-print-format` 10, and `shade-color` 10.

Then, we need to invoke the *inherited* `:isnew` method of `scatterplot-proto` to actually draw the graphical window.

18b ⟨*Distribution prototype :isnew method* 17c⟩+≡

```
(call-next-method 2 :title title)
(when (screen-has-color) (send self :use-color t))
```

We add two button overlays to the plot for calculating probabilities and quantiles. We need to figure out the locations of these buttons on the window. A look at figure 1 might help. We shall arrange it so that the two buttons are on the left and right side of the window respectively. The `:isnew` method for `button-overlay-proto` just requires the coordinates of the left upper corner of a rectangular box surrounding the button along with the string to be displayed and a function which is called when the button is pressed.

18c ⟨*Distribution prototype :isnew method* 17c⟩+≡

```
(let* ((em (send self :text-width "m"))
       (ascent (send self :text-ascent))
       (descent (send self :text-descent))
       (prob (send button-overlay-proto :new em ascent
                   "Find Probability"
                   #'(lambda() (funcall prob-dialog self))))
       (cw (send self :canvas-width))
       (qtw (send self :text-width "Find Quantile"))
       (quant (send button-overlay-proto :new (- cw em qtw em) ascent
                    "Find Quantile"
                    #'(lambda() (funcall quant-dialog self)))))
```

Before we add the overlays to the plot, we need to allow enough space in the margins for the display of the buttons, parameters and the answer. So we make the window slightly bigger to accommodate these quantities. Each line of text will occupy at most `ascent + descent` vertical space. For example, the buttons will occupy y `+ ascent + descent + em` vertical space where y is the y-coordinate of the top-left corner of the button. (The extra `em` is due to the fact that there is a gap of $0.5 \times$ `em` all around the text in the box). Let us also assume that each line of text takes up $1.5\times$`(ascent + descent)` vertical space. So we can calculate how much space we need in the margin at the top. We also need some space in the bottom for drawing arrows that identify the quantiles. Taking all these things into consideration, we are led to the following code.

19a ⟨*Distribution prototype :isnew method* 17c⟩+≡

```
(let* ((sz (send self :size))
       (ht (select sz 1))
       (top-margin (round (+ ascent ascent descent em
                             (* 1.5 (+ ascent descent))
                             (* 1.5 (+ ascent descent)))))
       (bot-margin (round (* 1.5 (+ ascent descent)))))
  (send self :size (select sz 0) (+ ht top-margin bot-margin))
  (send self :margin 0 top-margin 0 bot-margin))
(send self :add-overlay prob)
(send self :add-overlay quant)
```

Finally, we need to determine the locations where the parameters and the answer must be displayed. The answer string will be displayed centered horizontally in the window, and so its location will just be the y coordinate.

19b ⟨*Distribution prototype :isnew method* 17c⟩+≡

```
(send self :params-display-loc
      (list em (+ ascent ascent descent em ascent descent)))
(setf (slot-value 'answer-display-loc)
      (+ ascent ascent descent em
         (round (* 1.5 (+ ascent descent)))
         ascent descent)))
```

Defines:
`:isnew`, used in chunk 17c.
Uses `:params` 11b, `params` 10, `:params-display-loc` 12d, and `params-display-loc` 10.

3.4.4 Redrawing Methods.

To make sure that the plot redraws itself when it is moved around on the screen, we need to write a `:redraw` method. Several sub-tasks have to be addressed. These involve shading under the density, displaying the parameters, and displaying the answer. In addition, we may have to highlight certain quantiles along the x-axis. (By highlighting, we mean drawing arrows that point to the values.) It is best to write these sub-tasks as methods. The methods for displaying the parameters and the answer are easy.

20a ⟨*Other useful methods for distribution prototype* 15c⟩+≡

```
(defmeth dist-plot-proto :display-params ()
  "Method args: (None)
Displays the parameters on the plot."
   (when (send self :params-display-loc)
      (let ((str (apply #'format nil (send self :params-print-format)
                        (send self :params))))
        (apply #'send self :draw-string str
               (send self :params-display-loc)))))
```

Defines:
`:display-params`, used in chunk 25a.
Uses `dist-plot-proto` 10, `:params` 11b, `params` 10, `:params-display-loc` 12d, `params-display-loc` 10, `:params-print-format` 12c, and `params-print-format` 10.

20b ⟨*Other useful methods for distribution prototype* 15c⟩+≡

```
(defmeth dist-plot-proto :display-answer ()
  "Method args: (None)
Displays the answer centered horizontally on the plot."
  (let ((y (send self :answer-display-loc))
        (answer (send self :answer)))
     (when answer
        (let ((x (round (* 0.5 (- (send self :canvas-width)
                                  (send self :text-width answer))))))
           (send self :draw-string answer x y)))))
```

Defines:
`:display-answer`, used in chunk 25a.
Uses `:answer` 12a, `answer` 10, `:answer-display-loc` 12b, `answer-display-loc` 10, and `dist-plot-proto` 10.

It is also straight-forward to write a method for drawing a vertical arrow from point (a, b) to (a, c).

21a ⟨*Other useful methods for distribution prototype* 15c⟩+≡

```
(defmeth dist-plot-proto :draw-vert-arrow (a b c)
  "Method args: (a b c)
Draws a vertical arrow from (a b) ending at (a c). The arrow head will
be at (a c). The coordinates must be canvas coordinates."
  (send self :draw-line a b a c)
  (let* ((p (+ b (round (* 0.8 (- c b)))))
         (x1 (- a 10))
         (x2 (+ a 10)))
    (send self :paint-poly (list (list x1 p) (list x2 p) (list a c)))))
```

Defines:
`:draw-vert-arrow`, used in chunk 22a.
Uses `dist-plot-proto` 10.

Using the method for drawing vertical arrows, we can now highlight important x-values. We need the following constant.

21b ⟨*Implementation constants* 17a⟩+≡

```
(defparameter *quantile-print-format* "~,3f")
```

Defines:
`*quantile-print-format*`, used in chunks 22a and 28.

Not surprisingly, we need a similar one for probability later; we might as well define it here.

21c ⟨*Implementation constants* 17a⟩+≡

```
(defparameter *probability-print-format* "~,3f")
```

Defines:
`*probability-print-format*`, used in chunk 28.

So here is our method for highlighting some abscissae.

22a ⟨*Other useful methods for distribution prototype* 15c⟩+≡

```
(defmeth dist-plot-proto :highlight-important-abscissae ()
  "Method args: None
Draws a vertical arrows highlighting the abscissae in the slot
important-abscissae."
  (when (send self :important-abscissae)
        (let* ((list (send self :important-abscissae))
               (ascent (send self :text-ascent))
               (descent (send self :text-descent))
               (ht (send self :canvas-height))
               (arrow-start-y (- ht (round (* 1.5 (+ ascent descent)))))
               (str-start-y (- ht (round (* 0.35 (+ ascent descent))))))
          (dolist (val list)
            (let* ((coord (send self :real-to-canvas val 0.0))
                   (x (select coord 0))
                   (y (select coord 1))
                   (str (format nil *quantile-print-format* val))
                   (str-wid (send self :text-width str)))
              (send self :draw-string str
                (- x (round (* 0.5 str-wid))) str-start-y)
              (send self :draw-vert-arrow x arrow-start-y y))))))
```

Defines:
:highlight-important-abscissae, used in chunk 25b.
Uses dist-plot-proto 10, :draw-vert-arrow 21a, :important-abscissae 14a, important-abscissae 10, and *quantile-print-format* 21b.

Let us now tackle shading. Recall that if l-point is non-nil, we want to shade to the left, and if r-point is non-nil, we want to shade to the right, or if both are non-nil, we want to shade between. If both are nil, then we must skip shading.

22b ⟨*Other useful methods for distribution prototype* 15c⟩+≡

```
(defmeth dist-plot-proto :shade-under-plot ()
  "Shades the region under the curve determined by l-point and r-point.
If both are non-nil, shades between, otherwise to the left or right
as the case may be. Does nothing if both l-point, r-point are
nil. Note that a must be < b if both are non-nil."
  (when (or (send self :l-point) (send self :r-point))
     (let ((x (mapcar #'(lambda(x) (send self :linestart-coordinate 0 x))
                      (iseq (send self :num-lines))))
           (y (mapcar #'(lambda(x) (send self :linestart-coordinate 1 x))
                      (iseq (send self :num-lines))))
           (a (send self :l-point))
           (b (send self :r-point)))
```

Uses dist-plot-proto 10, :l-point 14c, l-point 10, :r-point 14d, r-point 10, and :shade-under-plot 24.

So how are we going to handle shading between two points? Well, the picture[1] in figure 2 illustrates the issues.

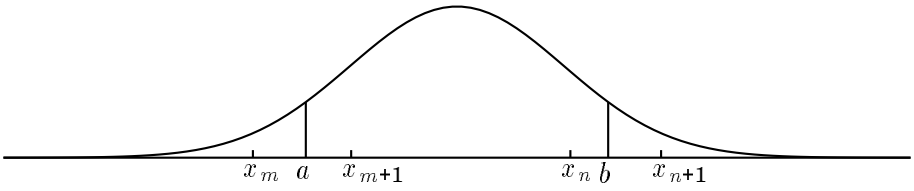

Figure 2: The Normal density function.

Suppose we wish to shade between a and b where $a < b$. Let $x_1, x_2, \ldots, x_k$ denote the sequence of x-values which were used in plotting the density. Set $x_0 = -\infty$ and $x_{k+1} = \infty$. Then, we need to find indices m and n such that $x_m < a \leq x_{m+1}$ and $x_n \leq b < x_{n+1}$. We need to shade the polygonal region $(a, 0)$, $(a, f(a))$, $(x_{m+1}, f(x_{m+1}))$, $\ldots$, $(x_n, f(x_n))$, $(b, f(b))$, $(b, 0)$. Also, since a and b will be in real coordinates, we have to convert them to canvas coordinates. There is a subtle issue one has to watch for when shading: the shading must be done *after* adjusting the plot to the newly redrawn density.

23 ⟨*Other useful methods for distribution prototype* 15c⟩+≡

```
(cond
 ((and a b) ; We need to shade between.
  (let ((m+1 (position a x :test #'<=))
        (n (position b x :test #'>= :from-end t)))
    (unless (and m+1 n (>= n m+1))
      (error "Bad left and right end points."))
    (let ((v-list (list
                   (send self :real-to-canvas a 0)
                   (send self :real-to-canvas a
                         (send self :dens-at a))))
          (middle
           (select (mapcar #'(lambda(x y)
                    (send self :real-to-canvas x y)) x y)
                   (iseq m+1 n)))
          (end (list
                (send self :real-to-canvas b
                      (send self :dens-at b))
                (send self :real-to-canvas b 0)))
          (dc (send self :draw-color)))
      (setf v-list (append v-list middle))
      (setf v-list (append v-list end))
      (send self :draw-color (send self :shade-color))
      (send self :paint-poly v-list)
      (send self :draw-color dc))))
```

[1] With perhaps the faint hope that Art + Literate Programming=Art of Computer Programming...

Uses :dens 13a, dens 10, :dens-at 16b, :shade-color 14b, and shade-color 10.

Next, we need to handle shading to the left or shading to the right. In light of the above discussion, these two tasks are straight-forward.

24 ⟨*Other useful methods for distribution prototype* 15c⟩+≡

```
(a ; We need to shade to the left.
 (let ((m+1 (position a x :test #'>= :from-end t)))
   (when m+1
     (let ((v-list
            (list
              (send self :real-to-canvas (select x 0) 0.0)))
           (middle
            (select (mapcar #'(lambda(x y)
                      (send self :real-to-canvas x y)) x y)
              (iseq m+1)))
            (end (list
                  (send self :real-to-canvas a
                        (send self :dens-at a))
                  (send self :real-to-canvas a 0.0)))
           (dc (send self :draw-color)))
       (setf v-list (append v-list middle))
       (setf v-list (append v-list end))
       (send self :draw-color (send self :shade-color))
       (send self :paint-poly v-list)
       (send self :draw-color dc)))))
(b ; We need to shade to the right.
 (let ((n (position b x :test #'<)))
   (when n
     (let ((v-list
            (list
              (send self :real-to-canvas b 0)
              (send self :real-to-canvas b (send self
              :dens-at b))))
           (end
            (select (mapcar #'(lambda(x y)
                       (send self :real-to-canvas x y)) x y)
                    (iseq n (1- (send self :num-lines)))))
           (dc (send self :draw-color)))
       (setf v-list (append v-list end))
       (send self :draw-color (send self :shade-color))
       (send self :paint-poly v-list)
       (send self :draw-color dc)))))))
```

Defines:
:shade-under-plot, used in chunks 22b and 25b.
Uses :dens 13a, dens 10, :dens-at 16b, :shade-color 14b, and shade-color 10.

That concludes the `:shade-under-plot` method.

Next, the `:redraw-content` method, which redraws the contents of the plot. Basically, we have to clear everything in the plot, redraw the density, adjust the axes, redraw the axes, shade under the plot, display the parameters of the distribution and the answer. Since the `:redraw-background` method is responsible for drawing the background, we write a new `:redraw-background` method for `dist-plot-proto`.

25a ⟨*Distribution prototype redrawing methods* 25a⟩≡

```
(defmeth dist-plot-proto :redraw-background ()
  (call-next-method)
  (send self :display-params)
  (send self :display-answer))
```

Defines:
`:redraw-background`, never used.
Uses `answer` 10, `:display-answer` 20b, `:display-params` 20a, `dist-plot-proto` 10, `params` 10, and `:redraw` 27a.
This definition is continued in chunk 25b.
This code is used in chunk 11a.

And, at last, our `:redraw-content` method.

25b ⟨*Distribution prototype redrawing methods* 25a⟩+≡

```
(defmeth dist-plot-proto :redraw-content ()
  "Method args: none
Redraws the content of the plot and the background."
  (call-next-method)
  (send self :clear-lines :draw nil)
  (let* ((x (rseq (send self :xmin) (send self :xmax)
                  (send self :num-points)))
         (y (mapcar #'(lambda(w) (send self :dens-at w)) x)))
    (send self :add-lines x y :draw nil))
  (send self :adjust-to-data :draw nil)
  (send self :shade-under-plot)
  (send self :highlight-important-abscissae))
```

Defines:
`:redraw-content`, never used.
Uses `:dens` 13a, `dens` 10, `:dens-at` 16b, `dist-plot-proto` 10, `:highlight-important-abscissae` 22a, `important-abscissae` 10, `:num-points` 13d, `num-points` 10, `:redraw` 27a, `:shade-under-plot` 24, `:xmax` 16a, and `:xmin` 15c.

4 The Button Overlay

The button overlay is really quite simple. The prototype has six slots, `llx`, the x coordinate of the left lower corner of the button, `ury`, the y coordinate of the upper right corner of the button, `urx` the x coordinate of the upper right corner and `lly` the y coordinate of the lower left corner. The `title` slot holds the text to be displayed in the button, while `action` is a function that is invoked when the button is pressed with a mouse.

26a ⟨*Button overlay prototype definition* 26a⟩≡
```
(defproto button-overlay-proto '(llx ury urx lly title action) ()
  graph-overlay-proto
  "The button overlay prototype. Title is the title displayed on the
  button, and action is the function that is called when the mouse is
  clicked in the box. The slots llx and ury hold the lower left and
  upper right coordinates of the box.")
```
Defines:
  `:button-overlay-proto`, never used.
This code is used in chunks 6 and 49c.

4.1 The Overlay Methods

The `:isnew` method which basically records the given arguments in the respective slots and invokes the inherited `:isnew` method of `graph-overlay-proto`.

26b ⟨*Button overlay prototype methods* 26b⟩≡
```
(defmeth button-overlay-proto :isnew (llx ury title action)
  "Method args: (llx ury title action)
Title is a string, action is a function that is invoked when
the mouse is clicked in the box, llx and ury are the coordinates of
the lower left and upper right x and y respectively. "
  (setf (slot-value 'action) action)
  (setf (slot-value 'title) title)
  (setf (slot-value 'llx) llx)
  (setf (slot-value 'ury) ury)
  (call-next-method))
```
Defines:
  `:isnew`, used in chunk 17c.
This definition is continued in chunk 27.
This code is used in chunks 6 and 49c.

The `:redraw` method below ensures that the button is drawn properly. It also calculates `urx` and `lly` for later use in the `:do-click` method.

27a ⟨*Button overlay prototype methods* 26b⟩+≡

```
(defmeth button-overlay-proto :redraw ()
  "Method args: none.
This method redraws the overlay."
  (let* ((graph (send self :graph))
         (em (send graph :text-width "m"))
         (gap (round (* .5 em)))
         (title (slot-value 'title))
         (llx (slot-value 'llx))
         (ury (slot-value 'ury))
         (tw (send graph :text-width title))
         (wid (+ gap tw gap))
         (ht (+ gap (send graph :text-ascent) (send graph :text-descent)
                gap)))
    (setf (slot-value 'urx) (+ llx wid))
    (setf (slot-value 'lly) (+ ury ht))
    (send graph :draw-string title (+ llx gap)  (+ ury (- ht gap)))
    (send graph :frame-rect llx ury wid ht)))
```

Defines:
`:redraw`, used in chunks 25, 30, 32, 34, 36, 38, 40, 42, 44, and 47–49.

The final method for `button-overlay-proto` invokes the function in the `action` slot when the mouse is clicked on the button.

27b ⟨*Button overlay prototype methods* 26b⟩+≡

```
(defmeth button-overlay-proto :do-click (x y m1 m2)
  "Method args: x y m1 m2
This method invokes the function in the action slot when the mouse is
clicked on the button."
  (let ((llx (slot-value 'llx))
        (ury (slot-value 'ury))
        (urx (slot-value 'urx))
        (lly (slot-value 'lly)))
    (when (and (< llx x urx) (< ury y lly))
          (funcall (slot-value 'action))
          t)))
```

Defines:
`:do-click`, never used.

5 Various Distributions

So it is time now to add various distributions along with the distribution specific dialogs. The following two functions `quantile-answer` and `probability-answer` return standard answers to questions.

28a ⟨*Utility functions* 7a⟩+≡

```
(defun quantile-answer (probability quantile)
"Method args: (probability quantile)
Returns a string for a quantile answer."
  (let ((fstr (concatenate 'string
               "ANSWER: The " *probability-print-format* "-quantile is "
               *quantile-print-format* ".")))
     (format nil fstr probability quantile)))
```

Defines:
`:quantile-answer`, never used.
Uses answer 10, `*probability-print-format*` 21c, and `*quantile-print-format*` 21b.

28b ⟨*Utility functions* 7a⟩+≡

```
(defun probability-answer (probability x1 x2)
"Method args: (probability x1 x2)
Returns a string for a probability answer."
  (if (and x2 (not x1))
     (let ((fstr (concatenate 'string
                  "ANSWER: The probability to the right of "
                  *quantile-print-format* " is "
                  *probability-print-format* ".")))
                  (format nil fstr x2 probability))
    (if (and x1 x2)
       (let ((fstr (concatenate 'string
                    "ANSWER: The probability between "
                    *quantile-print-format* " and "
                    *quantile-print-format* " is "
                    *probability-print-format* ".")))
              (format nil fstr x1 x2 probability))
     (let ((fstr (concatenate 'string
                  "ANSWER: The probability to the left of "
                  *quantile-print-format* " is "
                 *probability-print-format* ".")))
          (format nil fstr x1 probability)))))
```

Defines:
`:probability-answer`, never used.
Uses answer 10, `*probability-print-format*` 21c, and `*quantile-print-format*` 21b.

Let us begin with the most famous of them all, the normal distribution.

5.1 The Normal Distribution

Let us begin by defining the normal density, normal cdf, and inverse of the normal cdf functions. We have to watch the names we give these functions since `Lisp-Stat` has the densities built into it for common distributions. However, they are standard ones.

29a ⟨*Normal distribution* 29a⟩≡

```
(defun gaussian-density (x mu sigma)
  "Method args: (x mu sigma)
Returns the normal density with mean mu and std. dev. sigma at x."
  (/ (normal-dens (/ (- x mu) sigma)) sigma))
```

Defines:
`gaussian-density`, never used.
Uses `dens` 10.
This definition is continued in chunks 29, 30, 32, and 33.
This code is used in chunks 6 and 50b.

29b ⟨*Normal distribution* 29a⟩+≡

```
(defun gaussian-cdf (x mu sigma)
  "Method args: (x mu sigma)
Returns the normal cdf with mean mu and std. dev. sigma at x."
  (normal-cdf (/ (- x mu) sigma)))
```

Defines:
`gaussian-cdf`, never used.
Uses `cdf` 10.

29c ⟨*Normal distribution* 29a⟩+≡

```
(defun gaussian-icdf (x mu sigma)
  "Method args: (x mu sigma)
Returns the x-th normal quantile with mean mu and std. dev. sigma."
  (+ (* sigma (normal-quant x)) mu))
```

Defines:
`gaussian-cdf`, never used.
Uses `icdf` 10.

Here is the dialog for finding normal-quantiles. The code is pretty standard, except for the fact that before installing the text-items in the dialog, we make all dialog-items have the maximum width, so that they line up nicely in the dialog.

30 ⟨*Normal distribution* 29a⟩+≡

```
(defun normal-quant-dialog (dist)
  "Dialog for the quantiles of the Normal Distribution."
  (let* ((params (send dist :params))
         (prompt (send text-item-proto :new
                   (format nil
                     "To find Normal Distribution Quantiles,~%~
                      complete all fields and press OK.~%")))
         (mean-label (send text-item-proto :new "Mean"))
         (mean-val (send edit-text-item-proto :new
                         (format nil "~a" (select params 0)) :text-length 10))
         (sd-label (send text-item-proto :new "Std. Dev."))
         (sd-val (send edit-text-item-proto :new
                       (format nil "~a" (select params 1)) :text-length 10))
         (prob-label (send text-item-proto :new "Probability"))
         (prob-val (send edit-text-item-proto :new "" :text-length 10))
         (olist (list mean-label mean-val sd-label sd-val
                      prob-label prob-val))
         (mwid (max (mapcar #'(lambda(x) (send x :width)) olist)))
         (ok (send modal-button-proto :new "OK"
                   :action
                   #'(lambda ()
                      (let* ((inputs (get-numbers-from
                                       (list mean-val sd-val prob-val)))
                             (params (select inputs '(0 1)))
                             (prob (select inputs 2)))
                        (send dist :params params)
                        (send dist :l-point (send dist :icdf-at prob))
                        (send dist :important-abscissae
                              (list (send dist :l-point)))
                        (send dist :r-point nil)
                        (send dist :answer
                             (quantile-answer prob (send dist :l-point)))
                        (send dist :adjust-to-data)
                        (send dist :redraw)))))
         (cancel (send button-item-proto :new "Cancel"
                       :action
                       #'(lambda()
                           (send (send cancel :dialog)
                                 :modal-dialog-return nil)))))
    (dolist (x olist)
            (send x :width mwid))
```

```
(send (send modal-dialog-proto
            :new (list prompt
                       (list mean-label sd-label)
                       (list mean-val sd-val)
                       (list prob-label prob-val)
                       (list ok cancel))
            :title "Normal Quantile Dialog") :modal-dialog)))
```

Defines:
  normal-quantile-dialog, never used.
Uses :answer 12a, answer 10, get-numbers-from 8c, :icdf 13c, icdf 10, :icdf-at 16d, :important-abscissae 14a, important-abscissae 10, :l-point 14c, l-point 10, :params 11b, params 10, :redraw 27a, :r-point 14d, r-point 10, and :width 9a 9a 9a.

Next, the dialog for finding normal probabilities.

32 ⟨*Normal distribution* 29a⟩+≡

```
(defun normal-prob-dialog (dist)
  "Dialog for the probabilities of the Normal Distribution."
  (let* ((params (send dist :params))
         (prompt (send text-item-proto :new
                   (format nil
                     "To find Normal Distribution Probabilities,~%~
                      complete appropriate fields and press OK.~%~
                      For probability to the left, use Left Point;~%~
                      for probability to the right, use Right Point;~%~
                      for probability between, use both.~%")))
         (mean-label (send text-item-proto :new "Mean"))
         (mean-val (send edit-text-item-proto :new
                         (format nil "~a" (select params 0)) :text-length 10))
         (sd-label (send text-item-proto :new "Std. Dev."))
         (sd-val (send edit-text-item-proto :new
                       (format nil "~a" (select params 1)) :text-length 10))
         (l-label (send text-item-proto :new "Left Point"))
         (l-val (send edit-text-item-proto :new "" :text-length 10))
         (r-label (send text-item-proto :new "Right Point"))
         (r-val (send edit-text-item-proto :new "" :text-length 10))
         (olist (list mean-label mean-val sd-label sd-val
                      l-label l-val r-label r-val))
         (mwid (max (mapcar #'(lambda(x) (send x :width)) olist)))
         (ok (send modal-button-proto :new "OK"
                   :action
                   #'(lambda ()
                      (let* ((inputs (get-values-from
                                       (list mean-val sd-val l-val r-val)))
                             (params (select inputs '(0 1)))
                             (lx (select inputs 2))
                             (rx (select inputs 3))
                             (between (and rx lx))
                             (left (and lx (not rx)))
                             (prob (if between
                                       (- (send dist :cdf-at rx)
                                          (send dist :cdf-at lx))
                                     (if left
                                        (send dist :cdf-at lx)
                                       (- 1 (send dist :cdf-at rx))))))
                        (send dist :params params)
                        (send dist :l-point lx)
                        (send dist :r-point rx)
                        (send dist :important-abscissae
```

```
                                (if between
                                    (list lx rx)
                                  (if left
                                      (list lx)
                                    (list rx))))
                            (send dist :answer
                                 (probability-answer prob lx rx))
                            (send dist :redraw)))))
          (cancel (send button-item-proto :new "Cancel"
                        :action
                        #'(lambda()
                            (send (send cancel :dialog)
                                  :modal-dialog-return nil)))))
      (dolist (x olist)
              (send x :width mwid))
      (send (send modal-dialog-proto
                  :new (list prompt
                             (list mean-label sd-label)
                             (list mean-val sd-val)
                             (list l-label l-val)
                             (list r-label r-val)
                             (list ok cancel))
                  :title "Normal Probability Dialog") :modal-dialog)))
```

Defines:
  `normal-proba-dialog`, never used.
Uses `:answer` 12a, `answer` 10, `:cdf` 13b, `cdf` 10, `:cdf-at` 16c, `get-values-from` 8b, `:important-abscissae` 14a, `important-abscissae` 10, `:l-point` 14c, `l-point` 10, `:params` 11b, `params` 10, `:redraw` 27a, `:r-point` 14d, `r-point` 10, and `:width` 9a 9a 9a.

We write one last function that creates an instance of `dist-plot-proto` for the normal distribution.

33 ⟨*Normal distribution* 29a⟩+≡

```
(defun normal-distribution ()
  (send dist-plot-proto :new
        :dens #'gaussian-density
        :cdf #'gaussian-cdf
        :icdf #'gaussian-icdf
        :params (list 0 1)
        :params-print-format "Parameters: Mean = ~,3f, Std. Dev. = ~,3f"
        :prob-dialog #'normal-prob-dialog
        :quant-dialog #'normal-quant-dialog
        :num-points 50
        :title "Normal Distribution"))
```

Defines:
  `normal-distribution`, never used.
Uses `:cdf` 13b, `cdf` 10, `:dens` 13a, `dens` 10, `dist-plot-proto` 10, `:icdf` 13c, `icdf` 10, `:num-points` 13d, `num-points` 10, `:params` 11b, `params` 10, `:params-print-format` 12c, and `params-print-format` 10.

5.2 Student's t-distribution

This is pretty easy. We don't have to worry about the density as we did for the normal, since we are not going to muck around with non-central t distributions for now. Here is the dialog for finding t-quantiles.

34 ⟨*T distribution* 34⟩≡

```
(defun t-quant-dialog (dist)
  "Dialog for the quantiles of the T Distribution."
  (let* ((params (send dist :params))
         (prompt (send text-item-proto :new
                   (format nil
                     "To find T-Distribution Quantiles,~%~
                      complete all fields and press OK.~%")))
         (df-label (send text-item-proto :new "Degrees of Freedom"))
         (df-val (send edit-text-item-proto :new
                       (format nil "~a" (select params 0)) :text-length 10))
         (prob-label (send text-item-proto :new "Probability"))
         (prob-val (send edit-text-item-proto :new "" :text-length 10))
         (olist (list df-label df-val prob-label prob-val))
         (mwid (max (mapcar #'(lambda(x) (send x :width)) olist)))
         (ok (send modal-button-proto :new "OK"
                   :action
                   #'(lambda ()
                      (let* ((inputs (get-numbers-from
                                      (list df-val prob-val)))
                             (params (select inputs '(0)))
                             (prob (select inputs 1)))
                        (send dist :params params)
                        (send dist :l-point (send dist :icdf-at prob))
                        (send dist :important-abscissae
                              (list (send dist :l-point)))
                        (send dist :r-point nil)
                        (send dist :answer
                             (quantile-answer prob (send dist :l-point)))
                        (send dist :adjust-to-data)
                        (send dist :redraw)))))
         (cancel (send button-item-proto :new "Cancel"
                       :action
                       #'(lambda()
                           (send (send cancel :dialog)
                                 :modal-dialog-return nil)))))
    (dolist (x olist)
            (send x :width mwid))
    (send (send modal-dialog-proto
                :new (list prompt
```

```
                        (list df-label prob-label)
                        (list df-val prob-val)
                        (list ok cancel))
                :title "T Quantile Dialog") :modal-dialog)))
```

Defines:
  t-quant-dialog, never used.
Uses :answer 12a, answer 10, get-numbers-from 8c, :icdf 13c, icdf 10, :icdf-at 16d, :important-abscissae 14a, important-abscissae 10, :l-point 14c, l-point 10, :params 11b, params 10, :redraw 27a, :r-point 14d, r-point 10, and :width 9a 9a 9a.
This definition is continued in chunks 36 and 37.
This code is used in chunks 6 and 50b.

Here is the dialog for finding t-probabilities.

36 ⟨*T distribution* 34⟩+≡

```
(defun t-prob-dialog (dist)
  "Dialog for the probabilities of the T Distribution."
  (let* ((params (send dist :params))
         (prompt (send text-item-proto :new
                   (format nil
                     "To find T-Distribution Probabilities,~%~
                      complete appropriate fields and press OK.~%~
                      For probability to the left, use Left Point;~%~
                      for probability to the right, use Right Point;~%~
                      for probability between, use both.~%")))
         (df-label (send text-item-proto :new "Degrees of Freedom"))
         (df-val (send edit-text-item-proto :new
                       (format nil "~a" (select params 0)) :text-length 10))
         (l-label (send text-item-proto :new "Left Point"))
         (l-val (send edit-text-item-proto :new "" :text-length 10))
         (r-label (send text-item-proto :new "Right Point"))
         (r-val (send edit-text-item-proto :new "" :text-length 10))
         (olist (list df-label df-val l-label l-val r-label r-val))
         (mwid (max (mapcar #'(lambda(x) (send x :width)) olist)))
         (ok (send modal-button-proto :new "OK"
                   :action
                   #'(lambda ()
                      (let* ((inputs (get-values-from
                                       (list df-val l-val r-val)))
                             (params (select inputs '(0)))
                             (lx (select inputs 1))
                             (rx (select inputs 2))
                             (between (and rx lx))
                             (left (and lx (not rx)))
                             (prob (if between
                                       (- (send dist :cdf-at rx)
                                          (send dist :cdf-at lx))
                                     (if left
                                        (send dist :cdf-at lx)
                                      (- 1 (send dist :cdf-at rx))))))
                        (send dist :params params)
                        (send dist :l-point lx)
                        (send dist :r-point rx)
                        (send dist :important-abscissae
                              (if between
                                  (list lx rx)
                                (if left
                                    (list lx)
```

```
                                    (list rx))))
                            (send dist :answer
                                 (probability-answer prob lx rx))
                            (send dist :redraw)))))
          (cancel (send button-item-proto :new "Cancel"
                        :action
                        #'(lambda()
                            (send (send cancel :dialog)
                                  :modal-dialog-return nil)))))
     (dolist (x olist)
             (send x :width mwid))
     (send (send modal-dialog-proto
                 :new (list prompt
                            (list df-label df-val)
                            (list l-label l-val)
                            (list r-label r-val)
                            (list ok cancel))
                 :title "T Probability Dialog") :modal-dialog)))
```

Defines:
  `t-prob-dialog`, never used.
Uses `:answer` 12a, `answer` 10, `:cdf` 13b, `cdf` 10, `:cdf-at` 16c, `get-values-from` 8b, `:important-abscissae` 14a, `important-abscissae` 10, `:l-point` 14c, `l-point` 10, `:params` 11b, `params` 10, `:redraw` 27a, `:r-point` 14d, `r-point` 10, and `:width` 9a 9a 9a.

And a function that creates an instance of `dist-plot-proto` for the t-distribution.

37 ⟨*T distribution* 34⟩+≡

```
(defun t-distribution ()
  (send dist-plot-proto :new
        :dens #'t-dens
        :cdf #'t-cdf
        :icdf #'t-quant
        :params (list 15)
        :params-print-format "Parameter: Degrees of Freedom = ~,1f"
        :prob-dialog #'t-prob-dialog
        :quant-dialog #'t-quant-dialog
        :num-points 50
        :title "T-Distribution"))
```

Defines:
  `t-distribution`, never used.
Uses `:cdf` 13b, `cdf` 10, `:dens` 13a, `dens` 10, `dist-plot-proto` 10, `:icdf` 13c, `icdf` 10, `:num-points` 13d, `num-points` 10, `:params` 11b, `params` 10, `:params-print-format` 12c, and `params-print-format` 10.

5.3 Chi-square distribution

This is exactly like the t-distribution.

38 ⟨*Chi-square distribution* 38⟩≡

```
(defun chisq-quant-dialog (dist)
  "Dialog for the quantiles of the Chi-square Distribution."
  (let* ((params (send dist :params))
         (prompt (send text-item-proto :new
                   (format nil
                     "To find Chi-square Distribution Quantiles,~%~
                      complete all fields and press OK.~%")))
         (df-label (send text-item-proto :new "Degrees of Freedom"))
         (df-val (send edit-text-item-proto :new
                       (format nil "~a" (select params 0)) :text-length 10))
         (prob-label (send text-item-proto :new "Probability"))
         (prob-val (send edit-text-item-proto :new "" :text-length 10))
         (olist (list df-label df-val prob-label prob-val))
         (mwid (max (mapcar #'(lambda(x) (send x :width)) olist)))
         (ok (send modal-button-proto :new "OK"
                   :action
                   #'(lambda ()
                      (let* ((inputs (get-numbers-from
                                      (list df-val prob-val)))
                             (params (select inputs '(0)))
                             (prob (select inputs 1)))
                        (send dist :params params)
                        (send dist :l-point (send dist :icdf-at prob))
                        (send dist :important-abscissae
                              (list (send dist :l-point)))
                        (send dist :r-point nil)
                        (send dist :answer
                             (quantile-answer prob (send dist :l-point)))
                        (send dist :adjust-to-data)
                        (send dist :redraw)))))
         (cancel (send button-item-proto :new "Cancel"
                       :action
                       #'(lambda()
                           (send (send cancel :dialog)
                                 :modal-dialog-return nil)))))
    (dolist (x olist)
            (send x :width mwid))
    (send (send modal-dialog-proto
                :new (list prompt
                           (list df-label prob-label)
                           (list df-val prob-val)
```

```
                                (list ok cancel))
                        :title "Chi-square Quantile Dialog") :modal-dialog)))
```

Defines:
  chisq-quant-dialog, never used.
Uses :answer 12a, answer 10, get-numbers-from 8c, :icdf 13c, icdf 10, :icdf-at 16d, :important-abscissae 14a, important-abscissae 10, :l-point 14c, l-point 10, :params 11b, params 10, :redraw 27a, :r-point 14d, r-point 10, and :width 9a 9a 9a.
This definition is continued in chunks 40 and 41.
This code is used in chunks 6 and 50b.

Then the dialog for finding Chi-square probabilities.

40 ⟨*Chi-square distribution* 38⟩+≡

```
(defun chisq-prob-dialog (dist)
  "Dialog for the probabilities of the Chi-square Distribution."
  (let* ((params (send dist :params))
         (prompt (send text-item-proto :new
                   (format nil
                     "To find Chisq-Distribution Probabilities,~%~
                      complete appropriate fields and press OK.~%~
                      For probability to the left, use Left Point;~%~
                      for probability to the right, use Right Point;~%~
                      for probability between, use both.~%")))
         (df-label (send text-item-proto :new "Degrees of Freedom"))
         (df-val (send edit-text-item-proto :new
                       (format nil "~a" (select params 0)) :text-length 10))
         (l-label (send text-item-proto :new "Left Point"))
         (l-val (send edit-text-item-proto :new "" :text-length 10))
         (r-label (send text-item-proto :new "Right Point"))
         (r-val (send edit-text-item-proto :new "" :text-length 10))
         (olist (list df-label df-val l-label l-val r-label r-val))
         (mwid (max (mapcar #'(lambda(x) (send x :width)) olist)))
         (ok (send modal-button-proto :new "OK"
                   :action
                   #'(lambda ()
                      (let* ((inputs (get-values-from
                                       (list df-val l-val r-val)))
                             (params (select inputs '(0)))
                             (lx (select inputs 1))
                             (rx (select inputs 2))
                             (between (and rx lx))
                             (left (and lx (not rx)))
                             (prob (if between
                                       (- (send dist :cdf-at rx)
                                          (send dist :cdf-at lx))
                                     (if left
                                        (send dist :cdf-at lx)
                                      (- 1 (send dist :cdf-at rx))))))
                        (send dist :params params)
                        (send dist :l-point lx)
                        (send dist :r-point rx)
                        (send dist :important-abscissae
                              (if between
                                  (list lx rx)
                                (if left
                                    (list lx)
```

```
                                    (list rx))))
                            (send dist :answer
                                 (probability-answer prob lx rx))
                            (send dist :redraw)))))
          (cancel (send button-item-proto :new "Cancel"
                        :action
                        #'(lambda()
                            (send (send cancel :dialog)
                                  :modal-dialog-return nil)))))
     (dolist (x olist)
             (send x :width mwid))
     (send (send modal-dialog-proto
                 :new (list prompt
                            (list df-label df-val)
                            (list l-label l-val)
                            (list r-label r-val)
                            (list ok cancel))
                 :title "Chi-square Probability Dialog") :modal-dialog)))
```

Defines:
  chisq-prob-dialog, never used.
Uses :answer 12a, answer 10, :cdf 13b, cdf 10, :cdf-at 16c, get-values-from 8b, :important-abscissae 14a, important-abscissae 10, :l-point 14c, l-point 10, :params 11b, params 10, :redraw 27a, :r-point 14d, r-point 10, and :width 9a 9a 9a.

And, as usual,

41 ⟨*Chi-square distribution* 38⟩+≡

```
(defun chisq-distribution ()
  (send dist-plot-proto :new
        :dens #'chisq-dens
        :cdf #'chisq-cdf
        :icdf #'chisq-quant
        :params (list 15)
        :params-print-format "Parameter: Degrees of Freedom = ~,1f"
        :prob-dialog #'chisq-prob-dialog
        :quant-dialog #'chisq-quant-dialog
        :num-points 50
        :title "Chi-Square Distribution"))
```

Defines:
  chisq-distribution, never used.
Uses :cdf 13b, cdf 10, :dens 13a, dens 10, dist-plot-proto 10, :icdf 13c, icdf 10, :num-points 13d, num-points 10, :params 11b, params 10, :params-print-format 12c, and params-print-format 10.

5.4 The F Distribution

We wont bother describing the code for the F-distribution.

42 ⟨*F distribution* 42⟩≡

```
(defun f-quant-dialog (dist)
  "Dialog for the quantiles of the F Distribution."
  (let* ((params (send dist :params))
         (prompt (send text-item-proto :new
                   (format nil
                     "To find F Distribution Quantiles,~%~
                      complete all fields and press OK.~%")))
         (ndf-label (send text-item-proto :new "Numerator df"))
         (ndf-val (send edit-text-item-proto :new
                        (format nil "~a" (select params 0)) :text-length 10))
         (ddf-label (send text-item-proto :new "Denominator df"))
         (ddf-val (send edit-text-item-proto :new
                      (format nil "~a" (select params 1)) :text-length 10))
         (prob-label (send text-item-proto :new "Probability"))
         (prob-val (send edit-text-item-proto :new "" :text-length 10))
         (olist (list ndf-label ndf-val ddf-label ddf-val
                      prob-label prob-val))
         (mwid (max (mapcar #'(lambda(x) (send x :width)) olist)))
         (ok (send modal-button-proto :new "OK"
                   :action
                   #'(lambda ()
                      (let* ((inputs (get-numbers-from
                                       (list ndf-val ddf-val prob-val)))
                             (params (select inputs '(0 1)))
                             (prob (select inputs 2)))
                        (send dist :params params)
                        (send dist :l-point (send dist :icdf-at prob))
                        (send dist :important-abscissae
                              (list (send dist :l-point)))
                        (send dist :r-point nil)
                        (send dist :answer
                             (quantile-answer prob (send dist :l-point)))
                        (send dist :adjust-to-data)
                        (send dist :redraw)))))
         (cancel (send button-item-proto :new "Cancel"
                       :action
                       #'(lambda()
                           (send (send cancel :dialog)
                                 :modal-dialog-return nil)))))
    (dolist (x olist)
            (send x :width mwid))
```

```
(send (send modal-dialog-proto
            :new (list prompt
                       (list ndf-label ddf-label)
                       (list ndf-val ddf-val)
                       (list prob-label prob-val)
                       (list ok cancel))
            :title "F Quantile Dialog") :modal-dialog)))
```

Defines:
  f-quant-dialog, never used.
Uses :answer 12a, answer 10, get-numbers-from 8c, :icdf 13c, icdf 10, :icdf-at 16d, :important-abscissae 14a, important-abscissae 10, :l-point 14c, l-point 10, :params 11b, params 10, :redraw 27a, :r-point 14d, r-point 10, and :width 9a 9a 9a.
This definition is continued in chunks 44 and 45.
This code is used in chunks 6 and 50b.

44 ⟨*F distribution* 42⟩+≡

```
(defun f-prob-dialog (dist)
  "Dialog for the probabilities of the F Distribution."
  (let* ((params (send dist :params))
         (prompt (send text-item-proto :new
                   (format nil
                     "To find F istribution Probabilities,~%~
                      complete appropriate fields and press OK.~%~
                      For probability to the left, use Left Point;~%~
                      for probability to the right, use Right Point;~%~
                      for probability between, use both.~%")))
         (ndf-label (send text-item-proto :new "Numerator df"))
         (ndf-val (send edit-text-item-proto :new
                        (format nil "~a" (select params 0)) :text-length 10))
         (ddf-label (send text-item-proto :new "Denominator df"))
         (ddf-val (send edit-text-item-proto :new
                       (format nil "~a" (select params 1)) :text-length 10))
         (l-label (send text-item-proto :new "Left Point"))
         (l-val (send edit-text-item-proto :new "" :text-length 10))
         (r-label (send text-item-proto :new "Right Point"))
         (r-val (send edit-text-item-proto :new "" :text-length 10))
         (olist (list mean-label mean-val sd-label sd-val
                      l-label l-val r-label r-val))
         (mwid (max (mapcar #'(lambda(x) (send x :width)) olist)))
         (ok (send modal-button-proto :new "OK"
                   :action
                   #'(lambda ()
                      (let* ((inputs (get-values-from
                                       (list ndf-val ddf-val l-val r-val)))
                             (params (select inputs '(0 1)))
                             (lx (select inputs 2))
                             (rx (select inputs 3))
                             (between (and rx lx))
                             (left (and lx (not rx)))
                             (prob (if between
                                       (- (send dist :cdf-at rx)
                                          (send dist :cdf-at lx))
                                     (if left
                                        (send dist :cdf-at lx)
                                       (- 1 (send dist :cdf-at rx))))))
                        (send dist :params params)
                        (send dist :l-point lx)
                        (send dist :r-point rx)
                        (send dist :important-abscissae
                              (if between
```

```
                                    (list lx rx)
                                  (if left
                                      (list lx)
                                    (list rx))))
                            (send dist :answer
                                 (probability-answer prob lx rx))
                            (send dist :redraw)))))
          (cancel (send button-item-proto :new "Cancel"
                        :action
                        #'(lambda()
                            (send (send cancel :dialog)
                                  :modal-dialog-return nil)))))
      (dolist (x olist)
              (send x :width mwid))
      (send (send modal-dialog-proto
                  :new (list prompt
                             (list ndf-label ddf-label)
                             (list ndf-val ddf-val)
                             (list l-label l-val)
                             (list r-label r-val)
                             (list ok cancel))
                  :title "F Probability Dialog") :modal-dialog)))
```

Defines:
  f-prob-dialog, never used.
Uses :answer 12a, answer 10, :cdf 13b, cdf 10, :cdf-at 16c, get-values-from 8b, :important-abscissae 14a, important-abscissae 10, :l-point 14c, l-point 10, :params 11b, params 10, :redraw 27a, :r-point 14d, r-point 10, and :width 9a 9a 9a.

45 ⟨*F distribution* 42⟩+≡

```
(defun f-distribution ()
  (send dist-plot-proto :new
        :dens #'f-dens
        :cdf #'f-cdf
        :icdf #'f-quant
        :params (list 20 20)
        :params-print-format "Parameters: Num. df. = ~,1f, Den. df. = ~,1f"
        :prob-dialog #'f-prob-dialog
        :quant-dialog #'f-quant-dialog
        :num-points 50
        :title "FDistribution"))
```

Defines:
  f-distribution, never used.
Uses :cdf 13b, cdf 10, :dens 13a, dens 10, dist-plot-proto 10, :icdf 13c, icdf 10, :num-points 13d, num-points 10, :params 11b, params 10, :params-print-format 12c, and params-print-format 10.

6 The Main Function

The main function `stbl` installs the statistical tables in a menu.

46a ⟨*Main function* 46a⟩≡
```
(defun stbl ()
"Method args: none
 Installs statistical tables in a menu."
  (let ((menu (send menu-proto :new "Tables")))
    (send menu :append-items
          (send menu-item-proto :new "Normal Distribution"
                :action #'normal-distribution)
          (send menu-item-proto :new "T Distribution"
                :action #'t-distribution)
          (send menu-item-proto :new "Chi-square Distribution"
                :action #'chisq-distribution)
          (send menu-item-proto :new "F Distribution"
                :action #'F-distribution))
    (send menu :install)))
```
Defines:
  `stbl`, used in chunks 3, 46b, and 51.
This code is used in chunks 6 and 50c.

Oh!, we have to invoke the function!

46b ⟨*Invoke main function* 46b⟩≡
```
(stbl)
```
Uses `stbl` 46a.
This code is used in chunks 6 and 50c.

7 Other Auxiliary Programs

46c ⟨*Program hist.lsp* 46c⟩≡
```
⟨Copyright for code 5⟩
(def z (histogram (normal-rand 1000)))
(defmeth z :close () (send self :remove) (exit))
```
Defines:
  `hist.lsp`, used in chunks 3 and 51a.
This code is used in chunk 6.

47a ⟨*Program smhist.lsp* 47a⟩≡

```
⟨Copyright for code 5⟩
(def l (normal-rand 1000))
(def z (histogram l))
(send z :add-lines (kernel-dens l :type 'g))
(defmeth z :close () (send self :remove) (exit))
```

Defines:
smhist.lsp, used in chunks 3 and 51a.
Uses dens 10.
This code is used in chunk 6.

47b ⟨*Program 68.lsp* 47b⟩≡

```
⟨Copyright for code 5⟩
(require "dists")
(def z
  (send dist-plot-proto :new
        :dens #'gaussian-density
        :cdf #'gaussian-cdf
        :icdf #'gaussian-icdf
        :params (list 0 1)
        :params-print-format "Parameters: Mean = ~d, Std. Dev. = ~d"
        :prob-dialog #'normal-prob-dialog
        :quant-dialog #'normal-quant-dialog
        :num-points 50
        :title "Normal Distribution"))
(defmeth z :close () (send self :remove) (exit))
(send z :l-point -1)
(send z :r-point 1)
(send z :important-abscissae '(-1 1))
(send z :answer "The shaded area is 68%")
(send z :redraw)
```

Defines:
68.lsp, used in chunks 3 and 51a.
Uses :answer 12a, answer 10, :cdf 13b, cdf 10, :dens 13a, dens 10, dist-plot-proto 10, :icdf 13c, icdf 10, :important-abscissae 14a, important-abscissae 10, :l-point 14c, l-point 10, :num-points 13d, num-points 10, :params 11b, params 10, :params-print-format 12c, params-print-format 10, :redraw 27a, :r-point 14d, and r-point 10.
Root chunk (not used in this document).

48 ⟨*Program 95.lsp* 48⟩≡

```
⟨Copyright for code 5⟩
(require "dists")
(def z
  (send dist-plot-proto :new
        :dens #'gaussian-density
        :cdf #'gaussian-cdf
        :icdf #'gaussian-icdf
        :params (list 0 1)
        :params-print-format "Parameters: Mean = ~d, Std. Dev. = ~d"
        :prob-dialog #'normal-prob-dialog
        :quant-dialog #'normal-quant-dialog
        :num-points 50
        :title "Normal Distribution"))
(defmeth z :close () (send self :remove) (exit))
(send z :l-point -2)
(send z :r-point 2)
(send z :important-abscissae '(-2 2))
(send z :answer "The shaded area is 95%")
(send z :redraw)
```

Defines:
95.lsp, used in chunks 3 and 51a.
Uses :answer 12a, answer 10, :cdf 13b, cdf 10, :dens 13a, dens 10, dist-plot-proto 10, :icdf 13c, icdf 10, :important-abscissae 14a, important-abscissae 10, :l-point 14c, l-point 10, :num-points 13d, num-points 10, :params 11b, params 10, :params-print-format 12c, params-print-format 10, :redraw 27a, :r-point 14d, and r-point 10.
Root chunk (not used in this document).

49a ⟨*Program 99.lsp* 49a⟩≡
```
⟨Copyright for code 5⟩
(require "dists")
(def z
  (send dist-plot-proto :new
        :dens #'gaussian-density
        :cdf #'gaussian-cdf
        :icdf #'gaussian-icdf
        :params (list 0 1)
        :params-print-format "Parameters: Mean = ~d, Std. Dev. = ~d"
        :prob-dialog #'normal-prob-dialog
        :quant-dialog #'normal-quant-dialog
        :num-points 50
        :title "Normal Distribution"))
(defmeth z :close () (send self :remove) (exit))
(send z :l-point -3)
(send z :r-point 3)
(send z :important-abscissae '(-3 3))
(send z :answer "The shaded area is 99.7%")
(send z :redraw)
```
Defines:
`99.lsp`, used in chunks 3 and 51a.
Uses `:answer` 12a, `answer` 10, `:cdf` 13b, `cdf` 10, `:dens` 13a, `dens` 10, `dist-plot-proto` 10, `:icdf` 13c, `icdf` 10, `:important-abscissae` 14a, `important-abscissae` 10, `:l-point` 14c, `l-point` 10, `:num-points` 13d, `num-points` 10, `:params` 11b, `params` 10, `:params-print-format` 12c, `params-print-format` 10, `:redraw` 27a, `:r-point` 14d, and `r-point` 10.
Root chunk (not used in this document).

8 Miscellany

It is often the case that one might want some of the functions we have designed here for other purposes. So let us make it easy to extract various sections of the code as separate files. I have not bothered with packages, but that would be easy to fix.

49b ⟨*Utilities file* 49b⟩≡
```
⟨Copyright for code 5⟩
(provide "utils")
⟨Utility functions 7a⟩
⟨Additional methods for built-in prototypes 9a⟩
```
Root chunk (not used in this document).

49c ⟨*Button overlay file* 49c⟩≡
```
⟨Copyright for code 5⟩
(provide "button")
⟨Button overlay prototype definition 26a⟩
⟨Button overlay prototype methods 26b⟩
```
Root chunk (not used in this document).

50a ⟨*Distribution proto file* 50a⟩≡

```
⟨Copyright for code 5⟩
(require "utils")
(require "button")
(provide "distproto")
⟨Implementation constants 17a⟩
⟨Distribution prototype definition 10⟩
⟨Distribution prototype methods 11a⟩
```

Root chunk (not used in this document).

50b ⟨*Distributions file* 50b⟩≡

```
⟨Copyright for code 5⟩
(require "distproto")
(provide "dists")
⟨Normal distribution 29a⟩
⟨T distribution 34⟩
⟨Chi-square distribution 38⟩
⟨F distribution 42⟩
```

Root chunk (not used in this document).

50c ⟨*Statistical tables file* 50c⟩≡

```
⟨Copyright for code 5⟩
(require "dists")
(provide "stbls")
⟨Main function 46a⟩
⟨Invoke main function 46b⟩
```

Root chunk (not used in this document).

9 The Makefile

Our `Makefile` must be able to extract each of the individual files, the hyper-document and LaTeX the hyper-document. It is pretty simple.

51a ⟨*Makefile* 51a⟩≡

```
all:    stbl.nw
        noweave -index -delay stbl.nw > stbl.tex
        notangle -R'Hyper-document' stbl.nw > hyperdoc.tex
        notangle -R'Utilities file' stbl.nw > utils.lsp
        notangle -R'Button overlay file' stbl.nw > button.lsp
        notangle -R'Distribution proto file' stbl.nw > distproto.lsp
        notangle -R'Distributions file' stbl.nw > dists.lsp
        notangle -R'Statistical tables file' stbl.nw > stbls.lsp
        notangle -R'Program hist.lsp' stbl.nw > hist.lsp
        notangle -R'Program smhist.lsp' stbl.nw > smhist.lsp
        notangle -R'Program 68.lsp' stbl.nw > 68.lsp
        notangle -R'Program 95.lsp' stbl.nw > 95.lsp
        notangle -R'Program 99.lsp' stbl.nw > 99.lsp
        notangle -R'Readme file' stbl.nw > README
```

Uses `hist.lsp` 46c, `68.lsp` 47b, `95.lsp` 48, `99.lsp` 49a, `smhist.lsp` 47a, and `stbl` 46a.
This code is used in chunk 2a.

10 The Readme file

We shall just refer them to the literate programming introduction.

51b ⟨*Readme file* 51b⟩≡

```
Please look at the introduction section of stbl.ps and read it in its
entirety.

It contains a description of the program and how to use it. In
addition there are some pictures that will give you a better idea of
what the code can do.
```

Uses `stbl` 46a.
This code is used in chunk 2a.

11 Discussion

We devote this section to a discussion of problems and issues arising in the development of a full-fledged hyper-text. It is incomplete as of now.

References

[1] Moore, David S., and McCabe, George P., *Introduction to the Practice of Statistics*, Second Edition, edition, W. H. Freeman & Co., (1993).

List of code chunks

This list is generated automatically. The numeral is that of the first definition of the chunk.

⟨*T distribution* 34⟩
⟨*Utilities file* 49b⟩
⟨*Utility functions* 7a⟩

Index

Here is a list of the identifiers used, and where they appear. Underlined entries indicate the place of definition. This index is generated automatically.