PLT Scheme Science Collection

Reference Manual

Edition 2.0 for Version 2.0

M. Douglas Williams

December 2005

Contents

1 Introduction **1**
- 1.1 Routines Available in the Science Collection 2
- 1.2 The Science Collection is Free Software 2
- 1.3 Obtaining the Science Collection 2
- 1.4 No Warranty 3

2 Using the Science Collection **4**
- 2.1 An Example 4
- 2.2 Loading Modules in the Science Collection 5
- 2.3 Graphics Modules 6

3 Error Handling **7**
- 3.1 Contracts 7
- 3.2 Infinities and Not-a-Number 9
- 3.3 Exceptions 9

4 Mathematical Functions **10**
- 4.1 Mathematical Constants 10
- 4.2 Infinities and Not-a-Number 10
- 4.3 Elementary Functions 11
- 4.4 Testing the Sign of Numbers 13
- 4.5 Approximate Comparisons of Real Numbers 13

5 Special Functions **14**
- 5.1 Error Functions 14
  - 5.1.1 Error Function 14
  - 5.1.2 Complementary Error Function 15
  - 5.1.3 Hazard Function 16
- 5.2 Gamma Functions 17
  - 5.2.1 Gamma Function 18
  - 5.2.2 Regulated Gamma Function 20
  - 5.2.3 Factorial Function 21
  - 5.2.4 Double Factorial Function 21
  - 5.2.5 Binomial Coefficient Function 21

5.3 Psi Functions 22
5.3.1 Psi (Digamma) Functions 22
5.3.2 Psi-1 (Trigamma) Functions 23
5.3.3 Psi-n (Polygamma) Function 24
5.4 Zeta Functions 25
5.4.1 Riemann Zeta Functions 26
5.4.2 Riemann Zeta Functions Minus One 26
5.4.3 Hurwitz Zeta Function 27
5.4.4 Eta Functions 28

6 Random Number Generation 30
6.1 The SRFI 27 Specification 30
6.2 Additional Random Number Functionality 32
6.2.1 The `current-random-source` Parameter 32
6.2.2 Uniform Random Numbers 33
6.2.3 Miscellaneous Functions 34
6.2.4 Random Source Vectors 34
6.3 Examples 34

7 Random Number Distributions 37
7.1 The Beta Distribution 37
7.1.1 Random Variates from the Beta Distribution 38
7.1.2 Beta Distribution Density Functions 38
7.1.3 Beta Distribution Graphics 38
7.2 The Bivariate Gaussian Distribution 39
7.2.1 Random Variates from the Bivariate Gaussian Distribution 40
7.2.2 Bivariate Gaussian Distribution Density Functions 41
7.2.3 Bivariate Gaussian Distribution Graphics 42
7.3 The Chi-Squared Distribution 42
7.3.1 Random Variates from the Chi-Squared Distribution . . . 42
7.3.2 Chi-Squared Distribution Density Functions 44
7.3.3 Chi-Squared Distribution Graphics 44
7.4 The Exponential Distribution 45
7.4.1 Random Variates from the Exponential Distribution . . . 46
7.4.2 Exponential Distribution Density Functions 46
7.4.3 Exponential Distribution Graphics 46
7.5 The F-Distribution 48
7.5.1 Random Variates from the F-Distribution 48
7.5.2 F-Distribution Density Functions 49
7.5.3 F-Distribution Graphics 50
7.6 The Flat (Uniform) Distribution 50
7.6.1 Random Variates from the Flat Distribution 50
7.6.2 Flat Distribution Density Functions 52
7.6.3 Flat Distribution Graphics 53
7.7 The Gamma Distribution 53
7.7.1 Random Variates from the Gamma Distribution 53

7.7.2 Gamma Distribution Density Functions 54
7.7.3 Gamma Distribution Graphics 55
7.8 The Gaussian Distribution 56
7.8.1 Random Variates from the Gaussian Distribution 56
7.8.2 Gaussian Distribution Density Functions 59
7.8.3 Gaussian Distribution Graphics 60
7.9 The Gaussian Tail Distribution 61
7.9.1 Random Variates from the Gaussian Tail Distribution . . 61
7.9.2 Gaussian Tail Distribution Density Functions 63
7.9.3 Gaussian Tail Distribution Graphics 64
7.10 The Log Normal Distribution 66
7.10.1 Random Variates from the Log Normal Distribution . . . 67
7.10.2 Log Normal Distribution Density Functions 67
7.10.3 Log Normal Distribution Graphics 67
7.11 The Pareto Distribution 69
7.11.1 Random Variates from the Pareto Distribution 69
7.11.2 Pareto Distribution Density Functions 70
7.11.3 Pareto Distribution Graphics 71
7.12 The T-Distribution 71
7.12.1 Random Variates from the T-Distribution 71
7.12.2 T-Distribution Density Functions 72
7.12.3 T-Distribution Graphics 73
7.13 The Triangular Distribution 74
7.13.1 Random Variates from the Triangular Distribution 75
7.13.2 Triangular Distribution Density Functions 75
7.13.3 Triangular Distribution Graphics 76
7.14 The Bernoulli Distribution 78
7.14.1 Random Variates from the Bernoulli Distribution 78
7.14.2 Bernoulli Distribution Density Functions 78
7.14.3 Bernoulli Distribution Graphics 79
7.15 The Binomial Distribution 80
7.15.1 Random Variates from the Binomial Distribution 81
7.15.2 Binomial Distribution Density Functions 81
7.15.3 Binomial Distribution Graphics 81
7.16 The Geometric Distribution 82
7.16.1 Random Variates from the Geometric Distribution 83
7.16.2 Geometric Distribution Density Functions 84
7.16.3 Geometric Distribution Graphics 84
7.17 The Logarithmic Distribution 85
7.17.1 Random Variates from the Logarithmic Distribution . . . 86
7.17.2 Logarithmic Distribution Density Functions 86
7.17.3 Logarithmic Distribution Graphics 86
7.18 The Poisson Distribution 87
7.18.1 Random Variates from the Poisson Distribution 88
7.18.2 Poisson Distribution Density Functions 89
7.18.3 Poisson Distribution Graphics 89

7.19 The General Discrete Distribution 91
7.19.1 Creating Discrete Distributions 91
7.19.2 Random Variates from a Discrete Distribution 91
7.19.3 Discrete Distribution Density Functions 92
7.19.4 General Discrete Distribution Graphics 92

8 Statistics 94
8.1 Mean, Standard Deviation, and Variance 94
8.2 Absolute Deviation 95
8.3 Higher Moments (Skewness and Kurtosis) 95
8.4 Autocorrelation 96
8.5 Covariance 97
8.6 Weighted Samples 98
8.7 Maximum and Minimum 101
8.8 Median and Percentiles 102
8.9 Example 103

9 Histograms 107
9.1 1D Histograms 107
9.1.1 Creating 1D Histograms 107
9.1.2 Updating and Accessing 1D Histogram Elements 108
9.1.3 1D Histogram Statistics 109
9.1.4 1D Histogram Graphics 110
9.1.5 Examples 111
9.2 2D Histograms 112
9.2.1 Creating 2D Histograms 113
9.2.2 Updating and Accessing 2D Histogram Elements 113
9.2.3 2D Histogram Statistics 115
9.2.4 2D Histogram Graphics 116
9.2.5 Example 117
9.3 Discrete Histograms 117
9.3.1 Creating Discrete Histograms 119
9.3.2 Updating and Accessing Discrete Histogram Elements . . 119
9.3.3 Discrete Histogram Statistics 120
9.3.4 Discrete Histogram Graphics 121
9.3.5 Examples 121

10 Ordinary Differential Equations 124
10.1 Defining the ODE System 124
10.2 Stepping Functions 125
10.3 Adaptive Step-Size Control 126
10.4 Evolution 128
10.5 Examples 129

11 Chebyshev Approximations **135**
11.1 The `chebyshev-series` Structure 135
11.2 Creation and Calculation of Chebyshev Series 136
11.3 Chebyshev Series Evaluation 136
11.4 Examples 136

A GNU Lesser General Public License (LGPL) **140**

B SRFI 27: Sources of Random Bits **151**

Bibliography **162**

Index **163**

List of Figures

2.1 Histogram of Sum of Two Dice 5

5.1 Plot of Error Function on $[4, 4]$ 15
5.2 Plot of Complementary Error Function on $[4, 4]$ 16
5.3 Plot of Hazard Function on $[-5, 10]$ 17
5.4 Plot of Gamma Function on $(0, 6]$ 19
5.5 Plot of Gamma Function on $(-1, 0)$ 20
5.6 Plot of Psi (Digamma) Function on $(0, 5]$ 23
5.7 Plot of Psi-1 (Trigamma) Function on $(0, 5]$ 24
5.8 Plot of Psi-n (Polygamma), $n = 3$, Function on $(0, 5]$ 25
5.9 Plot of Reimann Zeta Function on $[-5, 5]$ 27
5.10 Plot of Hurwitz Zeta Function, $q = 2.0$, on $(1, 5]$ 28
5.11 Plot of Eta Function on $[-10, 10]$ 29

6.1 Histogram of Uniform Random Numbers 35
6.2 Histogram of Uniform Random Integers 36

7.1 Histogram of Random Variates from Beta (2.0, 3.0) 39
7.2 Plot of Probability Density for Beta(2.0, 3.0) 40
7.3 Histogram of Random Variates from Bivariate Gaussian (1.0, 1.0, 0.0) 41
7.4 Plot of Probability Density for Bivariate Gaussian (1.0, 1.0, 0.0) 43
7.5 Histogram of Random Variates from Chi-Squared (3.0)) 44
7.6 Plot of Probability Density for Chi-Squared (3.0) 45
7.7 Histogram of Random Variates from Exponential (1.0)) 47
7.8 Plot of Probability Density for Exponential (1.0) 48
7.9 Histogram of Random Variates from F-Distribution (2.0, 3.0)) . . 49
7.10 Plot of Probability Density for F-Distribution (2.0, 3.0) 51
7.11 Histogram of Random Variates from Flat (Uniform) (1.0, 4.0) . . 52
7.12 Plot of Probability Density for Flat (Uniform) (1.0, 4.0) 54
7.13 Histogram of Random Variates from Gamma (3.0, 3.0) 55
7.14 Plot of Probability Density for Gamma (3.0, 3.0) 56
7.15 Histogram of Random Variates from Gaussian (Normal) (10.0, 2.0) 57
7.16 Histogram of Random Variates from Unit Gaussian (Normal) . . 58

7.17 Plot of Probability Density for Gaussian (Normal) (10.0, 2.0) . . 61
7.18 Plot of Probability Density for Unit Gaussian (Normal) 62
7.19 Histogram of Random Variates from Gaussian Tail (16.0, 10.0, 2.0) 63
7.20 Histogram of Random Variates from Unit Gaussian Tail (3.0) . . 64
7.21 Plot of Probability Density for Gaussian Tail (16.0, 10.0, 2.0) . . 65
7.22 Plot of Probability Density for Unit Gaussian Tail (3.0) 66
7.23 Histogram of Random Variates from Log Normal (0.0, 1.0) . . . 68
7.24 Plot of Probability Density for Log Normal (0.0, 1.0) 69
7.25 Histogram of Random Variates from Pareto (1.0, 1.0) 70
7.26 Plot of Probability Density for Pareto (1.0, 1.0) 72
7.27 Histogram of Random Variates from T-Distribution (1.0) 73
7.28 Plot of Probability Density for T-Distribution (1.0) 74
7.29 Histogram of Random Variates from Triangular (1.0, 4.0, 2.0) . . 76
7.30 Plot of Probability Density for Triangular (1.0, 4.0, 2.0) 77
7.31 Histogram of Random Variates from Bernoulli (0.6) 79
7.32 Plot of Probability Density for Bernoulli (0.6) 80
7.33 Histogram of Random Variates from Binomial (0.5, 20) 82
7.34 Plot of Probability Density for Binomial (0.5 20) 83
7.35 Histogram of Random Variates from Geometric (0.5) 84
7.36 Plot of Probability Density for Geometric (0.5) 85
7.37 Histogram of Random Variates from Logarithmic (0.5) 87
7.38 Plot of Probability Density for Logarithmic (0.5) 88
7.39 Histogram of Random Variates from Poisson (10.0) 89
7.40 Plot of Probability Density for Poisson (10.0) 90
7.41 Histogram of Random Variates from a Discrete Distribution . . . 92
7.42 Plot of Probability Density for a General Discrete Distribution . 93

9.1 Histogram of Random Variates from Unit Gaussian (Normal) . . 111
9.2 Scaled Histogram of Random Variates from Exponential (1.0)) . 112
9.3 Histogram of Random Variates from Bivariate Gaussian (1.0, 1.0, 0.0) 118
9.4 Histogram of Random Variates from Poisson (10.0) 122
9.5 Histogram of Random Variates from Logarithmic (0.5) 123

10.1 ODE Example 1 Plot of y0 131
10.2 ODE Example 1 Plot of y1 132
10.3 ODE Example 2 Plot of y0 133
10.4 ODE Example 2 Plot of y1 134

11.1 Chebyshev Series Order 10 138
11.2 Chebyshev Series Order 40 139

List of Tables

4.1 Mathematical Constants 11

Chapter 1

Introduction

The PLT Scheme Science Collection is a collection of modules that provide functions for numerical computing. The structure of the science collection and many of the underlying algorithms were inspired by the GNU Scientific Library (GSL)[2]. The functions are written entirely in PLT Scheme[3] and present a true Scheme look-and-feel throughout. The source code is distributed with the science collection and licensed under the GNU Lesser General Public License (LGPL)[1].

The motivation behind the PLT Scheme Science Collection is to provide a numerical framework for knowledge-based simulation in PLT Scheme. Indeed, many of the routines were originally developed as part of the PLT Scheme Simulation Collection[8]. It was noted that much of the functionality (e.g., random number generation, random distributions, histograms and statistics) did not depend on, or could be separated from, the underlying simulation engine. When this was done, it was further noted that this functionality represented a subset of the functionality available in the GSL. At that point, it was decided to use the structure and, to the extent practical, the algorithms of the GSL as a reference model. Thus, the PLT Scheme Science Collection was born.

This reference manual is based on the GNU Scientific Library Reference Manual[2]. Because of the differences between C and PLT Scheme, and the fundamental differences between the underlying numeric models of each, the presentation of the functions is different here than in the GSL Reference Manual. We also rely more on the graphical representation of results, using the plot collection distributed with PLT Scheme (PLoT Scheme)[7].

We are indebted to Dr. M. Galassi and Dr. J. Theiler of Los Alamos National Laboratory and the others who have contributed to the development of the GNU Scientific Library (GSL). Any weaknesses in the PLT Scheme Science Collection are our own and must not be construed as having origins in the GSL.

1.1 Routines Available in the Science Collection

The PLT Scheme Science Collection covers a range of topics in numerical computing. Functions are available for the following areas:[1]

- Mathematical Constants and Functions
- Special Functions
- Random Numbers
- Random Distributions
- Statistics
- Histograms
- Ordinary Differential Equations
- Chebyshev Approximations

The use of these functions is described in this manual. Each chapter provides detailed definitions of the functions, with example code.

1.2 The Science Collection is Free Software

The PLT Scheme Science Collection is free software – this means that anyone is free to use it and to redistribute it in other free programs. The science collection is not in the public domain – it is copyrighted and there are conditions on its distribution. Specifically, the PLT Scheme Science Collection is distributed under the GNU Lesser General Public License (LGPL)[1]. A copy of the LGPL is provided as Appendix A of this document.

1.3 Obtaining the Science Collection

The preferred method for obtaining the PLT Scheme Science Collection is via the PLaneT Package Repository (PLaneT), PLT Scheme's centralized package distribution system[6]. The PLaneT identifier for the PLT Scheme Science Collection, Version 2.0 (or later) is `("williams" "science.plt" 2 0)`. PLT Scheme will automatically download and install the science collection from the PLaneT server. See Chapter 2 for an example.

Note that Version 2.0 of the PLT Scheme Science Collection requires PLT Scheme Version 300 or higher.

[1]This is only a fraction of the areas supported by the GSL. For a complete reference of the numerical computing areas supported by the GSL, please refer to the GSL Reference Manual[2].

1.4 No Warranty

The PLT Scheme Science Collection 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. It is your responsibility to validate the behavior of the software and their accuracy using the source code provided. See the GNU Lesser General Public License (LGPL)[1] for more details.

Chapter 2

Using the Science Collection

This chapter describes how to use the PLT Scheme Science Collection and introduces its conventions.

2.1 An Example

The following code demonstrates the use of the PLT Scheme Science Collection by plotting a histogram of 10,000 trials of rolling two dice.

```
(require (planet "random-source.ss"
                 ("williams" "science.plt" 2 0)))
(require (planet "discrete-histogram-with-graphics.ss"
                 ("williams" "science.plt" 2 0)))

(let ((s (make-random-source))
      (h (make-discrete-histogram)))
  (random-source-randomize! s)
  (do ((i 0 (+ i 1)))
      ((= i 10000) (void))
    (let ((die1 (+ (random-uniform-int 6) 1))
          (die2 (+ (random-uniform-int 6) 1)))
      (discrete-histogram-increment! h (+ die1 die2))))
  (discrete-histogram-plot h "Histogram of Sum of Two Dice"))
```

Figure 2.1 shows an example of the resulting histogram.

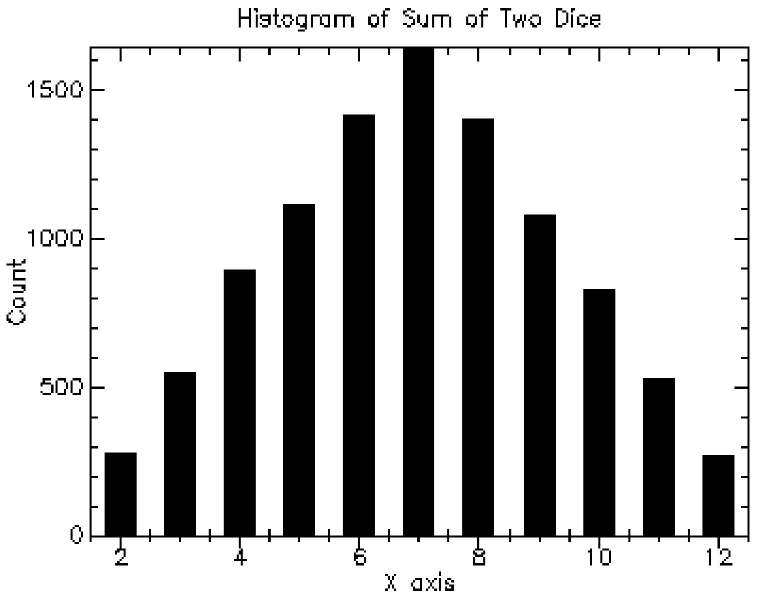

Figure 2.1: Histogram of Sum of Two Dice

2.2 Loading Modules in the Science Collection

The PLT Scheme Science Collection is a collection of modules each of which provides a specific area of functionality in numerical computing. Typical user code will only load the modules it requires using the `require` special form.

For example, the code in Section 2.1 requires two of the modules from the science collection: `random-source` and `discrete-histogram-with-graphics`. This is specified using the following:

```
(require (planet "random-source.ss"
                 ("williams" "science.plt" 2 0)))
(require (planet "discrete-histogram-with-graphics.ss"
                 ("williams" "science.plt" 2 0)))
```

Each of these statements will load the corresponding module from the science collection – assumming they are not already loaded – and make it available for use. PLT Scheme will automatically download and install the science collection from the PLaneT server as needed.

There are two sub-collections of the science collection. These are:

- special-functions
- random-distributions

Loading modules from one of these sub-collections requires that the sub-collection also be specified when using the `require` special form. For example, to load the module for the Gaussian random distribution, the following is used:

```
(require (planet "gaussian.ss" ("williams" "science.plt" 2 0)
                 "random-distributions"))
```

As a shortcut, the entire science collection can be loaded using one of the following, depending on whether or not the graphics routines are needed:

```
(require (planet "science.ss" ("williams" "science.plt" 2 0)))
(require (planet "science-with-graphics.ss"
                 ("williams" "science.plt" 2 0)))
```

2.3 Graphics Modules

Support for the graphical functions within the modules of the science collection has been separated from the fundamental numerical computing functionality of the modules. This facilitates the use of the numerical computing functions in non-graphical environments or when alternative graphical presentations are desired.

By convention, when graphical functions are included for a specific numerical computing area, there are three modules that provide the functions:

- *module* – the basic numerical computing functions
- *module*`-graphics` – the graphical functions
- *module*`-with-graphics` – both the basic numerical computing and graphical functions

In general, either the *module* or *module*`-with-graphics` module is loaded. However, the *module*`-graphics` module can be loaded when only the graphical routines are being referenced.[1]

For example, the code in Section 2.1 requires both the basic numerical computing and graphical functions for the discrete histogram functionality. Therefore, it loads the `discrete-histogram-with-graphics` module using the form:

```
(require (planet "discrete-histogram-with-graphics.ss"
                 ("williams" "science.plt" 2 0)))
```

The graphical routines are implemented using the plot collection provided with PLT Scheme (PLoT Scheme)[7]. The plot collection is, in turn, implemented using MrEd[5]. Both of these modules are required to be present to use the graphical functions.[2]

[1]This might be used in implementing higher-level graphical interfaces.
[2]This is normally the case for PLT Scheme Version 2.07 and higher.

Chapter 3

Error Handling

This chapter describes how error handling is performed in the PLT Scheme Science Collection and its error handling conventions.

3.1 Contracts

In PLT Scheme, a *contract* defines and enforces the interface to (and from) a function. Contracts are defined in Chapter 13: `contract.ss`: Contracts in the PLT MzLib: Libraries Manual[4].

The PLT Scheme Science Collection uses contracts to define and enforce the interfaces for all of the functions provided by the modules in the collection. This ensures that all calls to these functions are checked for the proper number and type of arguments (and results) as well as range checking and inter-argument constraints where practical.

All of the function descriptions in this manual include a specification of the contract for the function.

The following examples show some of the different contract specifications for functions and how to interpret them. All of these examples are from the `statistics.ss` file.

Fixed Number of Arguments with a Single Result

Example: `mean` from the `statistics.ss` file.

Function: `(mean data)`
Contract: `(-> (vectorof real?) real?)`

The contract specifies a function with one argument, which must match the contract `(vectorof real?)`, and returns a single value, which must match the contract `real?`.

```
> (require (planet "statistics.ss" ("williams" "science.plt" 2 0)))
> (mean #(1 2 3 4))
```

```
2.5
> (mean #(1 2 3 'a))
6:3: top-level broke the contract (-> (vectorof real?) real?) it had
with (planet "statistics.ss" ("williams" "science.plt" 2 0)) on mean;
expected <(vectorof real?)>, given: #4(1 2 3 'a)
```

Multiple Lambda Forms (`case-lambda`)

Example: `variance` from the `statistics.ss` file.

```
Function:   (variance data mu)
Function:   (variance data)
Contract:   (case->
               (-> (vectorof real?) real? (>=/c 0.0))
               (-> (vectorof real?) (>=/c 0.0)))
```

The contract specifies multiple lambda forms using `case->`. The first case specifies a function with two arguments, which must match the contracts `(vectorof real?)` and `real?`, and returns a single value, which must match the contract `(>=/c 0.0)`. The second case specifies a function with a single arguemnt, which must match the contract `(vectorof real?)`, and returns a single value, which must match the contract `(>=/c 0.0)`.

```
> (require (planet "statistics.ss" ("williams" "science.plt" 2 0)))
> (variance #(1 2 3 4))
1.6666666666666665
> (variance #(1 2 3 4) 2.5)
1.6666666666666665
> (variance #(1 2 3 4) 'a)
8:3: top-level broke the contract
  (case->
    (-> (vectorof real?) real? (>=/c 0.0))
    (-> (vectorof real?) (>=/c 0.0)))
it had with (planet "statistics.ss" ("williams" "science.plt" 2 0))
on variance; expected <real?>, given: a
```

Interparameter Constraints

Example: `weighted-mean` from the `statistics.ss` file.

```
Function:   (weighted-mean w data)
Contract:   (->r ((w (vectorof real?))
                  (data (and/c (vectorof real?)
                               (lambda (x)
                                 (= (vector-length w)
                                    (vector-length data))))))
                  real?)
```

The contract specifies a function that takes two arguments: the first argument must be a vector of real numbers and the second must be a vector of real number whose length is the same as the first argument; and the function returns one real result.

```
> (require (planet "statistics.ss" ("williams" "science.plt" 2 0)))
> (weighted-mean #(1 2 3 4) #(4 3 2 1))
2.0
> (weighted-mean #(1 2 3 4) #(4 3 2))
6:3: top-level broke the contract
  (->r ((w ...) (data ...)) ...)
it had with (planet "statistics.ss" ("williams" "science.plt" 2 0))
on weighted-mean; expected <(and/c (vectorof real?)
...\0\statistics.ss:119:23)>, given: #3(4 3 2)
```

3.2 Infinities and Not-a-Number

PLT Scheme provides `+inf.0` (positive infinity), `-inf.0` (negative infinity), `+nan.0` (not-a-number), and `-nan.0` (same as `+nan.0`). In general, these are contagious and are passed as the result in subsequent numerical computations. However, operations with infinities and zero (both exact and inexact) can give non-intuitive results. For example:

- `(* 0 +inf.0)` → `0`
- `(/ 0 +inf.0)` → `0`
- `(/ 0 -inf.0)` → `0`
- `(* 0.0 +inf.0)` → `+nan.0`
- `(/ 0.0 -inf.0)` → `-0.0`

(Note that some of these may break naïve algorithms.)

The PLT Scheme Science Collection uses `+inf.0` to represent overflow and `-inf.0` to represent underflow in numerical computations. This is used in cases where the arguments to the function are within the range of the function, but the result is too large or too small to be represented. For example, `(gamma 200.0)` → `+inf.0`.

The PLT Scheme Science Collection uses `+nan.0` for domain errors – where the arguments match the contract, but the value cannot be computed. For example, `(gamma 0.0)` → `+nan.0`.

3.3 Exceptions

In PLT Scheme, an *exception* is a change in control flow, typically as the result of an error. Exceptions are defined in Chapter 6 Execeptions and Control Flow in the PLT MzScheme Language Manual[3].

The PLT Scheme Science Collection may raise exceptions for errors other than underflow, overflow, and domain errors. Also, underlying procedures and/or modules used by the science collection may raise exceptions as may errors in the implementation.

Chapter 4

Mathematical Functions

This chapter describes the basic mathematical constants and functions provided by the PLT Scheme Science Collection.

The constants and functions described in this chapter are defined in the `math.ss` file in the science collection and are made available using the form:

```
(require (planet "math.ss" ("williams" "science.plt" 2 0)))
```

4.1 Mathematical Constants

Table 5.1 shows the mathematical constants defined by the PLT Scheme Science Collection.

4.2 Infinities and Not-a-Number

PLT Scheme provides `+inf.0` (positive infinity), `-inf.0` (negative infinity), `+nan.0` (not a number), and `-nan.0` (same as `+nan.0`) as inexact numerical constants. The following functions are provided as a convenience for checking for infinities and not-a-number.

`nan?`

Function: `(nan? x)`
Contract: `(-> any? boolean?)`

This function returns true, `#t`, if x is not-a-number (i.e., equivalent to `+nan.0` or, therefore, `-nan.0`), and false, `#f`, otherwise. Note that this is not the same as `(not (number? x))`, which is true if x is not a number.

`infinite?`

Function: `(infinite? x)`
Contract: `(-> any? (union (integer-in -1 1) boolean?))`

`e`	The base of exponentials, e
`log2e`	The base 2 logarithm of e, $log_2 e$
`log10e`	The base 10 logarithm of e, $log_{10} e$
`sqrt2`	The square root of 2, $\sqrt{2}$
`sqrt1/2`	The square root of $\frac{1}{2}$, $\sqrt{\frac{1}{2}}$
`sqrt3`	The square root of 3, $\sqrt{3}$
`pi`	The constant π
`pi/2`	π divided by 2, $\frac{\pi}{2}$
`pi/4`	π divided by four, $\frac{\pi}{4}$
`sqrtpi`	The square root of π, $\sqrt{\pi}$
`2/sqrtpi`	2 divided by the square root of π, $\frac{2}{\sqrt{\pi}}$
`1/pi`	The reciprocal of π, $\frac{1}{\pi}$
`2/pi`	Twice the reciprocal of π, $\frac{2}{\pi}$
`ln10`	The natural log of 10, $\ln 10$
`ln2`	The natural log of 2, $\ln 2$
`lnpi`	The natural log of π, $\ln \pi$
`euler`	Euler's constant, γ

Table 4.1: Mathematical Constants

This function returns 1 if x is positive infinity (i.e., equivalent to `+inf.0`), -1 if x is negative infinity (i.e., equivalent to `-inf.0`), and false, `#f`, otherwise. Note that `(finite? x)` is not equivalent to `(not (infinite? x))`, since both `finite?` and `infinite?` return false for anything that is not a real number.

`finite?`

Function: `(finite? x)`
Contract: `(-> any? boolean?)`

This function returns true, `#t`, if x is a finite real number, and false, `#f` otherwise. Note that `(finite? x)` is not equivalent to `(not (infinite? x))`, since both `finite?` and `infinite?` return false for anything that is not a real number.

4.3 Elementary Functions

The following functions provide some elementary mathematical functions that are not provided by PLT Scheme.

`log1p`

Function: `(log1p x)`
Contract: `(-> real? number?)`

This function computes the value of $\log(1 + x)$ in a way that is accurate for small x.

`expm1`

Function: `(expm1 x)`
Contract: `(-> real? real?)`

This function computes the value of $\exp x - 1$ in a way that is accurate for small x.

`hypot`

Function: `(hypot x y)`
Contract: `(-> real? real? real?)`

This function computes the value of $\sqrt{x^2 + y^2}$ in a way that avoids overflow.

`acosh`

Function: `(acosh x)`
Contract: `(-> real? real?)`

This function computes the value of the hyperbolic arccosine, *arccosh*, of x.

`asinh`

Function: `(asinh x)`
Contract: `(-> real? real?)`

This function computes the value of the hyperbolic arcsine, *arcsinh*, of x.

`atanh`

Function: `(atahh x)`
Contract: `(-> real? real?)`

This function computes the value of the hyperbolic arctangent, *arctanh*, of x.

`ldexp`

Function: `(ldexp x e)`
Contract: `(-> real? integer? real?)`

This function computes the value of $x \times 2^e$.

`frexp`

Function: `(frexp x)`
Contract: `(-> real? (values real? integer?))`

This function splits the real number x into a normalized fraction f and exponent e, such that $x = f \times 2^e$ and $0.5 \leq f < 1$. The function returns f and e as multiple values. If x is zero, both f and e are returned as zero.

4.4 Testing the Sign of Numbers

`sign`

Function: `(sign x)`
Contract: `(-> real? (integer-in -1 1))`

This function returns the sign of x: 1 if $x \geq 0$ and -1 if $x < 0$. Note that the sign of zero is positive, regardless of its floating-point sign bit.

4.5 Approximate Comparisons of Real Numbers

It is sometimes useful to be able to compare two real (in particular, floating-point) numbers approximately, to allow for rounding and truncation errors. The following function implements the approximate floating-point comparison algorithm proposed by D.E. Knuth in Section 4.2.2 of *Seminumerical Algorithms* (3^{rd} edition).

`fcmp`

Function: `(fcmp x y epsilon)`
Contract: `(-> real? real? real? (integer-in -1 1))`

This function determines whether x and y are approximately equal to a relatively accuracy, *epsilon*. The relative accuracy is measured using an interval of $2 \times delta$, where $delta = 2^k \times epsilon$ and k is the maximum base 2 exponent of x and y as computed by the function `frexp`. If x and y lie within this interval, they are considered approximately equal and the function returns 0. Otherwise, if $x < y$, the function returns -1, or if $x > y$, the function returns 1.

The implementation of this function is based on the package `fcmp` by T.C. Belding.

Chapter 5

Special Functions

This chapter describes the special functions provided by the PLT Scheme Science Collection.

The functions described in this chapter are defined in the special-functions sub-collection of the science collection. The entire special-functions sub-collection can be made available using the form:

```
(require (planet "special-functions.ss"
                 ("williams" "science.plt" 2 0)))
```

The individual modules in the special-functions sub-collection can also be made available as described in the sections below.

5.1 Error Functions

The *error function* is described in Abramowitz and Stegun, Chapter 7. The functions are defined in the `error.ss` file in the special-functions sub-collection of the science collection and are made available using the form:

```
(require (planet "error.ss" ("williams" "science.plt" 2 0)
                 "special-functions"))
```

5.1.1 Error Function

`erf`

Function: `(erf x)`
Contract: `(-> real? (real-in -1.0 1.0))`

This function computes the error function:

$$erf(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

Example: Plot of `(erf x)` on the interval $[4, 4]$.

```
(require (planet "error.ss" ("williams" "science.plt" 2 0)
                 "special-functions"))
(require (lib "plot.ss" "plot"))

(plot (line erf)
      (x-min -4) (x-max 4)
      (y-min -1) (y-max 1)
      (title "Error Function, erf(x)"))
```

Figure 5.1 shows the resulting plot.

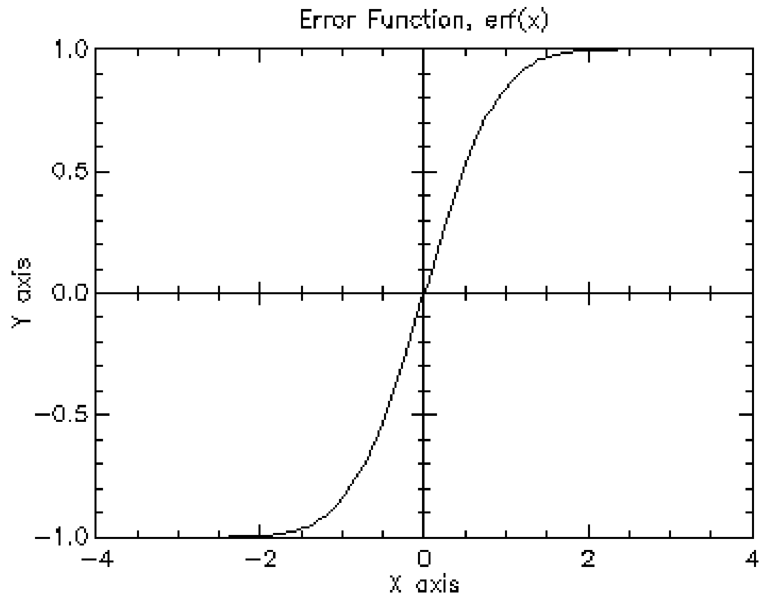

Figure 5.1: Plot of Error Function on [4, 4]

5.1.2 Complementary Error Function

`erfc`

Function: `(erfc x)`
Contract: `(-> real? (real-in 0.0 2.0))`

This function computes the complementary error function:

$$erfc(x) = 1 - erf(x) = \frac{2}{\sqrt{\pi}} \int_x^{\infty} e^{-t^2} dt$$

Example: Plot of `(erfc x)` on the interval [4, 4].

```
(require (planet "error.ss" ("williams" "science.plt" 2 0)
                 "special-functions"))
(require (lib "plot.ss" "plot"))

(plot (line erfc)
      (x-min -4) (x-max 4)
      (y-min 0) (y-max 2)
      (title "Complementary Error Function, erfc(x)"))
```

Figure 5.2 shows the resulting plot.

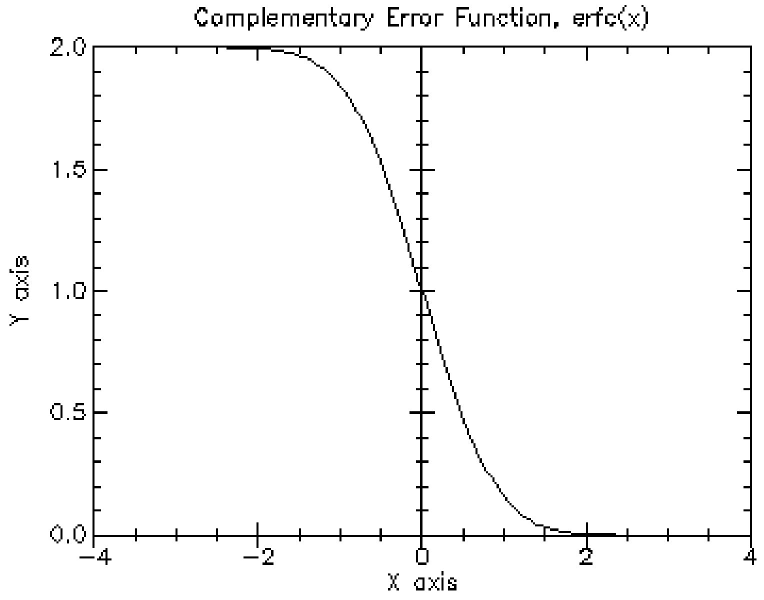

Figure 5.2: Plot of Complementary Error Function on [4, 4]

5.1.3 Hazard Function

The *hazard function* for the normal distribution, also known as the inverse Mill's ratio, is the ratio of the probability function, $P(x)$, to the survival function, $S(x)$, and is defined as:

$$h(x) = \frac{P(x)}{S(x)} = \sqrt{\frac{2\pi e^{\frac{x^2}{2}}}{erfc(x^{\sqrt{2}})}}$$

It decreases rapidly as x approaches $-\infty$ and asymptopes to $h(x)$ x as x approaches $+\infty$.

`hazard`

Function: `(hazard x)`
Contract: `(-> real? (>=/c 0.0))`

This function computes the hazard function for the normal distribution.

Example: Plot of `(hazard x)` on the interval $[-5, 10]$.

```
(require (planet "error.ss" ("williams" "science.plt" 2 0)
                 "special-functions"))
(require (lib "plot.ss" "plot"))

(plot (line hazard)
      (x-min -5) (x-max 10)
      (y-min 0) (y-max 10)
      (title "Hazard Function, hazard(x)"))
```

Figure 5.3 shows the resulting plot.

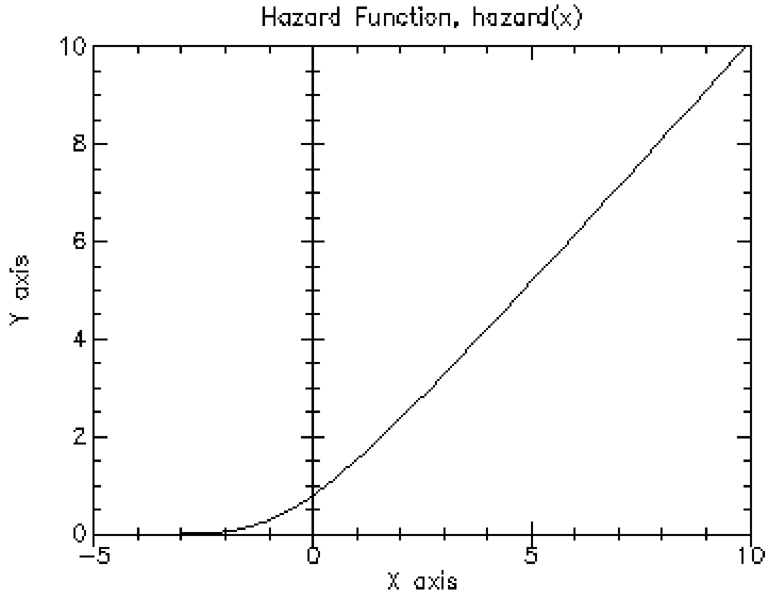

Figure 5.3: Plot of Hazard Function on $[-5, 10]$

5.2 Gamma Functions

The gamma functions are defined in the `gamma.ss` file in the special-functions sub-collection of the science collection and are made available using the form:

```
(require (planet "gamma.ss" ("williams" "science.plt" 2 0)
                 "special-functions"))
```

Note that the gamma functions (Section 5.2), psi functions (Section 5.3), and the zeta functions (Section 5.4) are defined in the same module, `gamma.ss`. This is because their definitions are interdependent and PLT Scheme does not allow circular module dependencies.

5.2.1 Gamma Function

The *gamma function* is defined by the integral:

$$\Gamma(x) \equiv \int_0^\infty t^{x-1} e^{-t} dt$$

It is related to the factorial function by $\Gamma(n) = (n-1)!$ for positive integer n. Further information on the gamma function can be found in Abramowitz & Stegun, Chapter 6.

`gamma`

Function: `(gamma x)`
Contract: `(-> real? real?)`

This function computes the gamma function, $\Gamma(x)$, subject to x not being a negative integer. The function is computed using the real Lanczos method. The maximum value of x such that $\Gamma(x)$ is not considered an overflow is given by the constant `gamma-xmax` and is 171.0.

Example: Plot of `(gamma x)` on the interval $(0, 6]$.

```
(require (planet "gamma.ss" ("williams" "science.plt" 2 0)
                 "special-functions"))
(require (lib "plot.ss" "plot"))

(plot (line gamma)
      (x-min 0.001) (x-max 6)
      (y-min 0) (y-max 120)
      (title "Gamma Function, gamma(x)"))
```

Figure 5.4 shows the resulting plot.

Example: Plot of `(gamma x)` on the interval $(-1, 0)$.

```
(require (planet "gamma.ss" ("williams" "science.plt" 2 0)
                 "special-functions"))
(require (lib "plot.ss" "plot"))

(plot (line gamma)
      (x-min -0.999) (x-max -0.001)
      (y-min -120) (y-max 0)
      (title "Gamma Function, gamma(x)"))
```

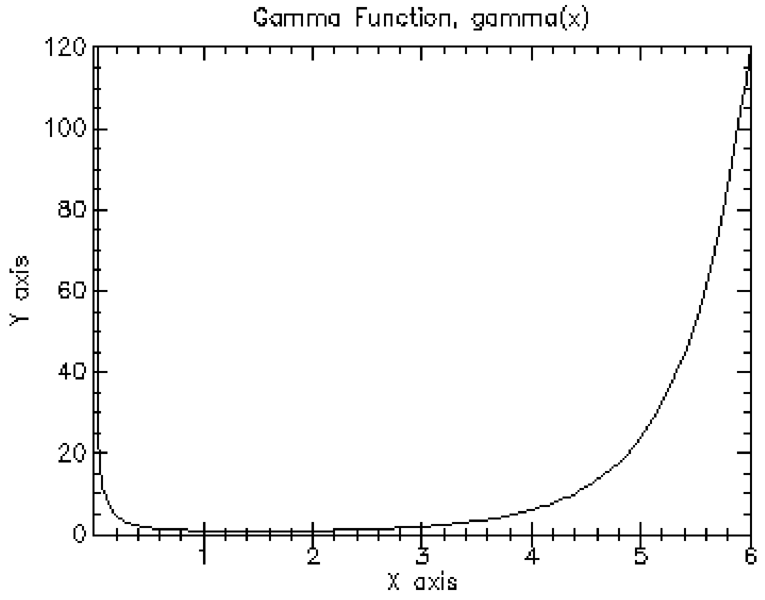

Figure 5.4: Plot of Gamma Function on $(0, 6]$

Figure 5.5 shows the resulting plot.

`lngamma`

Function: `(lngamma x)`
Contract: `(-> real? real?)`

This function computes the logarithm of the gamma function, $\log(\Gamma(x))$, subject to x not being a negative integer. For $x < 0$, the real part of $\log(\Gamma(x))$ is returned, which is equivalent to $\log(|\Gamma(x)|)$. The function is computed using the real Lanczos method.

`lngamma-sgn`

Function: `(lngamma-sgn x)`
Contract: `(-> real? (values real? (integer-in -1 1))`

This function computes the logarithm of the magnitude of the gamma function and its sign, subject to x not being a negative integer, and returns them as multiple values. The function is computed using the real Lanczos method. The value of the gamma function can be reconstructed using the relation $\Gamma(x) = sgn * \exp(resultlg)$, where *resultlg* and *sgn* are the returned values.

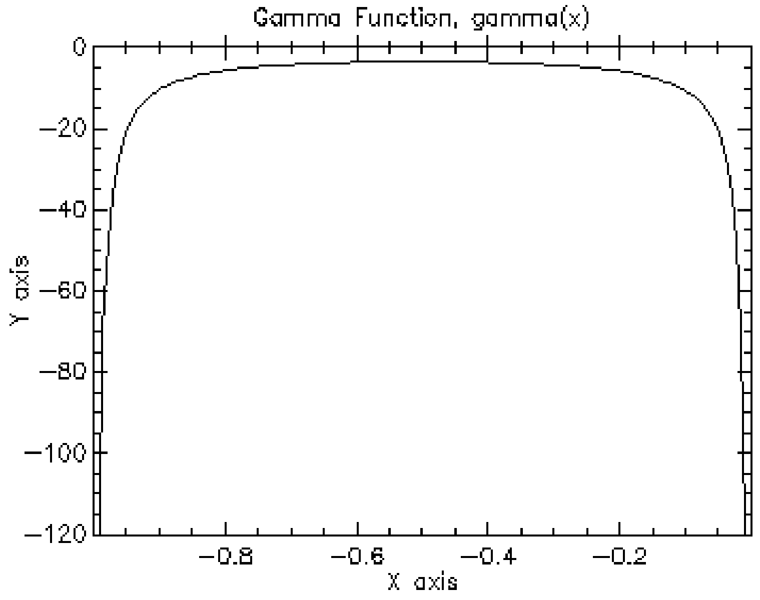

Figure 5.5: Plot of Gamma Function on $(-1, 0)$

`gamma-inv`

Function: `(gamma-inv x)`
Contract: `(-> real? real?)`

This function computes the reciprocal of the gamma function, $\frac{1}{\Gamma(x)}$, using the real Lanczos method.

5.2.2 Regulated Gamma Function

The *regulated gamma function* is given by

$$\Gamma^*(x) = \frac{\Gamma(x)}{\sqrt{2\pi}} x^{x-\frac{1}{2}} e^{-x}$$

`gammastar`

Function: `(gammastar x)`
Contract: `(-> (>/c 0.0) real?)`

This function computes the regulated gamma function, $\Gamma^*(x)$, for $x > 0$.

5.2.3 Factorial Function

The factorial of a positive integer n, $n!$, is defined as $n! = n \times (n-1) \times \ldots \times 2 \times 1$. By definition, $0! = 1$. The factorial function is related to the gamma function by $n! = \Gamma(n+1)$.

`factorial`

Function: `(fact n)`
Contract: `(-> natural-number? (>/c 1.0))`

This function computes the factorial of n, $n!$.

`lnfact`

Function: `(lnfact n)`
Contract: `(-> natural-number? (>=/c 0.0))`

This function computes the logarithm of the factorial of n, $\log(n!)$. The algorithm is faster than computing $\ln gamma(n+1)$ via `lngamma` for $n < 170$, but defers for larger n.

5.2.4 Double Factorial Function

The double factorial of n, $n!!$, is defined as $n!! = n \times (n-2) \times (n-4) \times \ldots$. By definition, $-1!! = 0!! = 1$.

`double-fact`

Function: `(double-fact n)`
Contract: `(-> natural-number? (>/c 1.0))`

This function computes the double factorial of n, $n!!$.

`lndouble-fact`

Function: `(lndouble-fact n)`
Contract: `(-> natural-number? (>=/c 0.0))`

This function computes the logarithm of the double factorial of n, $\log(n!!)$.

5.2.5 Binomial Coefficient Function

The binomial coefficient, b *choose* m, is defined as:

$$n \textit{ choose } m = \frac{n!}{m! \times (n-m)!}$$

`choose`

Function: `(choose n m)`
Contract: `(-> natural-number? natural-number? (>/c 1.0))`

This function computes the binomial coefficient for n and m, n *choose* m.

`lnchoose`

Function: `(lnchoose n m)`
Contract: `(-> natural-number? natural-number? (>/c 0.0))`

This function computes the logarithm of the binomial coefficient for n and m, $\log(n\ choose\ m)$.

5.3 Psi Functions

The psi functions are defined in `gamma.ss` in the special-functions sub-collection of the science collection and are made available using the form:

```
(require (planet "gamma.ss" ("williams" "science.plt" 2 0)
                 "special-functions"))
```

Note that the gamma functions (Section 5.2), psi functions (Section 5.3), and zeta functions (Section 5.4) are defined in the same module, `gamma.ss`. This is because their definitions are interdependent and PLT Scheme does not allow circular module dependencies.

5.3.1 Psi (Digamma) Functions

`psi-int`

Function: `(psi-int n)`
Contract: `(-> (integer-in 1 +inf.0) real?)`

This function computes the digamma function, $\psi(n)$, for positive integer n. The digamma function is also called the Psi function.

`psi`

Function: `(psi x)`
Contract: `(-> real? real?)`

This function returns the digamma function, $\psi(x)$, for general x, $x \neq 0$.

`psi-1piy`

Function: `(psi-1piy y)`
Contract: `(-> real? real?)`

This function computes the real part of the digamma function on the line $1 + iy$, $Re[\psi(1 + iy)]$.

Example: Plot of `(psi x)` on the interval $(0, 5]$.

```
(require (planet "gamma.ss" ("williams" "science.plt" 2 0)
                 "special-functions"))
(require (lib "plot.ss" "plot"))
```

```
(plot (line psi)
      (x-min 0.001) (x-max 5)
      (y-min -5) (y-max 5)
      (title "Psi (Digamma) Function, psi(x)"))
```

Figure 5.6 shows the resulting plot.

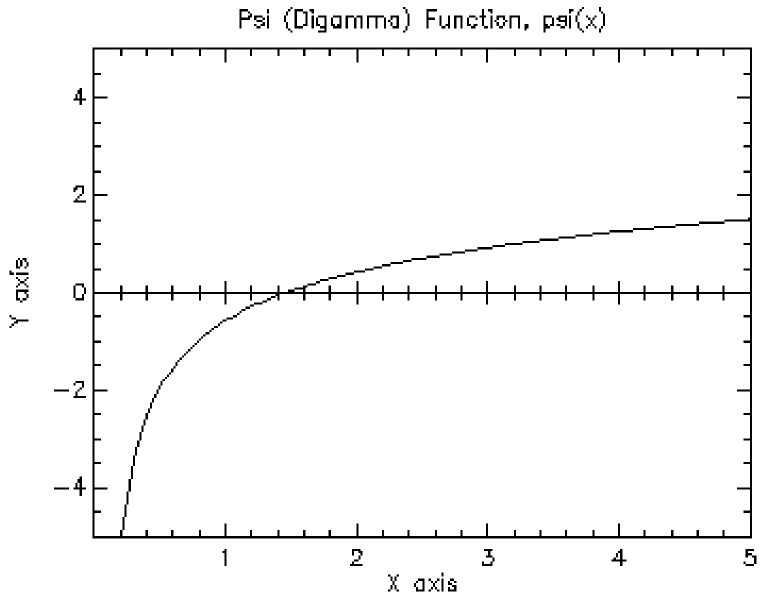

Figure 5.6: Plot of Psi (Digamma) Function on $(0, 5]$

5.3.2 Psi-1 (Trigamma) Functions

`psi-1-int`

Function: `(psi-1-int n)`
Contract: `(-> (integer-in 1 +inf.0) real?)`

This function computes the trigamma function $\psi'(n)$ for positive integer n.

`psi-1`

Function: `(psi-1 x)`
Contract: `(-> real? real?)`

This function computes the trigamma function $\psi'(x)$ for general x.

Example: Plot of `(psi-1 x)` on the interval $(0, 5]$.

```
(require (planet "gamma.ss" ("williams" "science.plt" 2 0)
                 "special-functions"))
(require (lib "plot.ss" "plot"))

(plot (line psi-1)
      (x-min 0.005) (x-max 5)
      (y-min 0) (y-max 5)
      (title "Psi-1 (Trigamma) Function, psi-1(x)"))
```

Figure 5.7 shows the resulting plot.

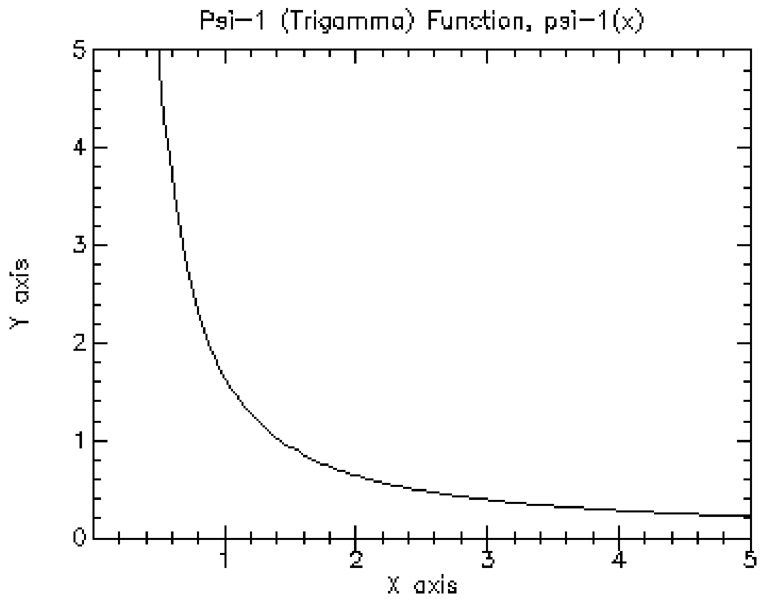

Figure 5.7: Plot of Psi-1 (Trigamma) Function on $(0, 5]$

5.3.3 Psi-n (Polygamma) Function

`psi-n`

Function: `(psi-n n x)`
Contract: `(-> natural-number? (>/c 0.0) real?)`

This function computes the polygamma function $\psi^m(x)$ for $m >= 0$, $x > 0$.

Example: Plot of `(psi-n n x)` on the interval $(0, 5]$.

```
(require (planet "gamma.ss" ("williams" "science.plt" 2 0)
                 "special-functions"))
(require (lib "plot.ss" "plot"))

(plot (line (lambda (x) (psi-n 3 x)))
      (x-min 0.05) (x-max 5)
      (y-min 0) (y-max 10)
      (title "Psi-n (Polygamma) Function, psi-n(3, x)"))
```

Figure 5.8 shows the resulting plot.

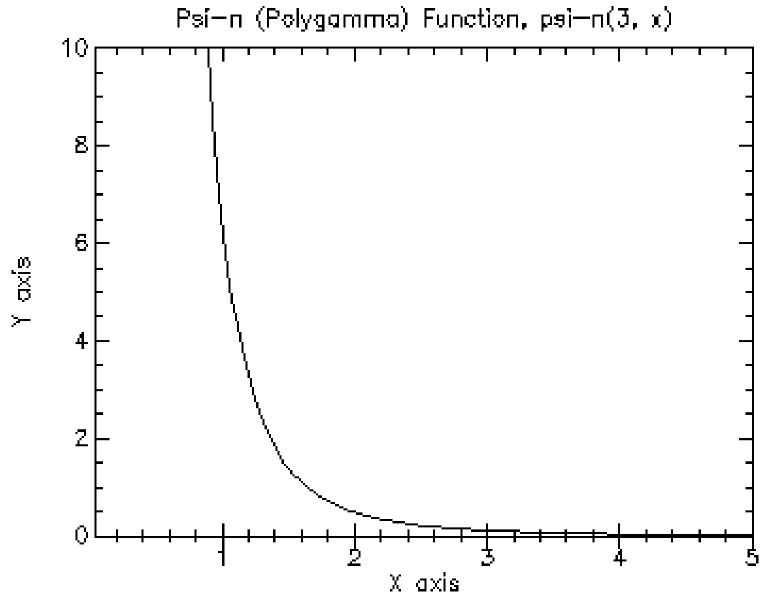

Figure 5.8: Plot of Psi-n (Polygamma), $n = 3$, Function on $(0, 5]$

5.4 Zeta Functions

The Riemann zeta function is defined in Abramowitz and Stegun, Section 23.2. The zeta functions are defined in the `gamma.ss` file in the special-functions sub-collection of the science collection and are made available using the form:

```
(require (lib "gamma.ss" "science" "special-functions"))
```

Note that the gamma functions (Section 5.2), psi functions (Section 5.3), and zeta functions (Section 5.4) are defined in the same module, `gamma.ss`. This is because their definitions are interdependent and PLT Scheme does not allow circular module dependencies.

5.4.1 Riemann Zeta Functions

The Riemann zeta function is defined by the infinite sum

$$\zeta(s) = \sum_{k=1}^{\infty} k^{-s}$$

`zeta-int`

<u>Function</u>: `(zeta-int n)`
<u>Contract</u>: `(-> integer? real?)`

This function computes the Reimann zeta function $\zeta(n)$ for integer n, $n \neq 1$.

`zeta`

<u>Function</u>: `(zeta s)`
<u>Contract</u>: `(-> real? real?)`

This function computes the Reimann zera function $\zeta(s)$ for arbitraty s, $s \neq 1$.

Example: Plot of `(zeta x)` on the interval $[-5, 5]$.

```
(require (planet "gamma.ss" ("williams" "science.plt" 2 0)
                 "special-functions"))
(require (lib "plot.ss" "plot"))

(plot (line zeta)
      (x-min -5) (x-max 5)
      (y-min -5) (y-max 5)
      (title "Reimann Zeta Function, zeta(x)"))
```

Figure 5.9 shows the resulting plot.

5.4.2 Riemann Zeta Functions Minus One

For large positive arguments, the Reimann zeta function approaches one. In this region the fractional part is interesting amd, therefore, we need a function to evaluate it explicitly.

`zetam1-int`

<u>Function</u>: `(zetam1-int n)`
<u>Contract</u>: `(-> integer? real?)`

This function computes $\zeta(n) - 1$ for integer n, $n \neq 1$.

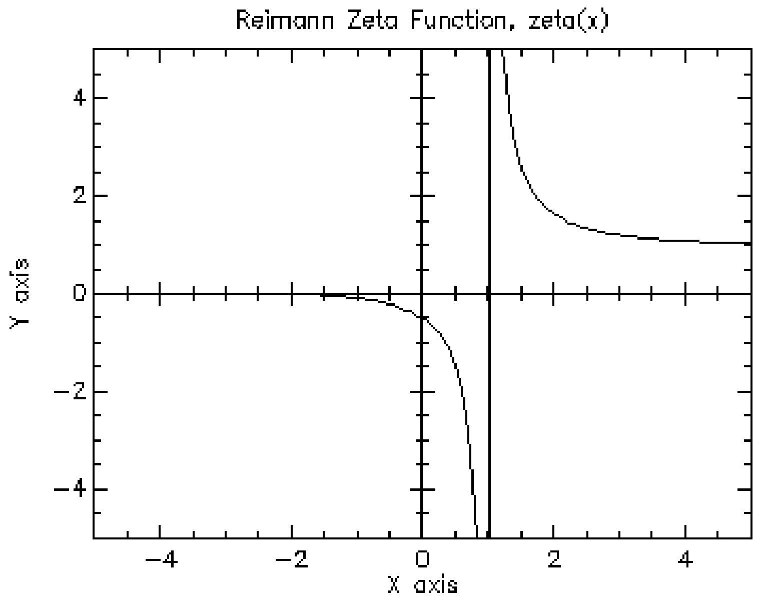

Figure 5.9: Plot of Reimann Zeta Function on $[-5, 5]$

`zetam1`

Function: `(zetam1 s)`
Contract: `(-> real? real?)`

This function computes $\zeta(n) - 1$ for arbitrary s, $s \neq 1$.

5.4.3 Hurwitz Zeta Function

The Hurwitx zeta function is defined by

$$\zeta(s,q) = \sum_{k=0}^{\infty} (k+q)^{-s}$$

`hzeta`

Function: `(hzeta s q)`
Contract: `(-> (>/c 1.0) (>/c 0.0) real?)`

This function computes the Hurwitz zeta function $\zeta(s,q)$ for $s > 1$, $q > 0$.

Example: Plot of `(hzeta x)` on the interval $(1, 5]$.

```
(require (planet "gamma.ss" ("williams" "science.plt" 2 0)
                 "special-functions"))
```

```
(require (lib "plot.ss" "plot"))

(plot (line (lambda (x) (hzeta x 2.0)))
      (x-min 1.001) (x-max 5)
      (y-min 0) (y-max 5)
      (title "Hurwitz Zeta Function, hzeta(x, 2.0)"))
```

Figure 5.10 shows the resulting plot.

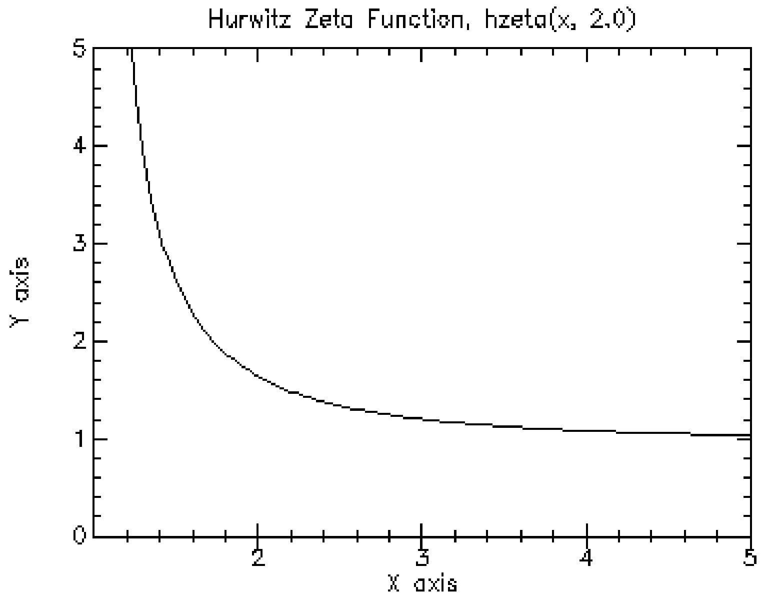

Figure 5.10: Plot of Hurwitz Zeta Function, $q = 2.0$, on $(1, 5]$

5.4.4 Eta Functions

The eta function is defined by

$$\eta(s) = (1 - 2^{1-s})\zeta(s)$$

`eta-int`

Function: `(eta-int n)`
Contract: `(-> integer? real?)`

This function computes the eta function $\eta(n)$ for integer n.

`eta`

<u>Function</u>: `(eta s)`
<u>Contract</u>: `(-> real? real?)`

This function computes the eta function $\eta(s)$ for arbitrary s.

Example: Plot of `(eta x)` on the interval $[-10, 10]$.

```
(require (planet "gamma.ss" ("williams" "science.plt" 2 0)
                 "special-functions"))
(require (lib "plot.ss" "plot"))

(plot (line eta)
      (x-min -10) (x-max 10)
      (y-min -5) (y-max 5)
      (title "Eta Function, eta(x)"))
```

Figure 5.11 shows the resulting plot.

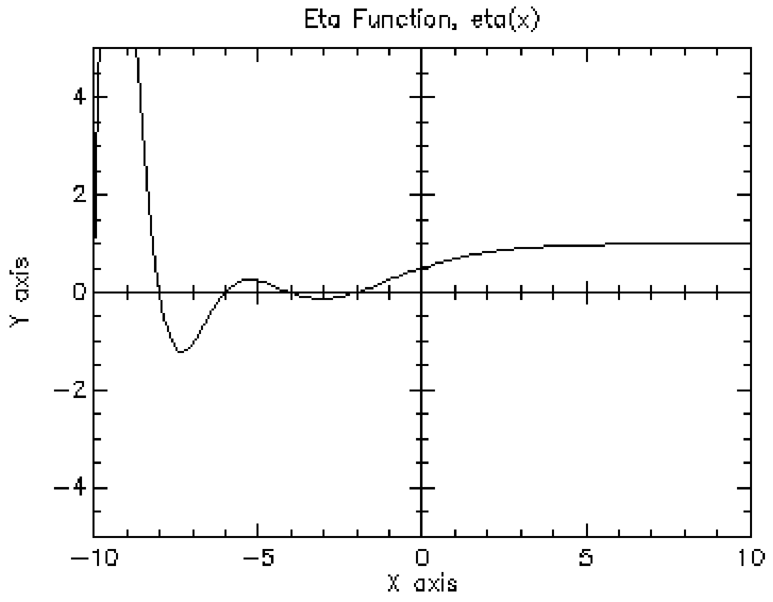

Figure 5.11: Plot of Eta Function on $[-10, 10]$

Chapter 6

Random Number Generation

The PLT Scheme Science Collection provides additional functionality to the PLT Scheme implementation of SRFI 27 by Sabastian Egner, which, in turn, is a 54-bit implementation of Pierre L'Ecuyer's MRG32k3a pseudo-random number generator.[1]

The functions described in this chapter are defined in the `random-source.ss` file in the science collection and are made available using the form:

```
(require (planet "random-source.ss" ("williams" "science.plt" 2 0)))
```

6.1 The SRFI 27 Specification

The following functions are defined in the SRFI specification and are, in turn, provided by the `random-source` module. The contract shows here are for documentation purposes only – the PLT Scheme implementation of SRFI 27 does not define contracts for its functions.

random-integer

Function: `(random-integer n)`
Contract: `(-> (integer-in 1 +inf.0) natural-number?)`

This functions returns the next integer in $\{0, \ldots, n-1\}$ obtained from the `default-random-source`. Subsequent results of this function appear to be independent uniformly distributed over the range $\{0, \ldots, n-1\}$. The argument n must be a positive integer, otherwise an error is signaled.

random-real

Function: `(random-real)`
Contract: `(-> (real-in 0.0 1.0))`

[1]This is equivalent to the `gsl-rmg-cmrg` combined multiple recursive generator by L'Ecuyer in the GSL.

This function returns the next number, x, $0 < x < 1$ obtained from `default-random--source`. Subsequent results of this procedure appear to be independently uniformly distributed.

`default-random-source`

Variable: `default-random-source`

Defines the random source from which `random-integer` and `random-real` have been derived using `random-source-make-integers` and `random-source-make-reals`. Note that an assignment to `default-random-source` does not change the behavior of `random` or `random-integer`; it is strongly recommended not to assign a new value to `default--random-source`.

`make-random-source`

Function: `(make-random-source)`
Contract: `(-> random-source?)`

This function creates a new random source. A random source created with `make-random-source` represents a deterministic stream of random bits. Each random stream obtained as `(make-random-source)` generates the same stream of values unless the state is modified with one of the functions below.

`random-source?`

Function: `(random-source? x)`
Contract: `(-> any? boolean?)`

This function returns true, `#t`, if x is a random source, otherwise false, `#f`, is returned.

`random-source-state-ref` and `random-source-state-set!`

Function: `(random-source-state-ref s)`
Contract: `(-> random-source? any)`
Function: `(random-source-state-set! s state)`
Contract: `(-> random-source? any? any)`

These functions get and set the current state of the random source s. The implementation of the internal state of a random source is not defined.

`random-source-randomize!`

Function: `(random-source-randomize! s)`
Contract: `(-> random-source? any)`

This function makes an effort to set the state if the random source s to a truly random state.

`random-source-pseudo-randomize!`

Function: `(random-source-pseudo-randomize! s i j)`
Contract: `(-> random-source? natural-number? natural-number> any)`

This function changes the state of the random stream *s* into the initial state of the $(i, j)^{th}$ independent random source, where *i* and *j* are non-negative integers. This procedure provides a mechanism to obtain a large number of independent random sources, indexed by two integers. In contrast to `random-source-randomize!`, this procedure is entirely deterministic.

`random-source-make-integers`

Function: `(random-source-make-integers s)`
Contract: `(-> random-source? procedure?)`

This function returns a procedure to generate random integers using the random source *s*. The resulting procedure takes a single argument *n*, which must be a positive integer, and returns the next independent uniformly distributed integer from the interval $\{0, \ldots, n-1\}$ by advancing the state of the random source *s*.

`random-source-make-reals`

Function: `(random-source-make-reals s unit)`
Function: `(random-source-make-reals s)`
Contract:
```
(case-> (-> random-source? real? procedure?
        (-> random-source? procedure?))
```

This function returns a procedure to generate random real numbers $0 < x < 1$ using the random source *s*. The resulting procedure is called without arguments.

6.2 Additional Random Number Functionality

The science collection provides additional functionality to that provided by SRFI 27.

6.2.1 The `current-random-source` Parameter

The main additional functionality is to define a parameter[2], `current-random-source`, which provides a separate random source reference for each thread. The default value for this random source reference is `default-random-stream`.

The use of the `current-random-source` parameter overcomes the difficulty with assignment to `default-random-source`. However, the routines `random-integer` and `random-real` use the `default-random-source` variable and are unaware of the `current-random-source` parameter.

[2]See PLT Scheme: Reference Manual, Section 7.7 Parameters

`current-random-source`

```
Function:   (current-random-source s)
Function:   (current-random-source)
Contract:   (case-> (-> random-source? any)
                    (-> random-source?))
```

Gets or sets the current random source. A guard procedure ensures that the value of `current-random-source` is indeed a random source, as determined by `random-source?`, otherwise a type error is raised.

`with-random-source`

```
Macro:   (with-random-source s
            body ...)
```

Executes the body with `current-random-source` set to the random source *s*.

`with-new-random-source`

```
Macro:   (with-new-random-source s
            body ...)
```

Executes the body with `current-random-source` set to a newly created random source.

6.2.2 Uniform Random Numbers

The science collection provides alternatives to the `random-integer` and `random-real` functions that are aware of the `current-random-source` parameter. They also provide a more convenient interface than `random-source-make-integers` and `random-source-make-reals`.

`random-uniform-int`

```
Function:   (random-uniform-int s n)
Function:   (random-uniform-int n)
Contract:   (case->
               (-> random-source? (integer-in 1 +inf.0)
                   natural-number?)
               (-> (integer-in 1 +inf.0) natural-number?))
```

This function returns the next integer in $\{0, \ldots, n-1\}$ obtained from the random source *s* or `(current-random-source)` if *s* is not specified. Subsequent results of this function appear to be independent uniformly distributed over the range $\{0, \ldots, n-1\}$. The argument *n* must be a positive integer.

`random-uniform`

```
Function:   (random-uniform s)
Function:   (random-uniform)
Contract:   (case-> random-source? (real-in 0.0 1.0))
                    (real-in 0.0 1.0)))
```

This function returns the next number x, $0 < x < 1$, obtained from the random source s or `(current-random-source)` if s is not specified. Subsequent results of this function appear to be independent uniformly distributed.

6.2.3 Miscellaneous Functions

These functions provide an alternative set of functions to get or set the state of a random state. These functions match the conventions for structures in PLT Scheme.

`random-source-state`

Function: `(random-source-state s)`
Contract: `(-> random-source? any)`

The same as `random-source-state-ref`.

`set-random-source-state!`

Function: `(set-random-source-state! s state)`
Contract: `(-> random-source? any? any)`

The same as `random-source-state-set!`.

6.2.4 Random Source Vectors

These function provide a convenient method for generating a vector of repeatable random sources.

`make-random-source-vector`

```
Function:   (make-random-source-vector n i)
Function:   (make-random-source-vector n)
Contract:   (case-> (-> natural-number/c natural-number/c
                        (vectorof random-source?))
                    (-> natural-number/c (vectorof random-source?)))
```

This funtion returns a vector of random sources of length n. If i is provided, the j^{th} random source is initialized using `(random-source-pseudo-randomize! i j)`. If i is not provided, the i^{th} random dource is initialized using `(random-source-pseudo-randomize! i 0)`. Note that this is not the same as having i default to 0.

6.3 Examples

Example: Histogram of uniform random numbers.

```
(require (planet "random-source.ss" ("williams" "science.plt" 2 0)))
(require (planet "histogram-with-graphics.ss"
                 ("williams" "science.plt" 2 0)))

(let ((h (make-histogram-with-ranges-uniform 40 0 1))
      (s (make-random-source)))
```

```
  (random-source-randomize! s)
  (with-random-source s
    (do ((i 0 (+ i 1)))
        ((= i 10000) (void))
      (histogram-increment! h (random-uniform))))
  (histogram-plot h "Histogram of Uniform Random Numbers"))
```

Figure 6.1 shows an example of the resulting histogram.

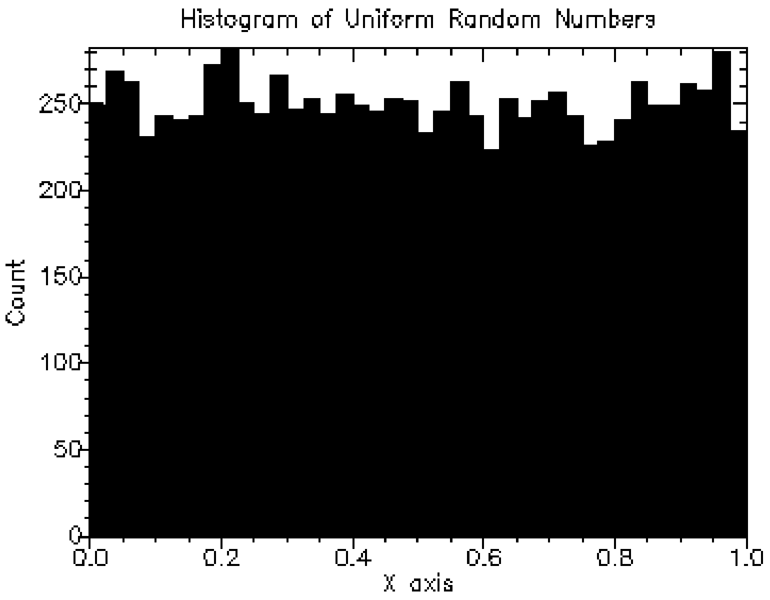

Figure 6.1: Histogram of Uniform Random Numbers

Example: Histogram of uniform random integers.

```
(require (planet "random-source.ss" ("williams" "science.plt" 2 0)))
(require (planet "histogram-with-graphics.ss"
                 ("williams" "science.plt" 2 0)))

(let ((h (make-discrete-histogram))
      (s (make-random-source)))
  (random-source-randomize! s)
  (with-random-source s
    (do ((i 0 (+ i 1)))
        ((= i 10000) (void))
      (discrete-histogram-increment! h (random-uniform-int 10))))
  (discrete-histogram-plot
   h "Histogram of Uniform Random Integers"))
```

Figure 6.2 shows an example of the resulting histogram.

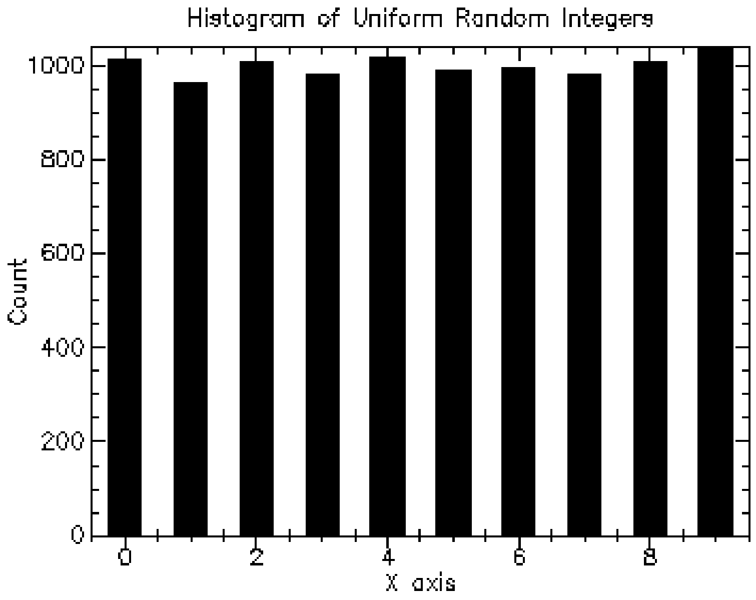

Figure 6.2: Histogram of Uniform Random Integers

Chapter 7

Random Number Distributions

This chapter describes the functions for generating random variates and computing their probability distributions provided by the PLT Scheme Science Collection.

The functions described in this chapter are defined in the random-distributions sub-collection of the science collection. All of the modules in the random-distributions sub-collection can be made available using the form:

```
(require (planet "random-distributions.ss"
                 ("williams" "science.plt" 2 0)))
```

The random distribution graphics are provided as separate modules. To include the random distribution graphics routines, use the following form:

```
(require (planet "random-distributions-with-graphics.ss"
                 ("williams" "science.plt" 2 0)))
```

The individual modules in the random-distributions sub-collection can also be made available as described in the sections below.

7.1 The Beta Distribution

The beta distribution functions are defined in the `beta.ss` file in the random-distributions sub-collection of the science-collection and are made available using the form:

```
(require (planet "beta.ss" ("williams" "science.plt" 2 0)
                 "random-distributions"))
```

7.1.1 Random Variates from the Beta Distribution

`random-beta`

```
Function:   (random-beta s a b)
Function:   (random-beta a b)
Contract:   (case->
               (-> random-source? real? real? (real-in 0.0 1.0))
               (-> real? real? (real-in 0.0 1.0)))
```

This function returns a random variate from the beta distribution with parameters a and b using the random source s or `(current-random-source)` if s is not provided.

Example: Histogram of random variates from the beta distribution with parameters 2.0 and 3.0.

```
(require (planet "beta.ss" ("williams" "science.plt" 2 0)
                 "random-distributions"))
(require (planet "histogram-with-graphics.ss"
                 ("williams" "science.plt" 2 0)))

(let ((h (make-histogram-with-ranges-uniform 40 0.0 1.0)))
  (do ((i 0 (+ i 1)))
      ((= i 10000) (void))
    (histogram-increment! h (random-beta 2.0 3.0)))
  (histogram-plot h "Histogram of Beta Distribution"))
```

Figure 7.1 shows the resulting histogram.

7.1.2 Beta Distribution Density Functions

`beta-pdf`

```
Function:   (beta-pdf x a b)
Contract:   (-> real? real? real? (>=/c 0.0))
```

This function computes the probability density, $p(x)$, at x for the beta distribution with parameters a and b.

7.1.3 Beta Distribution Graphics

The beta distribution graphics functions are defined in the file `beta-graphics.ss` in the random-distributions sub-collection of the science collection and are made available using the form:

```
(require (planet "beta-graphics.ss" ("williams" "science.plt" 2 0)
                 "random-distributions"))
```

`beta-plot`

```
Function:   (beta-plot a b)
Contract:   (-> real? real? any)
```

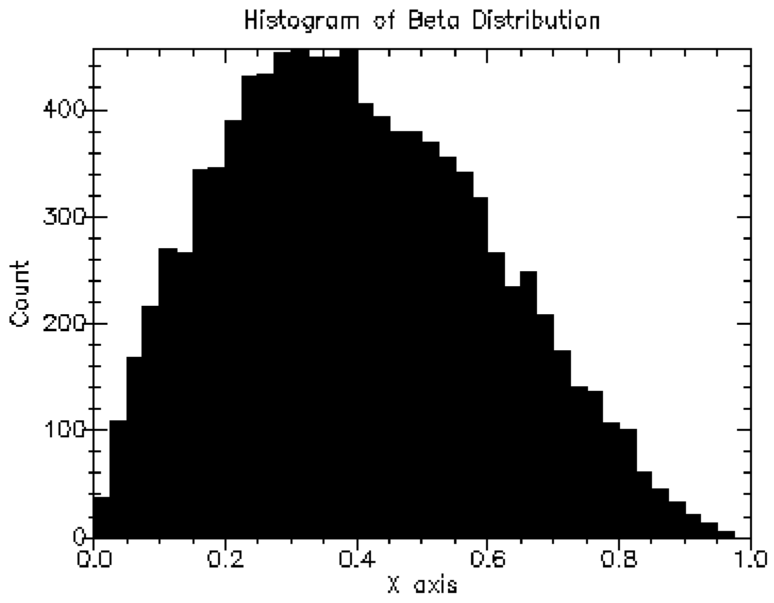

Figure 7.1: Histogram of Random Variates from Beta (2.0, 3.0)

This function returns a plot of the probability density of the beta distribution with parameters *a* and *b*. The plot is produced by the plot collection provided with PLT Scheme (PLoT Scheme).

Example: Plot of probability density of the beta distribution with parameters 2.0 and 3.0.

```
(require (planet "beta-graphics.ss" ("williams" "science.plt" 2 0)
                 "random-distributions"))
(beta-plot 2.0 3.0)
```

Figure 7.2 shows the resulting plot.

7.2 The Bivariate Gaussian Distribution

The bivariate Gaussian distribution functions are defined in the `bivariate-gaussian.ss` file in the random-distributions sub-collection of the science-collection and are made available using the form:

```
(require (planet "bivatiate-gaussian.ss" ("williams" "science.plt" 2 0)
                 "random-distributions"))
```

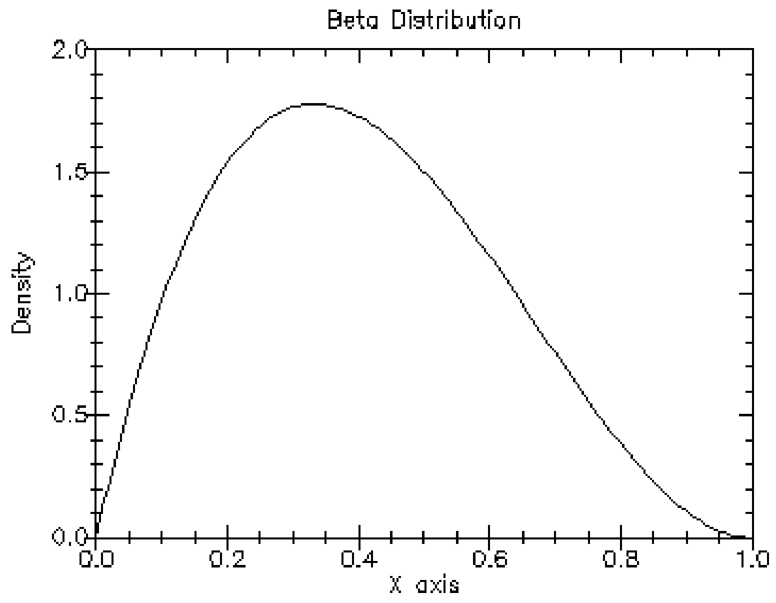

Figure 7.2: Plot of Probability Density for Beta(2.0, 3.0)

7.2.1 Random Variates from the Bivariate Gaussian Distribution

`random-bivariate-gaussian`

```
Function:  (random-bivariate-gaussian s sigma-x sigma-y rho)
Function:  (random-bivariate-gaussian sigma-x sigma-y rho)
Contract:  (case->
              (-> random-source? (>=/c 0.0) (>=/c 0.0)
                  (real-in -1.0 1.0) (values real? real?))
              (-> (>=/c 0.0) (>=/c 0.0)
                  (real-in -1.0 1.0) (values real? real?))
```

This function returns a pair of correlated Gaussian variates, with mean 0, correlation coefficient *rho*, and standard deviations *sigma-x* and *sigma-y* in the x and y directions using the random source s or `(current-random-source)` if s is not provided. The correlation coefficient *rho* must lie between -1 and 1.

Example: Histogram of random variates from the bivariate Gaussian distribution standard deviations 1.0 and 1.0 in the x and y directions and correlation coefficient 0.0.

```
(require (planet "bivariate-gaussian.ss" ("williams" "science.plt" 2 0)
                 "random-distributions"))
(require (planet "histogram-2d-with-graphics.ss"
                 ("williams" "science.plt" 2 0)))
```

```
(let ((h (make-histogram-2d-with-ranges-uniform
          20 20 -3.0 3.0 -3.0 3.0)))
  (do ((i 0 (+ i 1)))
      ((= i 10000) (void))
    (let-values (((x y) (random-bivariate-gaussian 1.0 1.0 0.0)))
      (histogram-2d-increment! h x y)))
  (histogram-2d-plot h "Histogram of Bivariate Gaussian Distribution"))
```

Figure 7.3 shows the resulting 2D histogram.

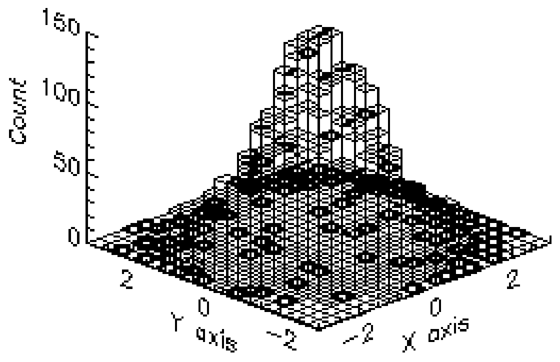

Figure 7.3: Histogram of Random Variates from Bivariate Gaussian (1.0, 1.0, 0.0)

7.2.2 Bivariate Gaussian Distribution Density Functions

`bivariate-gaussian-pdf`

Function: `(bivariate-gaussian-pdf x y sigma-x sigma-y rho)`
Contract: `(-> real? real? (>=/c 0.0 (>=/c 0.0) (real-in -1.0 0.0) real?)`

This function computes the probability density, $p(x, y)$, at (x, y) for the bivariate Gaussian distribution with standard deviations *sigma-x* and *sigma-y* in the x and y directions and correlation coefficient *rho*.

7.2.3 Bivariate Gaussian Distribution Graphics

The bivariate Gaussian distribution graphics functions are defined in the file `bivariate-gaussian-graphics.ss` in the random-distributions sub-collection of the science collection and are made available using the form:

```
(require (planet "bivariate-gaussian-graphics.ss"
                 ("williams" "science.plt" 2 0)
                 "random-distributions"))
```

`bivariate-gaussian-plot`

```
Function:   (bivariate-gaussian-plot sigma-x sigma-y rho)
Contract:   (-> (>=/c 0.0) (>=/c 0.0) (real-in -1.0 1.0) any)
```

This function returns a plot of the probability density of the bivariate Gaussian distribution with standard deviations *sigma-x* and *sigma-y* in the x and y directions and correlation coefficient *rho*. The plot is produced by the plot collection provided with PLT Scheme (PLoT Scheme).

Example: Plot of probability density of the bivariate Gaussian distribution with standard deviations 1.0 and 1.0 in the x and y directions and correlation coefficient 0.0.

```
(require (planet "bivariate-gaussian-graphics.ss"
                 ("williams" "science.plt" 2 0)
                 "random-distributions"))
(bivariate-gaussian-plot 1.0 1.0 0.0)
```

Figure 7.4 shows the resulting plot.

7.3 The Chi-Squared Distribution

The chi-squared distribution functions are defined in the `chi-squared.ss` file in the random-distributions sub-collection of the science-collection and are made available using the form:

```
(require (planet "chi-squared.ss" ("williams" "science.plt" 2 0)
                 "random-distributions"))
```

7.3.1 Random Variates from the Chi-Squared Distribution

`random-chi-squared`

```
Function:   (random-chi-squared s nu)
Function:   (random-chi-squared nu)
Contract:   (case->
               (-> random-source? real? (>=/c 0.0))
               (-> real? (>=/c 0.0)))
```

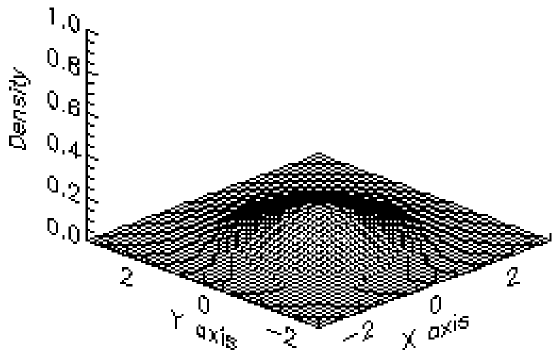

Figure 7.4: Plot of Probability Density for Bivariate Gaussian (1.0, 1.0, 0.0)

This function returns a random variate from the chi squared distribution with *nu* degrees of freedom using the random source *s* or `(current-random-source)` if *s* is not provided.

Example: Histogram of random variates from the chi squared distribution with 3.0 degrees of freedom.

```
(require (planet "chi-squared.ss" ("williams" "science.plt" 2 0)
                 "random-distributions"))
(require (planet "histogram-with-graphics.ss"
                 ("williams" "science.plt" 2 0)))

(let ((h (make-histogram-with-ranges-uniform 40 0.0 10.0)))
  (do ((i 0 (+ i 1)))
      ((= i 10000) (void))
    (histogram-increment! h (random-chi-squared 3.0)))
  (histogram-plot h "Histogram of Chi Squared Distribution"))
```

Figure 7.5 shows the resulting histogram.

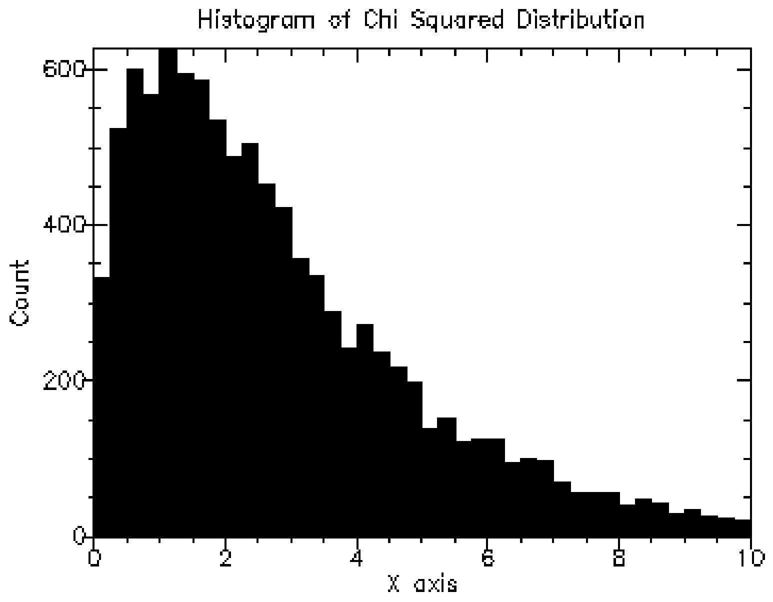

Figure 7.5: Histogram of Random Variates from Chi-Squared (3.0))

7.3.2 Chi-Squared Distribution Density Functions

`chi-squared-pdf`

Function: `(chi-squared-pdf x nu)`
Contract: `(-> real? real? (>=/c 0.0))`

This function computes the probability density, $p(x)$, at x for the chi squared distribution with *nu* degrees of freedom.

7.3.3 Chi-Squared Distribution Graphics

The chi squared distribution graphics functions are defined in the file `chi-squared--graphics.ss` in the random-distributions sub-collection of the science collection and are made available using the form:

```
(require (planet "chi-squared-graphics.ss"
                 ("williams" "science.plt" 2 0)
                 "random-distributions"))
```

`chi-squared-plot`

Function: `(chi-squared-plot nu)`
Contract: `(-> real? any)`

This function returns a plot of the probability density of the chi squared distribution with *nu* degrees of freedom. The plot is produced by the plot collection provided with PLT Scheme (PLoT Scheme).

Example: Plot of probability density of the chi squared distribution with 3.0 degrees of freedom.

```
(require (planet "chi-squared-graphics.ss"
                 ("williams" "science.plt" 2 0)
                 "random-distributions"))
(chi-squared-plot 3.0)
```

Figure 7.6 shows the resulting plot.

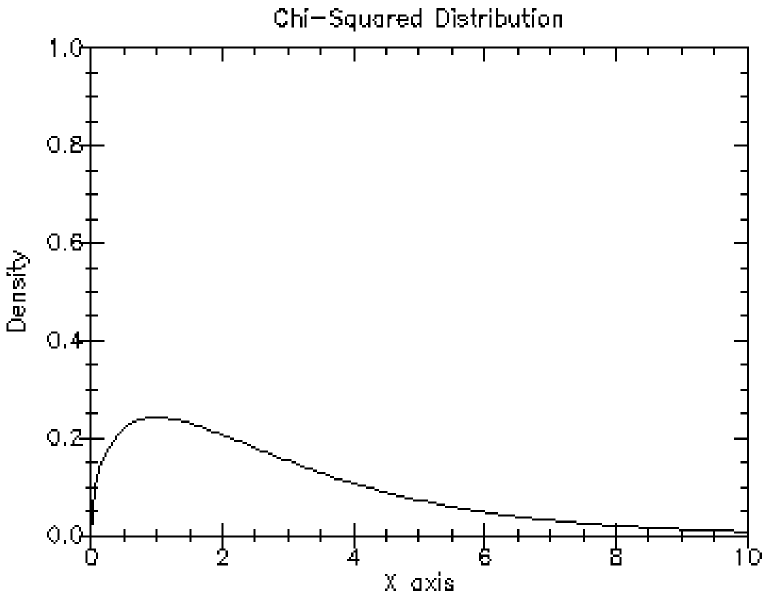

Figure 7.6: Plot of Probability Density for Chi-Squared (3.0)

7.4 The Exponential Distribution

The exponential distribution functions are defined in the `exponential.ss` file in the random-distributions sub-collection of the science-collection and are made available using the form:

```
(require (planet "exponential.ss" ("williams" "science.plt" 2 0)
                 "random-distributions"))
```

7.4.1 Random Variates from the Exponential Distribution

`random-exponential`

```
Function:  (random-exponential s mu)
Function:  (random-exponential mu)
Contract:  (case->
              (-> random-source? (>/c 0.0) (>=/c 0.0))
              (-> (>/c 0.0) (>=/c 0.0)))
```

This function returns a random variate from the exponential distribution with mean *mu* using the random source *s* or `(current-random-source)` if *s* is not provided.

Example: Histogram of random variates from the exponential distribution with mean 1.0.

```
(require (planet "exponential.ss" ("williams" "science.plt" 2 0)
                 "random-distributions"))
(require (planet "histogram-with-graphics.ss"
                 ("williams" "science.plt" 2 0)))

(let ((h (make-histogram-with-ranges-uniform 40 0.0 8.0)))
  (do ((i 0 (+ i 1)))
      ((= i 10000) (void))
    (histogram-increment! h (random-exponential 1.0)))
  (histogram-plot h "Histogram of Exponential Distribution"))
```

Figure 7.7 shows the resulting histogram.

7.4.2 Exponential Distribution Density Functions

`exponential-pdf`

```
Function:  (exponential-pdf x mu)
Contract:  (-> real? real? (>=/c 0.0))
```

This function computes the probability density, $p(x)$, at x for the exponential distribution with mean *mu*.

`exponential-cdf`

```
Function:  (exponential-cdf x mu)
Contract:  (-> real? real? (real-in 0.0 1.0))
```

This function computes the cumulative density, $d(x)$, at x for the exponential distribution with mean *mu*.

7.4.3 Exponential Distribution Graphics

The exponential distribution graphics functions are defined in the file `exponential-graphics.ss` in the random-distributions sub-collection of the science collection and are made available using the form:

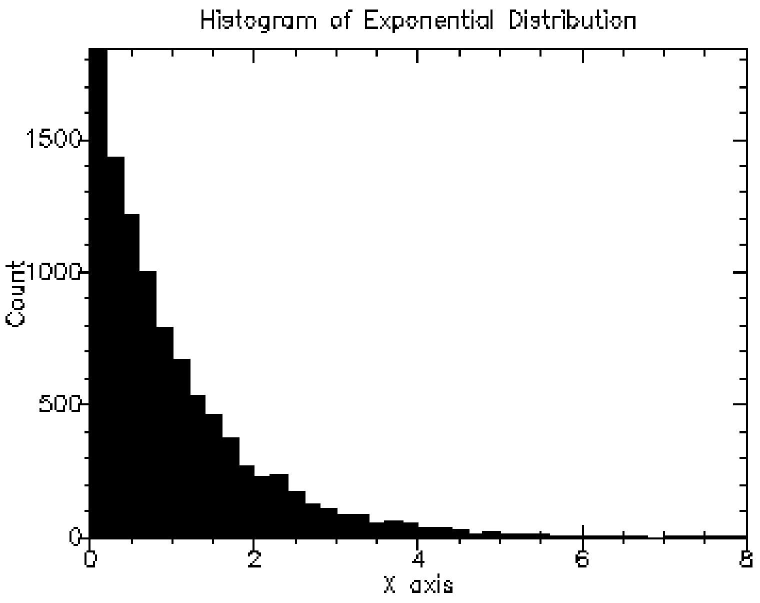

Figure 7.7: Histogram of Random Variates from Exponential (1.0))

```
(require (planet "exponential-graphics.ss"
                 ("williams" "science.plt" 2 0)
                 "random-distributions"))
```

`exponential-plot`

Function: `(exponential-plot mu)`
Contract: `(-> (>=/c 0.0) any)`

This function returns a plot of the probability density and cumulative density of the exponential distribution with mean *mu*. The plot is produced by the plot collection provided with PLT Scheme (PLoT Scheme).

Example: Plot of probability density of the exponential distribution with mean 3.0.

```
(require (planet "exponential-graphics.ss"
                 ("williams" "science.plt" 2 0)
                 "random-distributions"))
(exponential-plot 3.0)
```

Figure 7.8 shows the resulting plot.

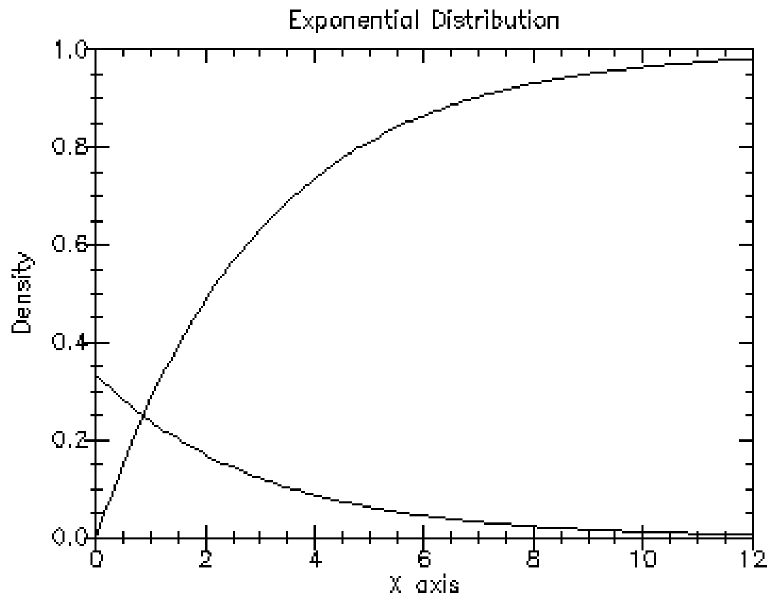

Figure 7.8: Plot of Probability Density for Exponential (1.0)

7.5 The F-Distribution

The F-distribution functions are defined in the `f-distribution.ss` file in the random-distributions sub-collection of the science-collection and are made available using the form:

```
(require (planet "f-distribution.ss" ("williams" "science.plt" 2 0)
                 "random-distributions"))
```

7.5.1 Random Variates from the F-Distribution

`random-f-distribution`

Function: `(random-f-distribution s nu1 nu2)`
Function: `(random-f-distribution nu1 nu2)`
Contract:
```
(case->
   (-> random-source? real? real? (>=/c 0.0))
   (-> real? real? (>=/c 0.0)))
```

This function returns a random variate from the F-distribution with *nu1* and *nu2* degrees of freedom using the random source *s* or `(current-random-source)` if *s* is not provided.

Example: Histogram of random variates from the F-distribution with 2.0 and 3.0 degrees of freedom.

```
(require (planet "f-distribution.ss" ("williams" "science.plt" 2 0)
                 "random-distributions"))
(require (planet "histogram-with-graphics.ss"
                 ("williams" "science.plt" 2 0)))

(let ((h (make-histogram-with-ranges-uniform 40 0.0 10.0)))
  (do ((i 0 (+ i 1)))
      ((= i 10000) (void))
    (histogram-increment! h (random-f-distribution 2.0 3.0)))
  (histogram-plot h "Histogram of F-Distribution"))
```

Figure 7.9 shows the resulting histogram.

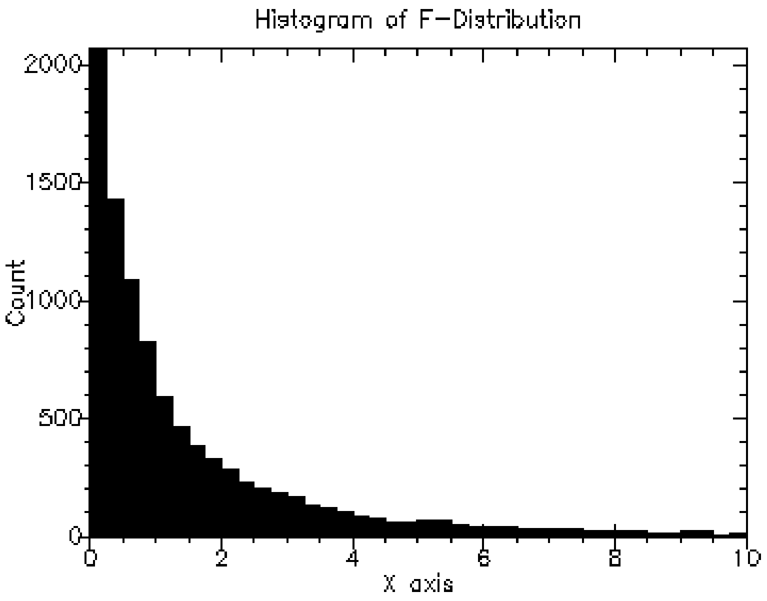

Figure 7.9: Histogram of Random Variates from F-Distribution (2.0, 3.0))

7.5.2 F-Distribution Density Functions

`f-distribution-pdf`

Function: `(f-distribution-pdf x nu1 nu2)`
Contract: `(-> real? real? real? (>=/c 0.0))`

This function computes the probability density, $p(x)$, at x for the F-distribution with *nu1* and *nu2* degrees of freedom.

7.5.3 F-Distribution Graphics

The exponential distribution graphics functions are defined in the file `f-distribution-graphics.ss` in the random-distributions sub-collection of the science collection and are made available using the form:

```
(require (planet "f-distribution-graphics.ss"
                 ("williams" "science.plt" 2 0)
                 "random-distributions"))
```

`f-distribution-plot`

Function: `(f-distribution-plot nu1 nu2)`
Contract: `(-> (>=/c 0.0) any)`

This function returns a plot of the probability density of the F-distribution with *nu1* and *nu2* degrees of freedom. The plot is produced by the plot collection provided with PLT Scheme (PLoT Scheme).

Example: Plot of probability density of the F-distribution distribution with 2.0 and 3.0 degrees of freedom.

```
(require (planet "f-distribution-graphics.ss"
                 ("williams" "science.plt" 2 0)
                 "random-distributions"))
(f-distribution-plot 2.0 3.0)
```

Figure 7.10 shows the resulting plot.

7.6 The Flat (Uniform) Distribution

The flat (uniform) distribution functions are defined in the `flat.ss` file in the random-distributions sub-collection of the science-collection and are made available using the form:

```
(require (planet "flat.ss" ("williams" "science.plt" 2 0)
                 "random-distributions"))
```

7.6.1 Random Variates from the Flat Distribution

`random-flat`

Function: `(random-flat s a b)`
Function: `(random-flat a b)`
Contract:
```
(case->
   (->r ((s random-source?)
         (a real?)
         (b (>/c a))
        real?)
   (->r ((a real?)
         (b (>/c a))
        real?))
```

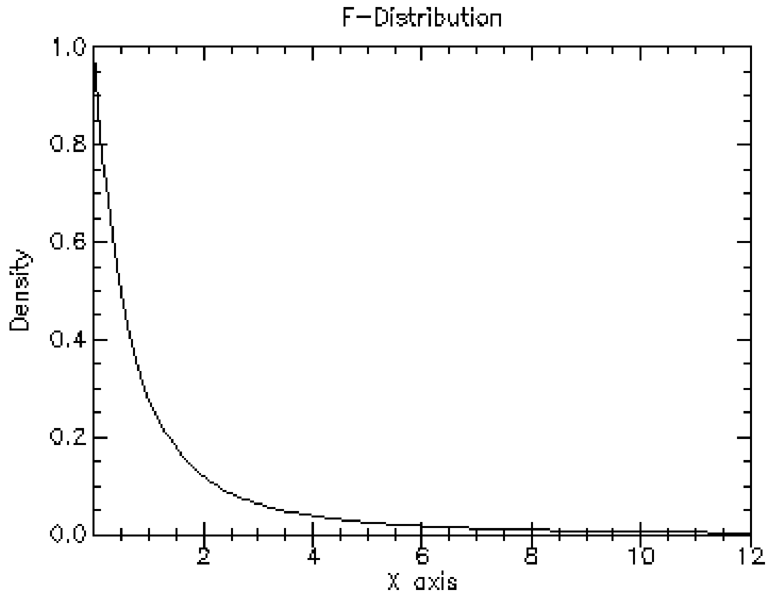

Figure 7.10: Plot of Probability Density for F-Distribution (2.0, 3.0)

This function returns a random variate from the flat (uniform) distribution from a to b using the random source s or `(current-random-source)` if s is not provided.

Example: Histogram of random variates from the flat (uniform) distribution from 1.0 to 4.0.

```
(require (planet "flat.ss" ("williams" "science.plt" 2 0)
                 "random-distributions"))
(require (planet "histogram-with-graphics.ss"
                 ("williams" "science.plt" 2 0)))

(let ((h (make-histogram-with-ranges-uniform 40 1.0 4.0)))
  (do ((i 0 (+ i 1)))
      ((= i 10000) (void))
    (histogram-increment! h (random-flat 1.0 4.0)))
  (histogram-plot h "Histogram of Flat (Uniform) Distribution"))
```

Figure 7.11 shows the resulting histogram.

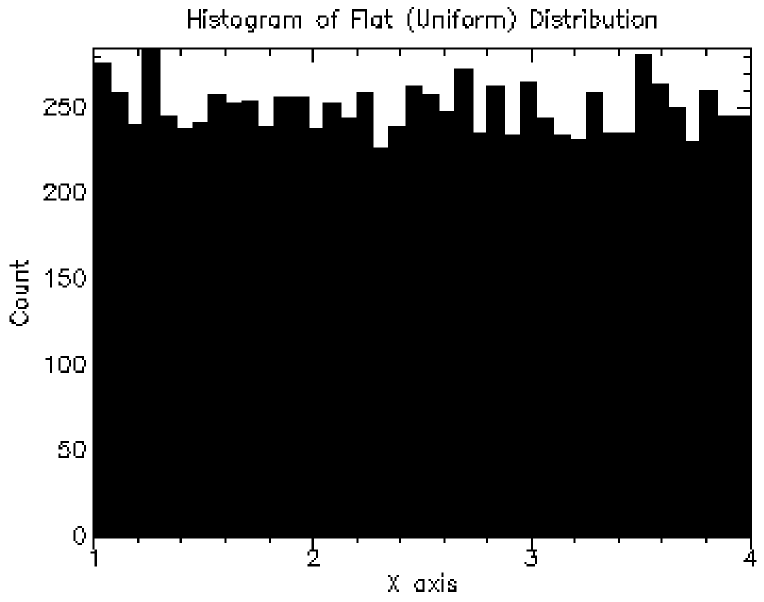

Figure 7.11: Histogram of Random Variates from Flat (Uniform) (1.0, 4.0)

7.6.2 Flat Distribution Density Functions

`flat-pdf`

Function: `(flat-pdf x a b)`

Contract:
```
(->r ((x real?)
      (a real?)
      (b (>/c a)))
     (>=/c 0.0))
```

This function computes the probability density, $p(x)$, at x for the flat (uniform) distribution from a to b.

`flat-cdf`

Function: `(flat-cdf x a b)`

Contract:
```
(->r ((x real?)
      (a real?)
      (b (>/c a)))
     (real-in 0.0 1.0))
```

This function computes the cumulative density, $d(x)$, at x for the flat (uniform) distribution from a to b.

7.6.3 Flat Distribution Graphics

The flat (uniform) distribution graphics functions are defined in the file `flat-graphics-.ss` in the random-distributions sub-collection of the science collection and are made available using the form:

```
(require (planet "flat-graphics.ss" ("williams" "science.plt" 2 0)
                 "random-distributions"))
```

`flat-plot`

Function: `(flat-plot a b)`
Contract:
```
(->r ((a real?)
      (b (>/c a)))
     any)
```

This function returns a plot of the probability density of the flat distribution from *a* to *b*. The plot is produced by the plot collection provided with PLT Scheme (PLoT Scheme).

Example: Plot of probability density and cumulative density of the flat (uniform) distribution from 1.0 to 4.0.

```
(require (planet "flat-graphics.ss" ("williams" "science.plt" 2 0)
                 "random-distributions"))
(flat-plot 1.0 4.0)
```

Figure 7.12 shows the resulting plot.

7.7 The Gamma Distribution

The gamma distribution functions are defined in the `gamma.ss` file in the random-distributions sub-collection of the science-collection and are made available using the form:

```
(require (planet "gamma.ss" ("williams" "science.plt" 2 0)
                 "random-distributions"))
```

7.7.1 Random Variates from the Gamma Distribution

`random-gamma`

Function: `(random-gamma s a b)`
Function: `(random-gamma a b)`
Contract:
```
(case->
   (-> random-source? (>/c 0.0) real? (>=/c 0.0))
   (-> (>/c 0.0 real? (>=/c 0.0)))
```

This function returns a random variate from the gamma distribution with parameters *a* and *b* using the random source *s* or `(current-random-source)` if *s* is not provided.

Example: Histogram of random variates from the gamma distribution with parameters 3.0 and 3.0.

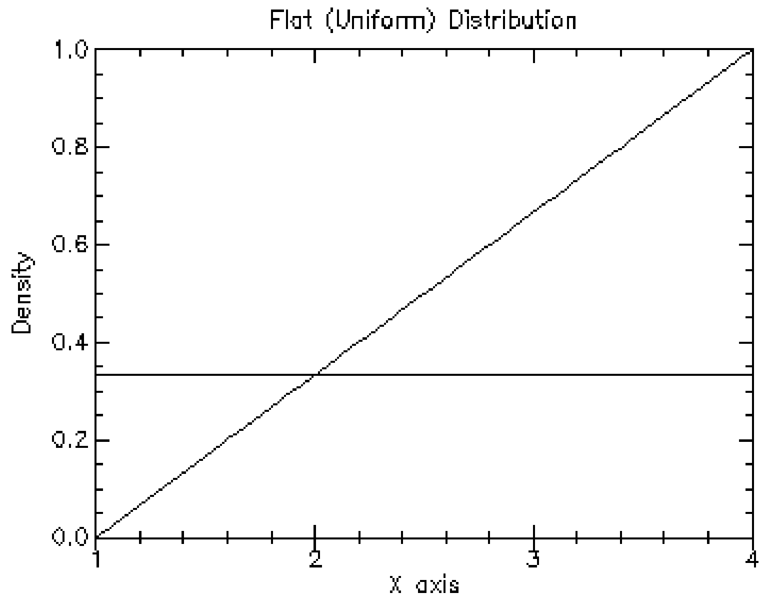

Figure 7.12: Plot of Probability Density for Flat (Uniform) (1.0, 4.0)

```
(require (planet "gamma.ss" ("williams" "science.plt" 2 0)
                 "random-distributions"))
(require (planet "histogram-with-graphics.ss"
                 ("williams" "science.plt" 2 0)))

(let ((h (make-histogram-with-ranges-uniform 40 0.0 24.0)))
  (do ((i 0 (+ i 1)))
      ((= i 10000) (void))
    (histogram-increment! h (random-gamma 3.0 3.0)))
  (histogram-plot h "Histogram of Gamma Distribution"))
```

Figure 7.13 shows the resulting histogram.

7.7.2 Gamma Distribution Density Functions

`gamma-pdf`

Function: `(gamma-pdf x a b)`
Contract: `(-> real? (>/c 0.0) real? (>=/c 0.0))`

This function computes the probability density, $p(x)$, at x for the gamma distribution with parameters a and b.

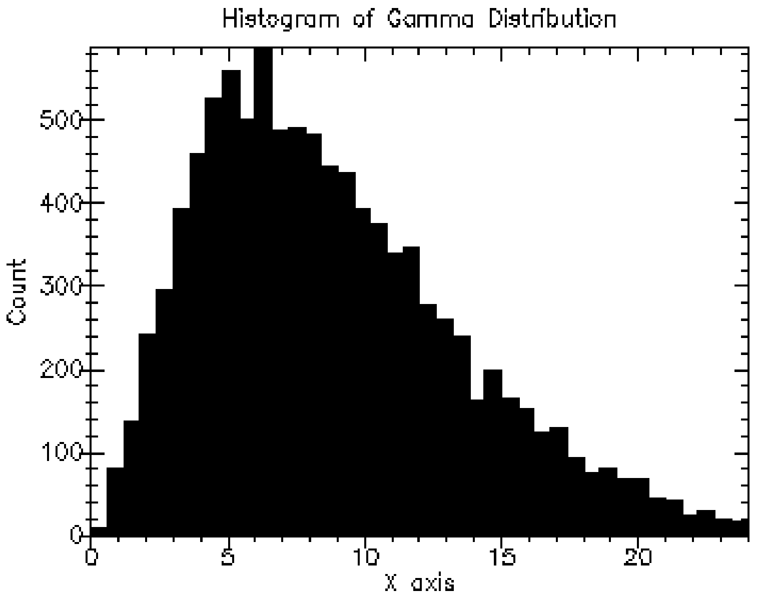

Figure 7.13: Histogram of Random Variates from Gamma (3.0, 3.0)

7.7.3 Gamma Distribution Graphics

The gamma distribution graphics functions are defined in the file `gamma-graphics.ss` in the random-distributions sub-collection of the science collection and are made available using the form:

```
(require (planet "gamma-graphics.ss" ("williams" "science.plt" 2 0)
                 "random-distributions"))
```

`gamma-plot`

Function: `(gamma-plot a b)`
Contract: `(-> (>/c 0.0) real?)`

This function returns a plot of the probability density of the gamma distribution with parameters *a* and *b*. The plot is produced by the plot collection provided with PLT Scheme (PLoT Scheme).

Example: Plot of probability density of the gamma distribution with parameters 3.0 and 3.0.

```
(require (planet "gamma-graphics.ss" ("williams" "science.plt" 2 0)
                 "random-distributions"))
(gamma-plot 3.0 3.0)
```

Figure 7.14 shows the resulting plot.

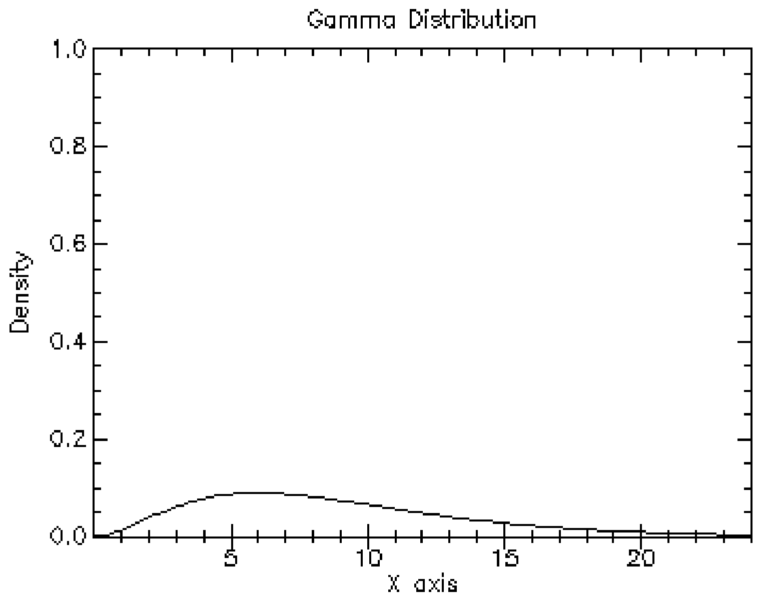

Figure 7.14: Plot of Probability Density for Gamma (3.0, 3.0)

7.8 The Gaussian Distribution

The Gaussian distribution functions are defined in the `gaussian.ss` file in the random-distributions sub-collection of the science-collection and are made available using the form:

```
(require (planet "gaussian.ss" ("williams" "science.plt" 2 0)
                 "random-distributions"))
```

7.8.1 Random Variates from the Gaussian Distribution

`random-gaussian`

Function: `(random-gaussian s mu sigma)`
Function: `(random-gaussian mu sigma)`
Contract:
```
(case->
   (-> random-source? real? (>=/c 0.0) real?)
   (-> real? (>=/c 0.0) real?))
```

This function returns a random variate from the Gaussian (normal) distribution with mean *mu* and standard deviation *sigma* using the random source *s* or `(current--random-source)` if *s* is not provided. This function uses the Box-Mueller algorithm that requires two calls to the random source *s*.

Example: Histogram of random variates from the Gaussian (normal) distribution with mean 10.0 and standard deviation 2.0.

```
(require (planet "gaussian.ss" ("williams" "science.plt" 2 0)
                 "random-distributions"))
(require (planet "histogram-with-graphics.ss"
                 ("williams" "science.plt" 2 0)))

(let ((h (make-histogram-with-ranges-uniform 40 4.0 16.0)))
  (do ((i 0 (+ i 1)))
      ((= i 10000) (void))
    (histogram-increment! h (random-gaussian 10.0 2.0)))
  (histogram-plot h "Histogram of Gaussian Distribution"))
```

Figure 7.15 shows the resulting histogram.

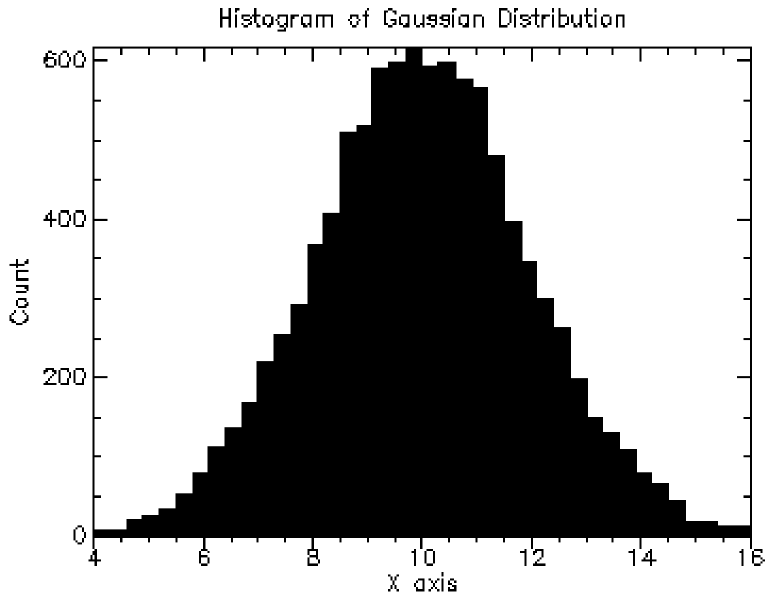

Figure 7.15: Histogram of Random Variates from Gaussian (Normal) (10.0, 2.0)

```
random-unit-gaussian
```

Function:	`(random-unit-gaussian s)`
Function:	`(random-unit-gaussian)`

Contract:	`(case->`
	`  (-> random-source? real?)`
	`  (-> real?))`

This function returns a random variate from the Gaussian (normal) distribution with mean 0.0 and standard deviation 1.0 using the random source *s* or `(current-random-source)` if *s* is not provided. This function uses the Box-Mueller algorithm that requires two calls to the random source *s*.

Example: Histogram of random variates from the unit Gaussian (normal) distribution.

```
(require (planet "gaussian.ss" ("williams" "science.plt" 2 0)
                 "random-distributions"))
(require (planet "histogram-with-graphics.ss"
                 ("williams" "science.plt" 2 0)))

(let ((h (make-histogram-with-ranges-uniform 40 -3.0 3.0)))
  (do ((i 0 (+ i 1)))
      ((= i 10000) (void))
    (histogram-increment! h (random-unit-gaussian)))
  (histogram-plot h "Histogram of Unit Gaussian Distribution"))
```

Figure 7.16 shows the resulting histogram.

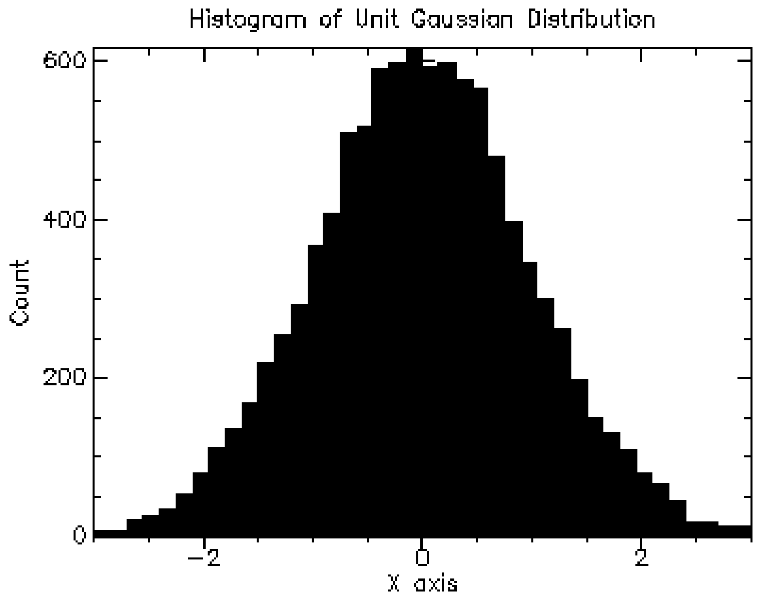

Figure 7.16: Histogram of Random Variates from Unit Gaussian (Normal)

`random-gaussian-ratio-method`

```
Function:   (random-gaussian-ratio-method s mu sigma)
Function:   (random-gaussian-ratio-method mu sigma)
Contract:   (case->
               (-> random-source? real? (>=/c 0.0) real?)
               (-> real? (>=/c 0.0) real?))
```

This function returns a random variate from the Gaussian (normal) distribution with mean *mu* and standard deviation *sigma* using the random source *s* or `(current-random-source)` if *s* is not provided. This function uses the Kinderman-Monahan ratio method.

`random-unit-gaussian-ratio-method`

```
Function:   (random-unit-gaussian-ratio-method s)
Function:   (random-unit-gaussian-ratio-method)
Contract:   (case->
               (-> random-source? real?)
               (-> real?))
```

This function returns a random variate from the Gaussian (normal) distribution with mean 0.0 and standard deviation 1.0 using the random source *s* or `(current-random-source)` if *s* is not provided. This function uses the Kinderman-Monahan ratio method.

7.8.2 Gaussian Distribution Density Functions

`gaussian-pdf`

```
Function:   (gaussian-pdf x mu sigma)
Contract:   (-> real? real? (>/c 0.0) (>=/c 0.0))
```

This function computes the probability density, $p(x)$, at x for the Gaussian (normal) distribution with mean *mu* and standard deviation *sigma*.

`gaussian-cdf`

```
Function:   (gaussian-cdf x mu sigma)
Contract:   (-> real? real? (>/c 0.0) (real-in 0.0 1.0))
```

This function computes the cumulative density, $d(x)$, at x for the Gaussian (normal) distribution with mean *mu* and standard deviation *sigma*.

`unit-gaussian-pdf`

```
Function:   (unit-gaussian-pdf x)
Contract:   (-> real? (>=/c 0.0))
```

This function computes the probability density, $p(x)$, at x for the Gaussian (normal) distribution with mean 0.0 and standard deviation 1.0.

`unit-gaussian-pdf`

Function: `(unit-gaussian-cdf x)`
Contract: `(-> real? (real-in 0.0 1.0))`

This function computes the cumulative density, $d(x)$, at x for the Gaussian (normal) distribution with mean 0.0 and standard deviation 1.0.

7.8.3 Gaussian Distribution Graphics

The Gaussian distribution graphics functions are defined in the file `gaussian-graphics-.ss` in the random-distributions sub-collection of the science collection and are made available using the form:

```
(require (planet "gaussian-graphics.ss" ("williams" "science.plt" 2 0)
                 "random-distributions"))
```

`gaussian-plot`

Function: `(gaussian-plot mu sigma)`
Contract: `(-> real? (>/c 0.0) any)`

This function returns a plot of the probability density and cumulative density of the Gaussian (normal) distribution with mean *mu* and standard deviation *sigma*. The plot is produced by the plot collection provided with PLT Scheme (PLoT Scheme).

Example: Plot of probability density and cumulative density of the Gaussian (normal) distribution with mean 10.0 and standard deviation 2.0.

```
(require (planet "gaussian-graphics.ss" ("williams" "science.plt" 2 0)
                 "random-distributions"))
(gaussian-plot 10.0 2.0)
```

Figure 7.17 shows the resulting plot.

`unit-gaussian-plot`

Function: `(unit-gaussian-plot)`
Contract: `(-> any)`

This function returns a plot of the probability density and cumulative density of the Gaussian (normal) distribution with mean 0.0 and standard deviation 1.0. The plot is produced by the plot collection provided with PLT Scheme (PLoT Scheme).

Example: Plot of probability density and cumulative density of the Gaussian (normal) distribution with mean 0.0 and standard deviation 1.0.

```
(require (planet "gaussian-graphics.ss" ("williams" "science.plt" 2 0)
                 "random-distributions"))
(unit-gaussian-plot)
```

Figure 7.18 shows the resulting plot.

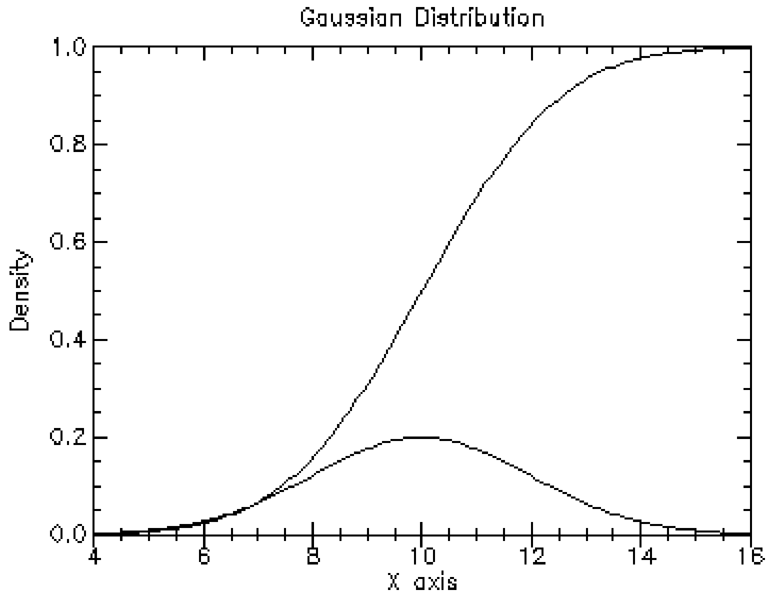

Figure 7.17: Plot of Probability Density for Gaussian (Normal) (10.0, 2.0)

7.9 The Gaussian Tail Distribution

The Gaussian tail distribution functions are defined in the `gaussian-tail.ss` file in the random-distributions sub-collection of the science-collection and are made available using the form:

```
(require (planet "gaussian-tail.ss" ("williams" "science.plt" 2 0)
                 "random-distributions"))
```

7.9.1 Random Variates from the Gaussian Tail Distribution

`random-gaussian-tail`

Function: `(random-gaussian-tail s a mu sigma)`
Function: `(random-gaussian-tail a mu sigma`
Contract:
```
(case->
   (-> random-source? real? real? (>/c 0.0) real?)
   (-> real? real? (>/c 0.0) real?))
```

This function returns a random variate from the upper tail of the Gaussian distribution with mean *mu* and standard deviation *sigma* using the random source *s* or `(current-random-source)` if *s* is not provided. The value returned is larger than the lower limit *a*, which must be greater than the mean *mu*.

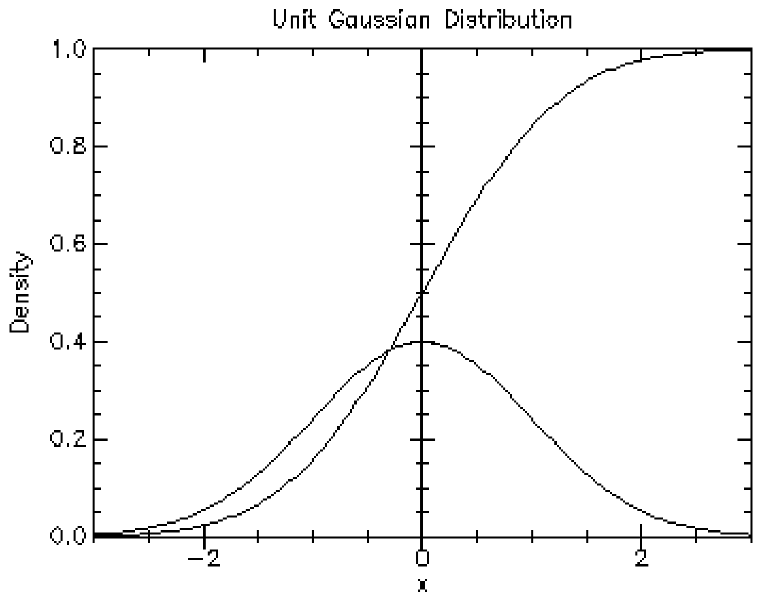

Figure 7.18: Plot of Probability Density for Unit Gaussian (Normal)

```
random-unit-gaussian-tail
```

Function: `(random-unit-gaussian-tail s a)`
Function: `(random-unit-gaussian-tail a`
Contract:
```
(case->
   (-> random-source? (>/c 0.0) (>/c 0.0))
   (-> (>/c 0.0)))
```

This function returns a random variate from the upper tail of the Gaussian distribution with mean 0 and standard deviation 1 using the random source *s* or `(current-random-source)` if *s* is not provided. The value returned is larger than the lower limit *a*, which must be positive.

Example: Histogram of random variates from the upper tail greater than 16.0 of the Gaussian distribution with mean 10.0 and standard deviation 2.0.

```
(require (planet "gaussian-tail.ss" ("williams" "science.plt" 2 0)
                 "random-distributions"))
(require (planet "histogram-with-graphics.ss"
                 ("williams" "science.plt" 2 0)))

(let ((h (make-histogram-with-ranges-uniform 40 16.0 22.0)))
  (do ((i 0 (+ i 1)))
      ((= i 10000) (void))
    (histogram-increment! h (random-gaussian-tail 16.0 10.0 2.0)))
  (histogram-plot h "Histogram of Gaussian Tail Distribution"))
```

Figure 7.19 shows the resulting histogram.

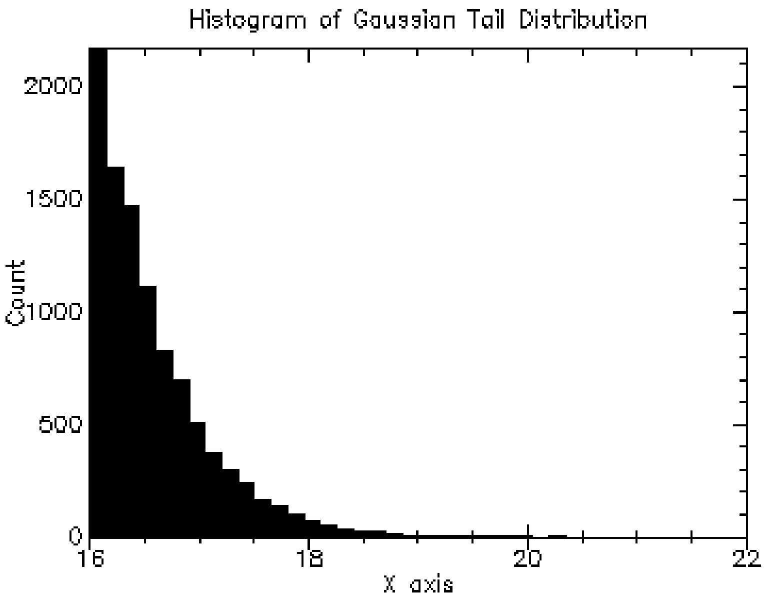

Figure 7.19: Histogram of Random Variates from Gaussian Tail (16.0, 10.0, 2.0)

Example: Histogram of random variates from the upper tail greater than 3.0 of the unit Gaussian distribution.

```
(require (planet "gaussian-tail.ss" ("williams" "science.plt" 2 0)
                 "random-distributions"))
(require (planet "histogram-with-graphics.ss"
                 ("williams" "science.plt" 2 0)))

(let ((h (make-histogram-with-ranges-uniform 40 3.0 6.0)))
  (do ((i 0 (+ i 1)))
      ((= i 10000) (void))
    (histogram-increment! h (random-unit-gaussian-tail 3.0)))
  (histogram-plot h "Histogram of Unit Gaussian Tail Distribution"))
```

Figure 7.20 shows the resulting histogram.

7.9.2 Gaussian Tail Distribution Density Functions

`gaussian-tail-pdf`

Function: `(gaussian-tail-pdf x a mu sigma)`
Contract: `(-> real? real? real? (>/c 0.0) (>=/c 0.0))`

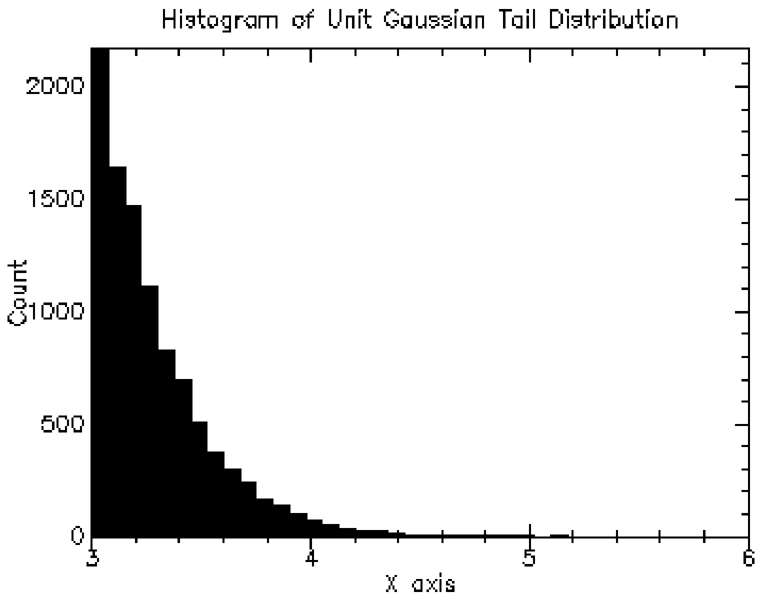

Figure 7.20: Histogram of Random Variates from Unit Gaussian Tail (3.0)

This function computes the probability density, $p(x)$, at x for the upper tail greater than a of the Gaussian distribution with mean *mu* and standard deviation *sigma*.

`unit-gaussian-tail-pdf`

Function: `(unit-gaussian-tail-pdf x a)`
Contract: `(-> real? (>=/c 0.0) (>=/c 0.0))`

This function computes the probability density, $p(x)$, at x for the upper tail greater than a of the Gaussian distribution with mean *mu* and standard deviation *sigma*.

7.9.3 Gaussian Tail Distribution Graphics

The Gaussian tail distribution graphics functions are defined in the file **`gaussian-tail-graphics.ss`** in the random-distributions sub-collection of the science collection and are made available using the form:

```
(require (planet "gaussian-tail-graphics.ss"
                 ("williams" "science.plt" 2 0)
                  "random-distributions"))
```

gaussian-tail-plot

Function: `(gaussian-tail-plot a mu sigma)`
Contract: `(-> real? real? (>/c 0.0) any)`

This function returns a plot of the probability density of the upper tail greater than *a* of the Gaussian distribution with mean *mu* and standard deviation *sigma*. The plot is produced by the plot collection provided with PLT Scheme (PLoT Scheme).

Example: Plot of probability density and cumulative density of the upper tail greater than 16.0 of the Gaussian distribution with mean 10.0 and standard deviation 2.0.

```
(require (planet "gaussian-tail-graphics.ss"
                 ("williams" "science.plt" 2 0)
                 "random-distributions"))
(gaussian-tail-plot 16.0 10.0 2.0)
```

Figure 7.21 shows the resulting plot.

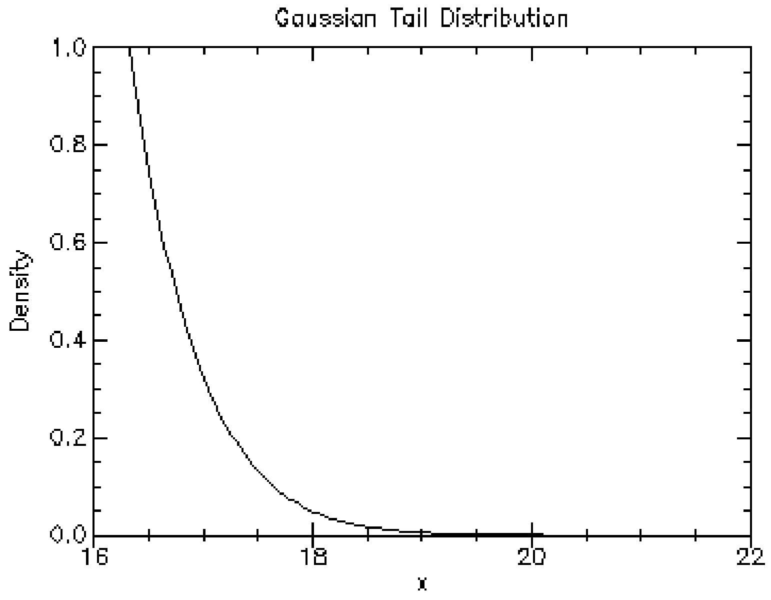

Figure 7.21: Plot of Probability Density for Gaussian Tail (16.0, 10.0, 2.0)

unit-gaussian-tail-plot

Function: `(unit-gaussian-tail-plot a)`
Contract: `(-> (>=/c 0.0) any)`

This function returns a plot of the probability density of the upper tail greater than a of the Gaussian distribution with mean 0.0 and standard deviation 1.0. The plot is produced by the plot collection provided with PLT Scheme (PLoT Scheme).

Example: Plot of probability density and cumulative density of the upper tail greater than 3.0 of the Gaussian distribution with mean 0.0 and standard deviation 1.0.

```
(require (planet "gaussian-tail-graphics.ss"
                 ("williams" "science.plt" 2 0)
                 "random-distributions"))
(unit-gaussian-tail-plot 3.0)
```

Figure 7.22 shows the resulting plot.

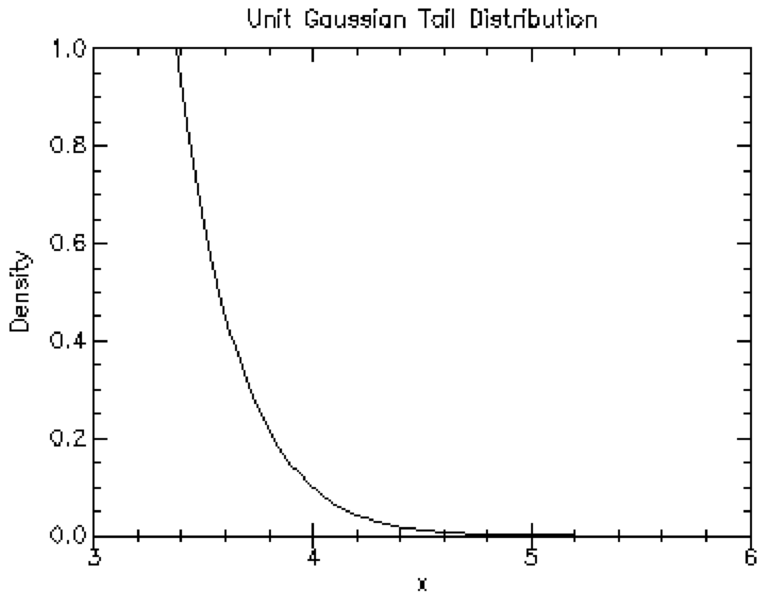

Figure 7.22: Plot of Probability Density for Unit Gaussian Tail (3.0)

7.10 The Log Normal Distribution

The log normal distribution functions are defined in the `lognormal.ss` file in the random-distributions sub-collection of the science-collection and are made available using the form:

```
(require (planet "lognormal.ss" ("williams" "science.plt" 2 0)
                 "random-distributions"))
```

7.10.1 Random Variates from the Log Normal Distribution

`random-lognormal`

Function: `(random-lognormal s mu sigma)`
Function: `(random-lognormal mu sigma)`
Contract:
```
(case->
   (-> random-source? real? (>=/c 0.0) real?)
   (-> real? real?))
```

This function returns a random variate from the log normal distribution with mean *mu* and standard deviation *sigma* using the random source *s* or `(current-random-source)` if *s* is not provided.

Example: Histogram of random variates from the log normal distribution with mean 0.0 and standard deviation 1.0.

```
(require (planet "lognormal.ss" ("williams" "science.plt" 2 0)
                 "random-distributions"))
(require (planet "histogram-with-graphics.ss"
                 ("williams" "science.plt" 2 0)))

(let ((h (make-histogram-with-ranges-uniform 40 0.0 6.0)))
  (do ((i 0 (+ i 1)))
      ((= i 10000) (void))
    (histogram-increment! h (random-lognormal 0.0 1.0)))
  (histogram-plot h "Histogram of Log Normal Distribution"))
```

Figure 7.23 shows the resulting histogram.

7.10.2 Log Normal Distribution Density Functions

`lognormal-pdf`

Function: `(lognormal-pdf x mu sigma)`
Contract: `(-> real? real? (>/c 0.0) (>=/c 0.0))`

This function computes the probability density, $p(x)$, at x for the log normal distribution with mean *mu* and standard deviation *sigma*.

`lognormal-cdf`

Function: `(lognormal-cdf x mu sigma)`
Contract: `(-> real? real? (>/c 0.0) (real-in 0.0 1.0))`

This function computes the cumulative density, $d(x)$, at x for the log normal distribution with mean *mu* and standard deviation *sigma*.

7.10.3 Log Normal Distribution Graphics

The log normal distribution graphics functions are defined in the file `lognormal-graphics.ss` in the random-distributions sub-collection of the science collection and are made available using the form:

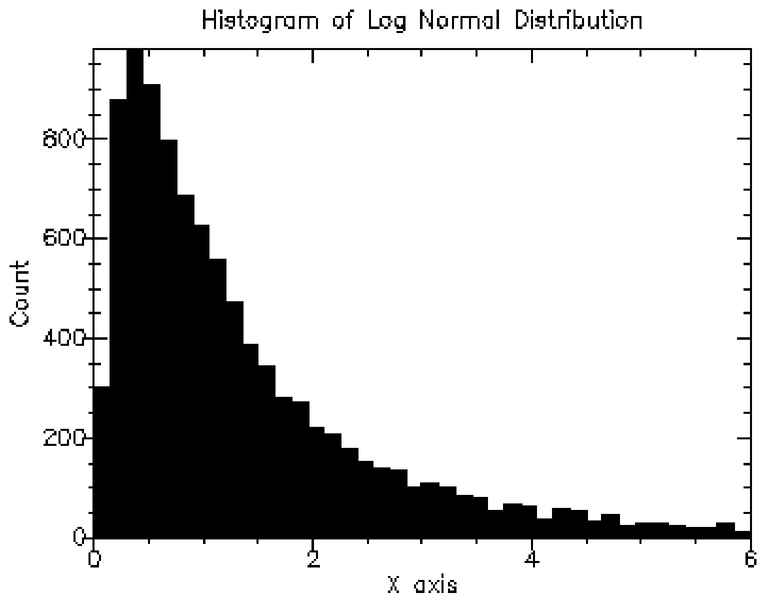

Figure 7.23: Histogram of Random Variates from Log Normal (0.0, 1.0)

```
(require (planet "lognormal-graphics.ss"
                 ("williams" "science.plt" 2 0)
                 "random-distributions"))
```

`lognormal-plot`

Function: `(lognormal-plot mu sigma)`
Contract: `(-> real? (>/c 0.0) any)`

This function returns a plot of the probability density and cumulative density of the log normal distribution with mean *mu* and standard deviation *sigma*. The plot is produced by the plot collection provided with PLT Scheme (PLoT Scheme).

Example: Plot of probability density and cumulative density of the log normal distribution with mean 0.0 and standard deviation 1.0.

```
(require (planet "lognormal-graphics.ss"
                 ("williams" "science.plt" 2 0)
                 "random-distributions"))
(lognormal-plot 0.0 1.0)
```

Figure 7.24 shows the resulting plot.

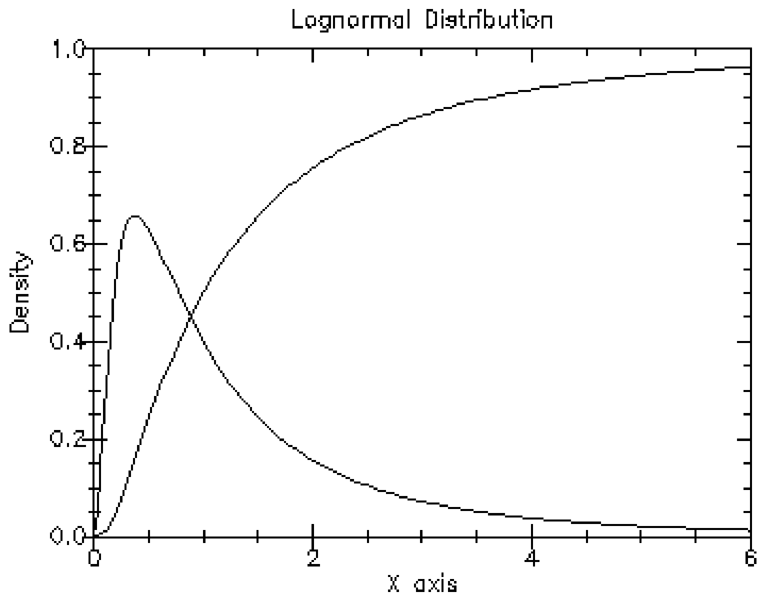

Figure 7.24: Plot of Probability Density for Log Normal (0.0, 1.0)

7.11 The Pareto Distribution

The Pareto distribution functions are defined in the `pareto.ss` file in the random-distributions sub-collection of the science-collection and are made available using the form:

```
(require (planet "pareto.ss" ("williams" "science.plt" 2 0)
                 "random-distributions"))
```

7.11.1 Random Variates from the Pareto Distribution

`random-pareto`

```
Function:   (random-pareto s a b)
Function:   (random-pareto a b)
Contract:   (case->
               (-> random-source? real? real? real?)
               (-> real? real? real?))
```

This function returns a random variate from the Pareto distribution with parameters a and b using the random source s or `(current-random-source)` if s is not provided.

Example: Histogram of random variates from the Pareto distribution with parameters 1.0 and 1.0.

```
(require (planet "pareto.ss" ("williams" "science.plt" 2 0)
                 "random-distributions"))
(require (planet "histogram-with-graphics.ss"
                 ("williams" "science.plt" 2 0)))

(let ((h (make-histogram-with-ranges-uniform 40 1.0 21.0)))
  (do ((i 0 (+ i 1)))
      ((= i 10000) (void))
    (histogram-increment! h (random-pareto 1.0 1.0)))
  (histogram-plot h "Histogram of Pareto Distribution"))
```

Figure 7.25 shows the resulting histogram.

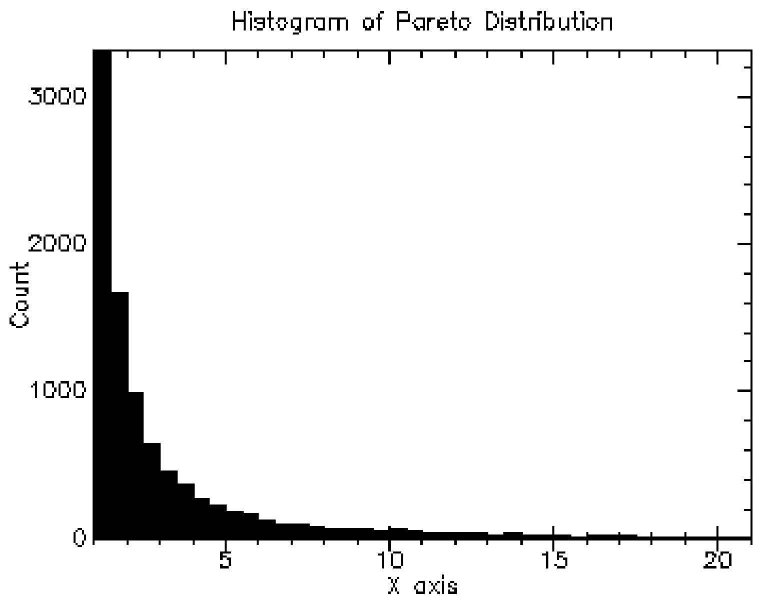

Figure 7.25: Histogram of Random Variates from Pareto (1.0, 1.0)

7.11.2 Pareto Distribution Density Functions

`pareto-pdf`

Function: `(pareto-pdf x a b)`
Contract: `(-> real? real? real? (>=/c 0.0))`

This function computes the probability density, $p(x)$, at x for the Pareto distribution with parameters a and b.

`pareto-cdf`

Function: `(pareto-cdf x a b)`
Contract: `(-> real? real? real? (real-in 0.0 1.0))`

This function computes the cumulative density, $d(x)$, at x for the Pareto distribution with parameters a and b.

7.11.3 Pareto Distribution Graphics

The Pareto distribution graphics functions are defined in the file `pareto-graphics.ss` in the random-distributions sub-collection of the science collection and are made available using the form:

```
(require (planet "pareto-graphics.ss" ("williams" "science.plt" 2 0)
                 "random-distributions"))
```

`pareto-plot`

Function: `(pareto-plot a b)`
Contract: `(-> real? real? any)`

This function returns a plot of the probability density and cumulative density of the Pareto distribution with parameters a and b. The plot is produced by the plot collection provided with PLT Scheme (PLoT Scheme).

Example: Plot of probability density and cumulative density of the Pareto distribution with parameters 1.0 and 1.0.

```
(require (planet "pareto-graphics.ss" ("williams" "science.plt" 2 0)
                 "random-distributions"))
(pareto-plot 1.0 1.0)
```

Figure 7.26 shows the resulting plot.

7.12 The T-Distribution

The t-distribution functions are defined in the `t-distribution.ss` file in the random-distributions sub-collection of the science-collection and are made available using the form:

```
(require (planet "t-distribution.ss" ("williams" "science.plt" 2 0)
                 "random-distributions"))
```

7.12.1 Random Variates from the T-Distribution

`random-t-distribution`

```
Function:   (random-t-distribution s nu)
Function:   (random-t-distribution nu)
Contract:   (case->
               (-> random-source? real? real?)
               (-> real? real?))
```

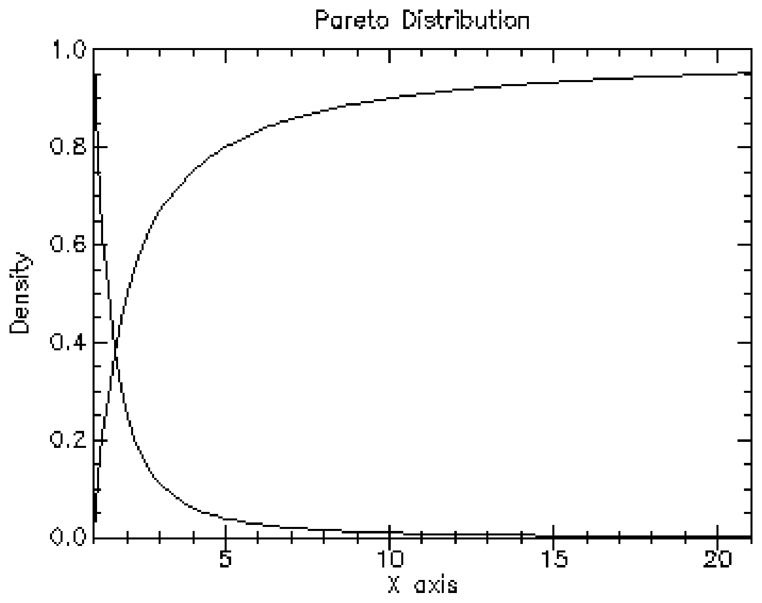

Figure 7.26: Plot of Probability Density for Pareto (1.0, 1.0)

This function returns a random variate from the t-distribution with *nu* degrees of freedom using the random source *s* or `(current-random-source)` if *s* is not provided.

Example: Histogram of random variates from the t-distribution with 1.0 degrees of freedom.

```
(require (planet "t-distribution.ss" ("williams" "science.plt" 2 0)
                 "random-distributions"))
(require (planet "histogram-with-graphics.ss"
                 ("williams" "science.plt" 2 0)))

(let ((h (make-histogram-with-ranges-uniform 40 -6.0 6.0)))
  (do ((i 0 (+ i 1)))
      ((= i 10000) (void))
    (histogram-increment! h (random-t-distribution 1.0)))
  (histogram-plot h "Histogram of t-Distribution"))
```

Figure 7.27 shows the resulting histogram.

7.12.2 T-Distribution Density Functions

`t-distribution-pdf`

Function: `(t-distribution-pdf x nu`
Contract: `(-> real? real? (>=/c 0.0))`

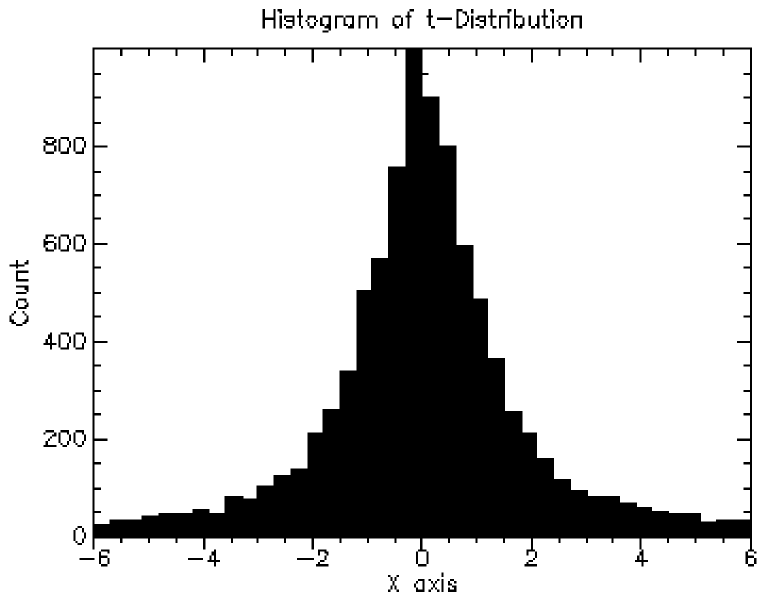

Figure 7.27: Histogram of Random Variates from T-Distribution (1.0)

This function computes the probability density, $p(x)$, at x for the t-distribution with *nu* degrees of freedom.

7.12.3 T-Distribution Graphics

The t-distribution graphics functions are defined in the file `t-distribution-graphics .ss` in the random-distributions sub-collection of the science collection and are made available using the form:

```
(require (planet "t-distribution-graphics.ss"
                 ("williams" "science.plt" 2 0)
                  "random-distributions"))
```

`t-distribution-plot`

Function: `(t-distribution-plot nu)`
Contract: `(-> real? any)`

This function returns a plot of the probability density of the t-distribution with *nu* degrees of freedom. The plot is produced by the plot collection provided with PLT Scheme (PLoT Scheme).

Example: Plot of probability density of the t-distribution with 1.0 degrees of freedom.

```
(require (planet "t-distribution-graphics.ss"
                 ("williams" "science.plt" 2 0)
                 "random-distributions"))
(t-distribution-plot 1.0)
```

Figure 7.28 shows the resulting plot.

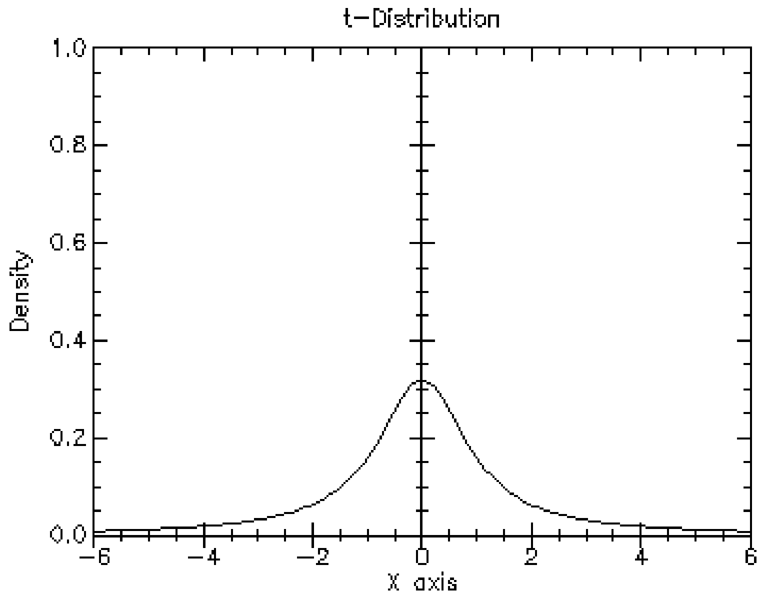

Figure 7.28: Plot of Probability Density for T-Distribution (1.0)

7.13 The Triangular Distribution

The triangular distribution functions are defined in the `triangular.ss` file in the random-distributions sub-collection of the science-collection and are made available using the form:

```
(require (planet "triangular.ss" ("williams" "science.plt" 2 0)
                 "random-distributions"))
```

7.13.1 Random Variates from the Triangular Distribution

`random-triangular`

```
Function:   (random-triangular s a b c)
Function:   (random-triangular a b c)
Contract:   (case->
               (->r ((s random-source?)
                     (a real?)
                     (b (>/c a))
                     (c (real-in a b)))
                    real?)
               (->r ((a real?)
                     (b (>/c a))
                     (c (real-in a b)))
                    real?))
```

This function returns a random variate from the triangular distribution with minimum value *a*, maximum value *b*, and most likely value *c* using the random source *s* or `(current-random-source)` if *s* is not provided.

Example: Histogram of random variates from the triangular distribution with minimum value 1.0, maximum value 4.0, and most likely value 2.0.

```
(require (planet "triangular.ss" ("williams" "science.plt" 2 0)
                 "random-distributions"))
(require (planet "histogram-with-graphics.ss"
                 ("williams" "science.plt" 2 0)))

(let ((h (make-histogram-with-ranges-uniform 40 1.0 4.0)))
  (do ((i 0 (+ i 1)))
      ((= i 10000) (void))
    (histogram-increment! h (random-triangular 1.0 4.0 2.0)))
  (histogram-plot h "Histogram of Triangular Distribution"))
```

Figure 7.29 shows the resulting histogram.

7.13.2 Triangular Distribution Density Functions

`triangular-pdf`

```
Function:   (triangular-pdf x a b c)
Contract:   (->r ((x real?)
                  (a real?)
                  (b (>/c a))
                  (c (real-in a b)))
                 (>=/c 0.0))
```

This function computes the probability density, $p(x)$, at x for the triangular distribution with minimum value a, maximum value b, and most likely value c.

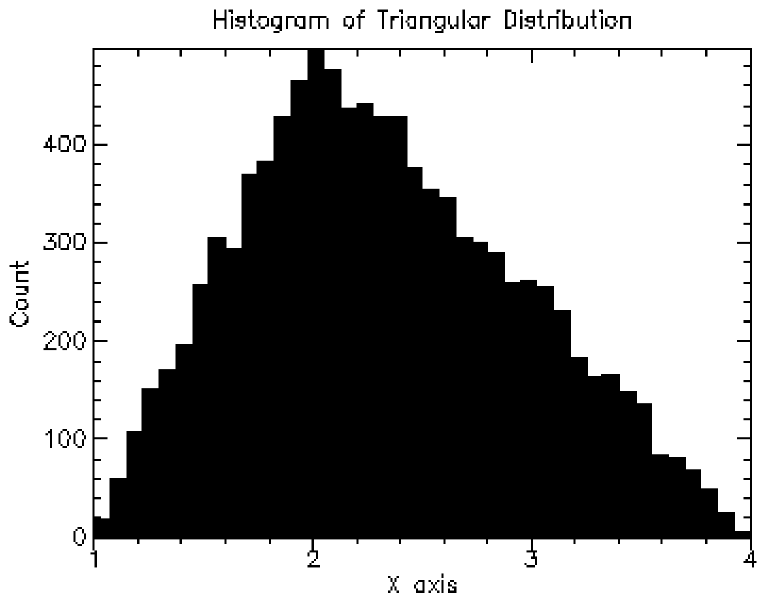

Figure 7.29: Histogram of Random Variates from Triangular (1.0, 4.0, 2.0)

`triangular-cdf`

Function: `(triangular-cdf x a b c)`
Contract:
```
(->r ((x real?)
      (a real?)
      (b (>/c a))
      (c (real-in a b)))
     (real-in 0.0 1.0))
```

This function computes the cumulative density, $d(x)$, at x for the triangular distribution with minimum value a, maximum value b, and most likely value c.

7.13.3 Triangular Distribution Graphics

The triangular distribution graphics functions are defined in the file **`triangular-graphics.ss`** in the random-distributions sub-collection of the science collection and are made available using the form:

```
(require (planet "triangular-graphics.ss"
                 ("williams" "science.plt" 2 0)
                 "random-distributions"))
```

`triangular-plot`

Function: `(triangular-plot a b c)`

Contract:
```
(->r ((a real?)
      (b (>/c a))
      (c (real-in a b)))
     any)
```

This function returns a plot of the probability density and cumulative density of the triangular distribution with minimum value *a*, maximum value *b*, and most likely value *c*. The plot is produced by the plot collection provided with PLT Scheme (PLoT Scheme).

Example: Plot of probability density of the triangular distribution with minimum value 1.0, maximum value 4.0, and most likely value 2.0.

```
(require (planet "triangular-graphics.ss"
                 ("williams" "science.plt" 2 0)
                 "random-distributions"))
(triangular-plot 1.0 4.0 2.0)
```

Figure 7.30 shows the resulting plot.

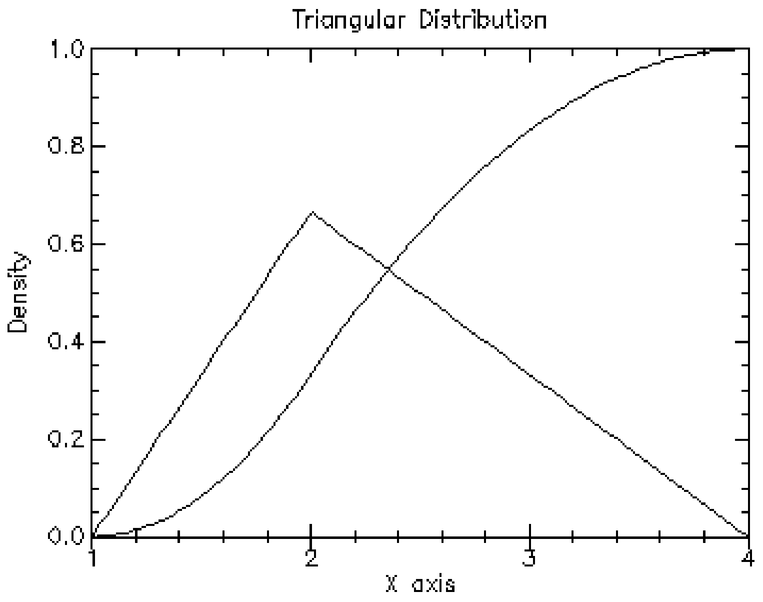

Figure 7.30: Plot of Probability Density for Triangular (1.0, 4.0, 2.0)

7.14 The Bernoulli Distribution

The Bernoulli distribution functions are defined in the `bernoulli.ss` file in the random-distributions sub-collection of the science-collection and are made available using the form:

```
(require (planet "bernoulli.ss" ("williams" "science.plt" 2 0)
                 "random-distributions"))
```

7.14.1 Random Variates from the Bernoulli Distribution

`random-bernoulli`

Function: `(random-bernoulli s p)`
Function: `(random-bernoulli p)`
Contract:
```
(case->
   (-> random-source? (real-in 0.0 1.0) (integer-in 0 1))
   (-> (real-in 0.0 1.0) (integer-in 0 1)))
```

This function returns a random variate from the Bernoulli distribution with probability p using the random source s or `(current-random-source)` if s is not provided.

Example: Histogram of random variates from the Bernoulli distribution with probability 0.6.

```
(require (planet "bernoulli.ss" ("williams" "science.plt" 2 0)
                 "random-distributions"))
(require (planet "discrete-histogram-with-graphics.ss"
                 ("williams" "science.plt" 2 0)))

(let ((h (make-discrete-histogram)))
  (do ((i 0 (+ i 1)))
      ((= i 10000) (void))
    (discrete-histogram-increment! h (random-bernoulli 0.6)))
  (discrete-histogram-plot h "Histogram of Bernoulli Distribution"))
```

Figure 7.31 shows the resulting histogram.

7.14.2 Bernoulli Distribution Density Functions

`bernoulli-pdf`

Function: `(bernoulli-pdf k p)`
Contract: `(-> integer? (real-in 0.0 1.0) (>=/c 0.0))`

This function computes the probability density, $p(k)$, at k for the Bernoulli distribution with probability p.

`bernoulli-cdf`

Function: `(bernoulli-cdf k p)`
Contract: `(-> integer? (real-in 0.0 1.0) (real-in 0.0 1.0))`

This function computes the cumulative density, $d(k)$, at k for the Bernoulli distribution with probability p.

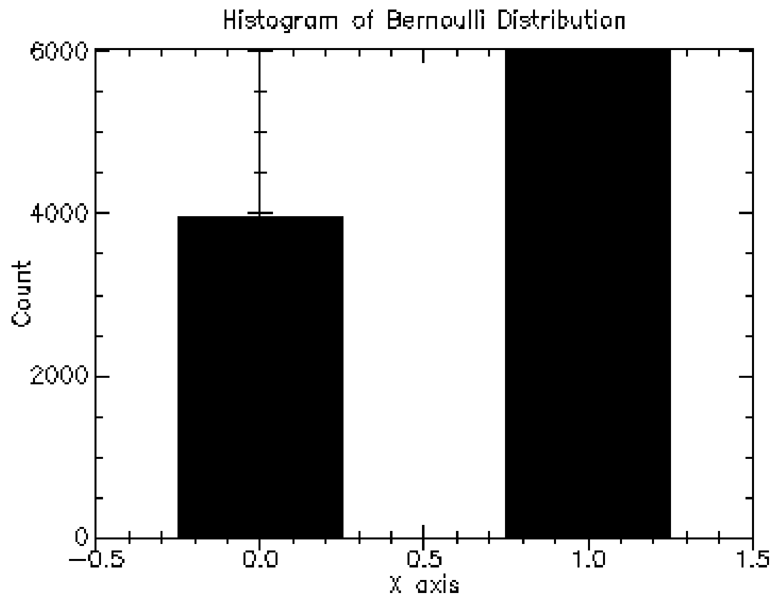

Figure 7.31: Histogram of Random Variates from Bernoulli (0.6)

7.14.3 Bernoulli Distribution Graphics

The Bernoulli distribution graphics functions are defined in the file **`bernoulli-graphics.ss`** in the random-distributions sub-collection of the science collection and are made available using the form:

```
(require (planet "bernoulli-graphics.ss"
                 ("williams" "science.plt" 2 0)
                 "random-distributions"))
```

`bernoulli-plot`

Function: `(bernoulli-plot p)`
Contract: `(-> (real-in 0.0 1.0) any)`

This function returns a plot of the probability density of the Bernoulli distribution with probability *p*. The plot is produced by the plot collection provided with PLT Scheme (PLoT Scheme).

Example: Plot of probability density of the Berboulli distribution with probability 0.6.

```
(require (planet "bernoulli-graphics.ss"
                 ("williams" "science.plt" 2 0)
```

```
                  "random-distributions"))
(bernoulli-plot 0.6)
```

Figure 7.32 shows the resulting plot.

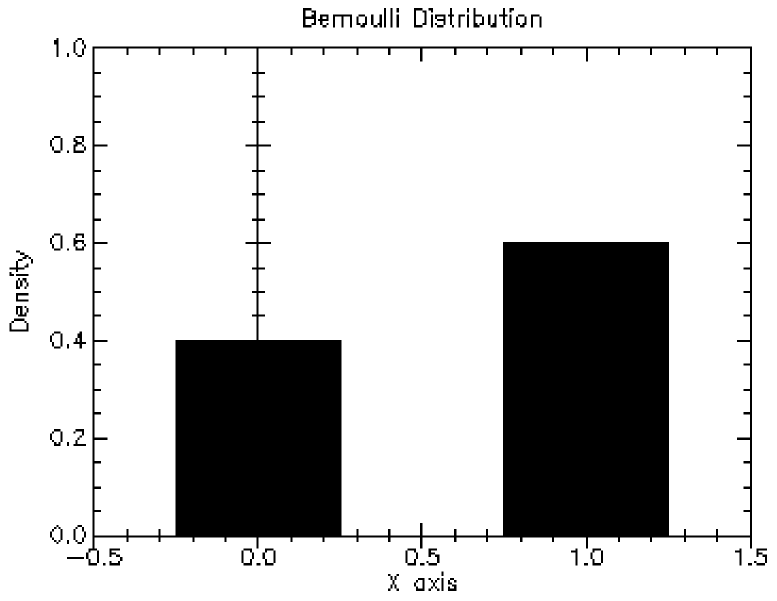

Figure 7.32: Plot of Probability Density for Bernoulli (0.6)

7.15 The Binomial Distribution

The binomial distribution functions are defined in the `binomial.ss` file in the random-distributions sub-collection of the science-collection and are made available using the form:

```
(require (planet "binomial.ss" ("williams" "science.plt" 2 0)
                  "random-distributions"))
```

7.15.1 Random Variates from the Binomial Distribution

`random-binomial`

```
Function:   (random-binomial s p n)
Function:   (random-binomial p n)
Contract:   (case->
               (-> random-source? (real-in 0.0 1.0) natural-number?
                    natural-number?)
               (-> (real-in 0.0 1.0) natural-number? natural-number?))
```

This function returns a random variate from the binomial distribution with parameters p and n using the random source s or `(current-random-source)` if s is not provided.

Example: Histogram of random variates from the binomial distribution with parameters 0.5 and 20.

```
(require (planet "binomial.ss" ("williams" "science.plt" 2 0)
                 "random-distributions"))
(require (planet "discrete-histogram-with-graphics.ss"
                 ("williams" "science.plt" 2 0)))

(let ((h (make-discrete-histogram)))
  (do ((i 0 (+ i 1)))
      ((= i 10000) (void))
    (discrete-histogram-increment! h (random-binomial 0.5 20)))
  (discrete-histogram-plot h "Histogram of Binomial Distribution"))
```

Figure 7.33 shows the resulting histogram.

7.15.2 Binomial Distribution Density Functions

`binomial-pdf`

```
Function:   ((binomial-pdf k p n)
Contract:   (-> integer? (real-in 0.0 1.0) natural-number? (>=/c 0.0))
```

This function computes the probability density, $p(k)$, at k for the binomial distribution with parameters p and n.

7.15.3 Binomial Distribution Graphics

The binomial distribution graphics functions are defined in the file `binomial-graphics-.ss` in the random-distributions sub-collection of the science collection and are made available using the form:

```
(require (planet "binomial-graphics.ss" ("williams" "science.plt" 2 0)
                 "random-distributions"))
```

`binomial-plot`

```
Function:   (binomial-plot p n)
Contract:   (-> (real-in 0.0 1.0) natural-number? any)
```

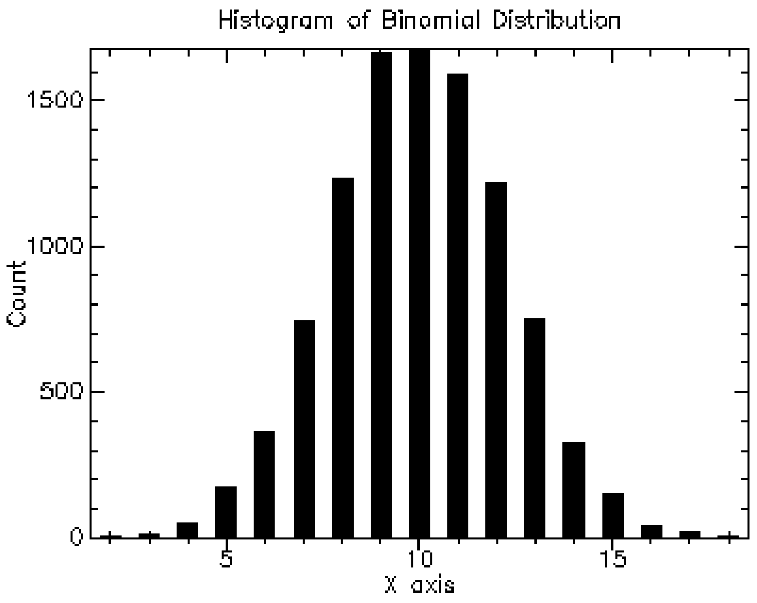

Figure 7.33: Histogram of Random Variates from Binomial (0.5, 20)

This function returns a plot of the probability density of the binomial distribution with parameters p and n. The plot is produced by the plot collection provided with PLT Scheme (PLoT Scheme).

Example: Plot of probability density of the binomial distribution with parameters 0.5 and 20.

```
(require (planet "binomial-graphics.ss" ("williams" "science.plt" 2 0)
                 "random-distributions"))
(binomial-plot 0.5 20)
```

Figure 7.34 shows the resulting plot.

7.16 The Geometric Distribution

The geometric distribution functions are defined in the `geometric.ss` file in the random-distributions sub-collection of the science-collection and are made available using the form:

```
(require (planet "geometric.ss" ("williams" "science.plt" 2 0)
                 "random-distributions"))
```

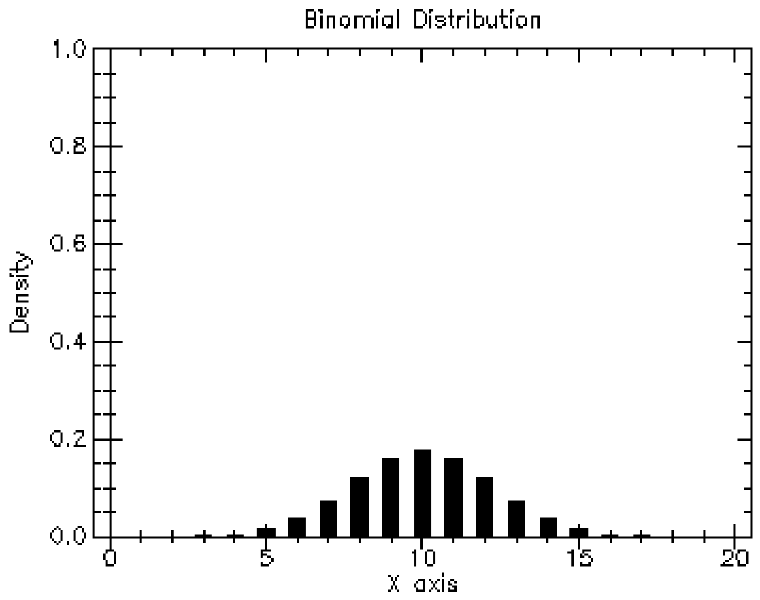

Figure 7.34: Plot of Probability Density for Binomial (0.5 20)

7.16.1 Random Variates from the Geometric Distribution

`random-geometric`

Function: `(random-geometric s p)`
Function: `(random-geometric p)`
Contract:
```
(case->
   (-> random-source? (real-in 0.0 1.0) natural-number?)
   (-> (real-in 0.0 1.0) natural-number?))
```

This function returns a random variate from the geometric distribution with probability p using the random source s or `(current-random-source)` if s is not provided.

Example: Histogram of random variates from the geometric distribution with probability 0.5.

```
(require (planet "geometric.ss" ("williams" "science.plt" 2 0)
                 "random-distributions"))
(require (planet "discrete-histogram-with-graphics.ss"
                 ("williams" "science.plt" 2 0)))

(let ((h (make-discrete-histogram)))
  (do ((i 0 (+ i 1)))
      ((= i 10000) (void))
    (discrete-histogram-increment! h (random-geometric 0.5)))
  (discrete-histogram-plot h "Histogram of Geometric Distribution"))
```

Figure 7.35 shows the resulting histogram.

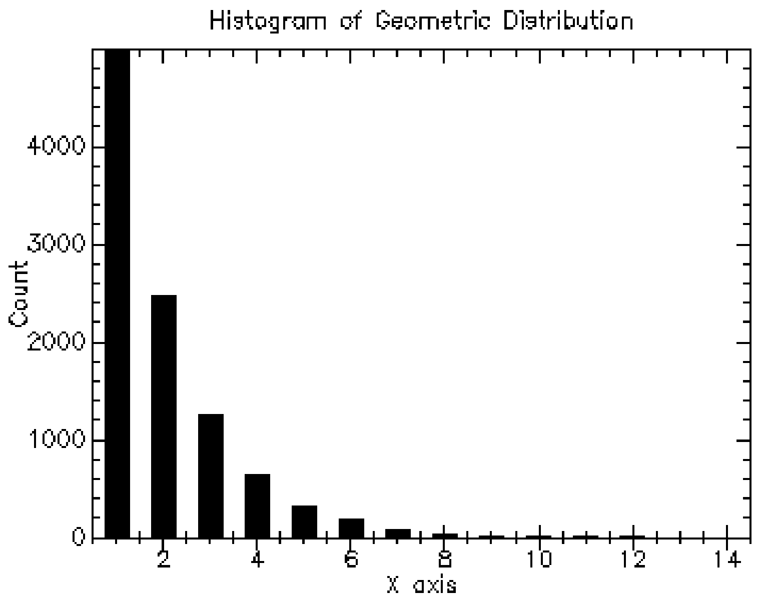

Figure 7.35: Histogram of Random Variates from Geometric (0.5)

7.16.2 Geometric Distribution Density Functions

`geometric-pdf`

Function: `(geometric-pdf k p)`
Contract: `(-> integer? (real-in 0.0 1.0) (>=/c 0.0))`

This function computes the probability density, $p(k)$, at k for the geometric distribution with probability p.

7.16.3 Geometric Distribution Graphics

The geometric distribution graphics functions are defined in the file `geometric-graphics.ss` in the random-distributions sub-collection of the science collection and are made available using the form:

```
(require (planet "geometric-graphics.ss" ("williams" "science.plt" 2 0)
                 "random-distributions"))
```

`geometric-plot`

Function: `(geometric-plot p)`
Contract: `(-> (real-in 0.0 1.0) any)`

This function returns a plot of the probability density of the geometric distribution with probability *p*. The plot is produced by the plot collection provided with PLT Scheme (PLoT Scheme).

Example: Plot of probability density of the geometric distribution with probability 0.5.

```
(require (planet "geometric-graphics.ss" ("williams" "science.plt" 2 0)
                 "random-distributions"))
(binomial-plot 0.5)
```

Figure 7.36 shows the resulting plot.

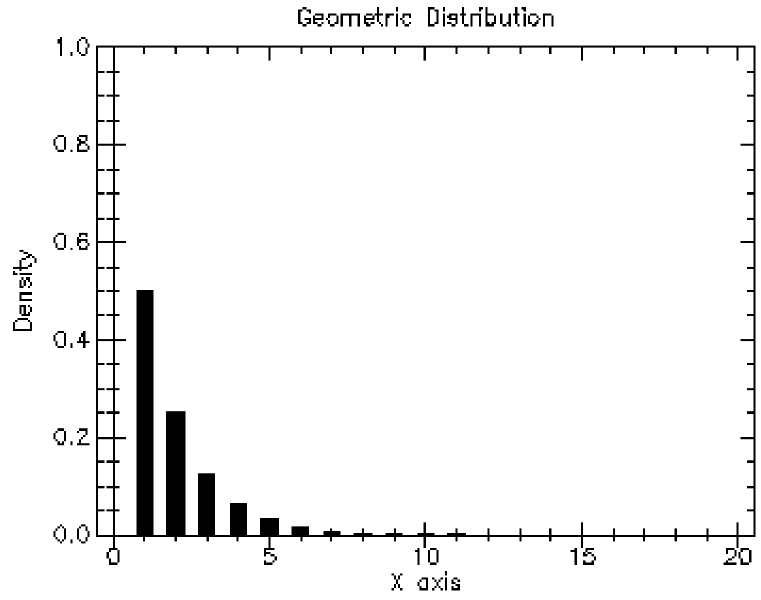

Figure 7.36: Plot of Probability Density for Geometric (0.5)

7.17 The Logarithmic Distribution

The logarithmic distribution functions are defined in the `logarithmic.ss` file in the random-distributions sub-collection of the science-collection and are made available using the form:

```
(require (planet "logarithmic.ss" ("williams" "science.plt" 2 0)
                 "random-distributions"))
```

7.17.1 Random Variates from the Logarithmic Distribution

random-logarithmic

Function: `(random-logarithmic s p)`
Function: `(random-logarithmic p)`
Contract:
```
(case->
   (-> random-source? (real-in 0.0 1.0) natural-number?)
   (-> (real-in 0.0 1.0) natural-number?))
```

This function returns a random variate from the logarithmic distribution with probabiliity p using the random source s or `(current-random-source)` if s is not provided.

Example: Histogram of random variates from the logarithmic distribution with probability 0.5.

```
(require (planet "logarithmic.ss" ("williams" "science.plt" 2 0)
                 "random-distributions"))
(require (planet "discrete-histogram-with-graphics.ss"
                 ("williams" "science.plt" 2 0)))

(let ((h (make-discrete-histogram)))
  (do ((i 0 (+ i 1)))
      ((= i 10000) (void))
    (discrete-histogram-increment! h (random-logarithmic 0.5)))
  (discrete-histogram-plot h "Histogram of Logarithmic Distribution"))
```

Figure 7.37 shows the resulting histogram.

7.17.2 Logarithmic Distribution Density Functions

logarithmic-pdf

Function: `(logarithmic-pdf k p)`
Contract: `(-> integer? (real-in 0.0 1.0) (>=/c 0.0))`

This function computes the probability density, $p(k)$, at k for the logarithmic distribution with probability p.

7.17.3 Logarithmic Distribution Graphics

The logarithmic distribution graphics functions are defined in the file `logarithmic-graphics.ss` in the random-distributions sub-collection of the science collection and are made available using the form:

```
(require (planet "logarithmic-graphics.ss"
                 ("williams" "science.plt" 2 0)
                 "random-distributions"))
```

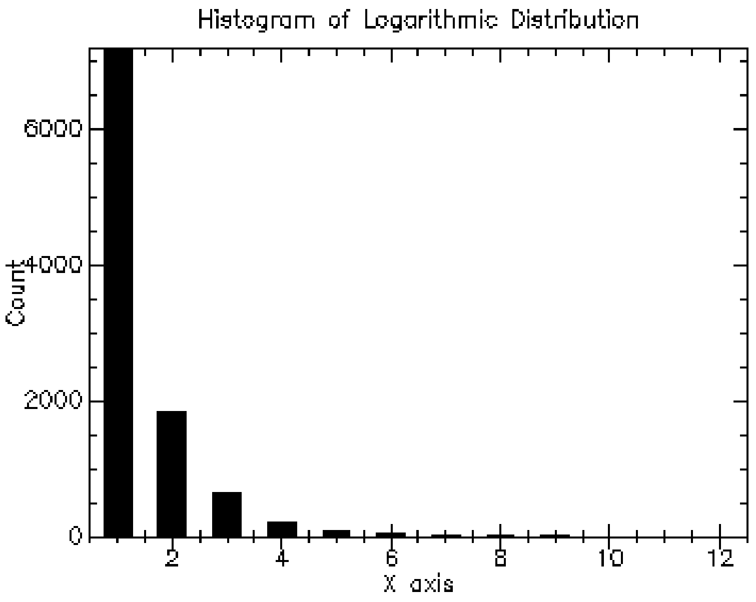

Figure 7.37: Histogram of Random Variates from Logarithmic (0.5)

`logarithmic-plot`

Function: `(logarithmic-plot p)`
Contract: `(-> (real-in 0.0 1.0) any)`

This function returns a plot of the probability density of the logarithmic distribution with probability *p*. The plot is produced by the plot collection provided with PLT Scheme (PLoT Scheme).

Example: Plot of probability density of the logarithmic distribution with probability 0.5.

```
(require (planet "logarithmic-graphics.ss"
                 ("williams" "science.plt" 2 0)
                 "random-distributions"))
(logarithmic-plot 0.5)
```

Figure 7.38 shows the resulting plot.

7.18 The Poisson Distribution

The Poisson distribution functions are defined in the `poisson.ss` file in the random-distributions sub-collection of the science-collection and are made available using the form:

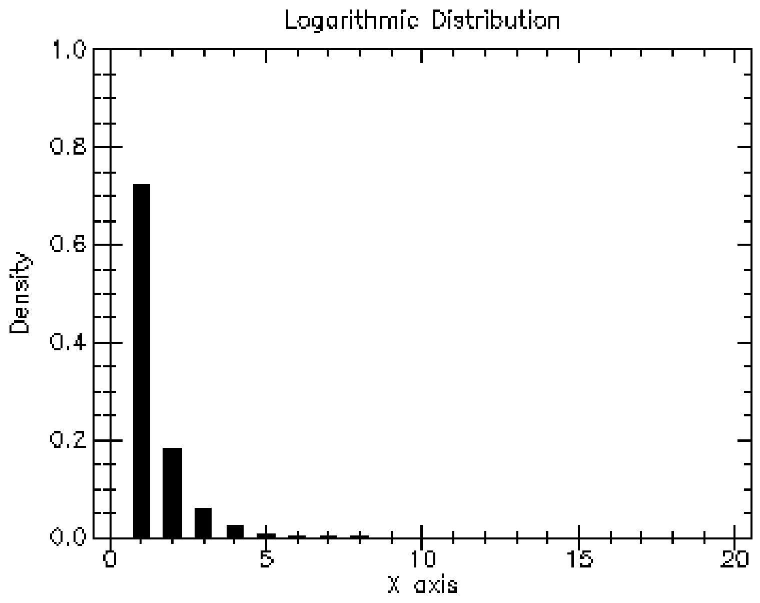

Figure 7.38: Plot of Probability Density for Logarithmic (0.5)

```
(require (planet "poisson.ss" ("williams" "science.plt" 2 0)
                 "random-distributions"))
```

7.18.1 Random Variates from the Poisson Distribution

`random-poisson`

Function: `(random-poisson s mu)`
Function: `(random-poisson mu)`
Contract:
```
(case->
   (-> random-source? real? natural-number?)
   (-> real? natural-number?))
```

This function returns a random variate from the Poisson distribution with mean *mu* using the random source *s* or `(current-random-source)` if *s* is not provided.

Example: Histogram of random variates from the Poisson distribution with mean 10.0.

```
(require (planet "poisson.ss" ("williams" "science.plt" 2 0)
                 "random-distributions"))
(require (planet "discrete-histogram-with-graphics.ss"
                 ("williams" "science.plt" 2 0)))
```

```
(let ((h (make-discrete-histogram)))
  (do ((i 0 (+ i 1)))
      ((= i 10000) (void))
    (discrete-histogram-increment! h (random-poisson 10.0)))
  (discrete-histogram-plot h "Histogram of Poisson Distribution"))
```

Figure 7.39 shows the resulting histogram.

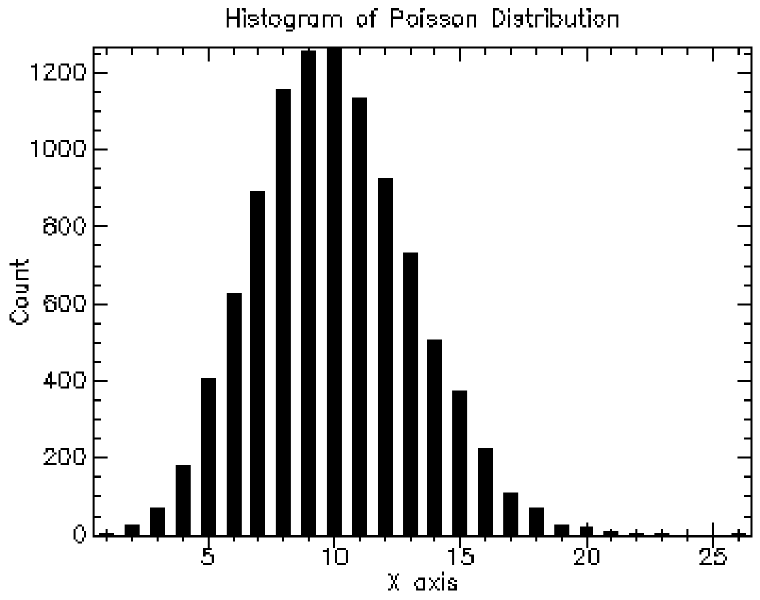

Figure 7.39: Histogram of Random Variates from Poisson (10.0)

7.18.2 Poisson Distribution Density Functions

`poisson-pdf`

Function: `(poisson-pdf k mu)`
Contract: `(-> integer? real? (>=/c 0.0))`

This function computes the probability density, $p(k)$, at k for the Poisson distribution with mean *mu*.

7.18.3 Poisson Distribution Graphics

The Poisson distribution graphics functions are defined in the file `poisson-graphics .ss` in the random-distributions sub-collection of the science collection and are made available using the form:

```
(require (planet "poisson-graphics.ss" ("williams" "science.plt" 2 0)
                 "random-distributions"))
```

`poisson-plot`

Function: `(poisson-plot mu)`
Contract: `(-> (>/c 0.0) any)`

This function returns a plot of the probability density of the Poisson distribution with mean *mu*. The plot is produced by the plot collection provided with PLT Scheme (PLoT Scheme).

Example: Plot of probability density of the Poisson distribution with mean 10.0.

```
(require (planet "poisson-graphics.ss" ("williams" "science.plt" 2 0)
                 "random-distributions"))
(poisson-plot 10.0)
```

Figure 7.40 shows the resulting plot.

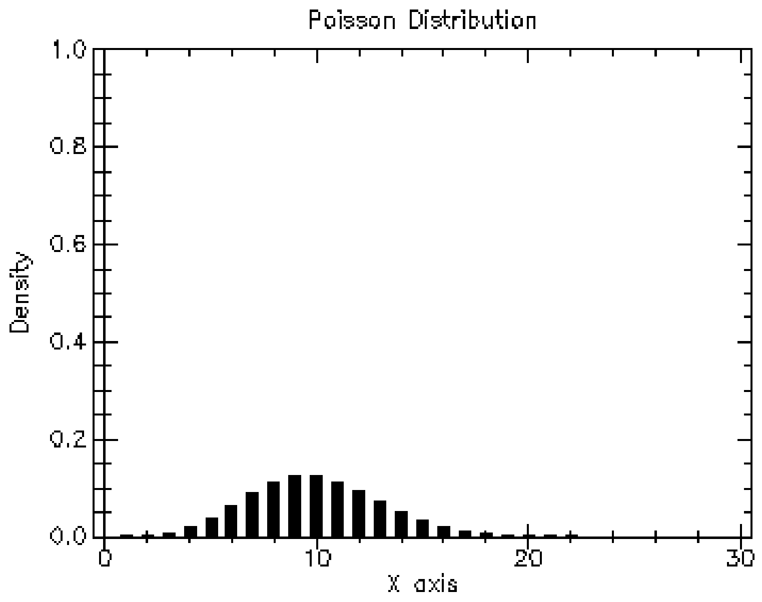

Figure 7.40: Plot of Probability Density for Poisson (10.0)

7.19 The General Discrete Distribution

The discrete distribution functions are defined in the `discrete.ss` file in the random-distributions sub-collection of the science-collection and are made available using the form:

```
(require (planet "discrete.ss" ("williams" "science.plt" 2 0)
                 "random-distributions"))
```

`discrete?`

Function: `(discrete? x)`
Contract: `(-> any? boolean?)`

This function returns true, `#t`, if *x* is a discrete distribution and false, `#f`, otherwise.

7.19.1 Creating Discrete Distributions

`make-discrete`

Function: `(make-discrete weights)`
Contract: `(-> (vectorof real?) discrete?)`

This function returns a discrete distribution whose probability density is given by the specified *weights*. Note that the *weights* do not have to sum to one.

7.19.2 Random Variates from a Discrete Distribution

`random-discrete`

Function: `(random-discrete s d)`
Function: `(random-discrete d)`
Contract:
```
(case->
   (-> random-source? discrete?)
   (-> discrete?))
```

This function returns a random variate from the discrete distribution *d* using the random source *s* or `(current-random-source)` if *s* is not provided.

Example: Histogram of random variates from a discrete distribution with weights #(.1 .4 .9 .8 .7 .6 .5 .4 .3 .2 .1).

```
(require (planet "discrete.ss" ("williams" "science.plt" 2 0)
                 "random-distributions"))
(require (planet "discrete-histogram-with-graphics.ss"
                 ("williams" "science.plt" 2 0)))

(let ((h (make-discrete-histogram))
      (d (make-discrete #(.1 .4 .9 .8 .7 .6 .5 .4 .3 .2 .1))))
  (do ((i 0 (+ i 1)))
      ((= i 10000) (void))
    (discrete-histogram-increment! h (random-discrete d)))
  (discrete-histogram-plot h "Histogram of Discrete Distribution"))
```

Figure 7.41 shows the resulting histogram.

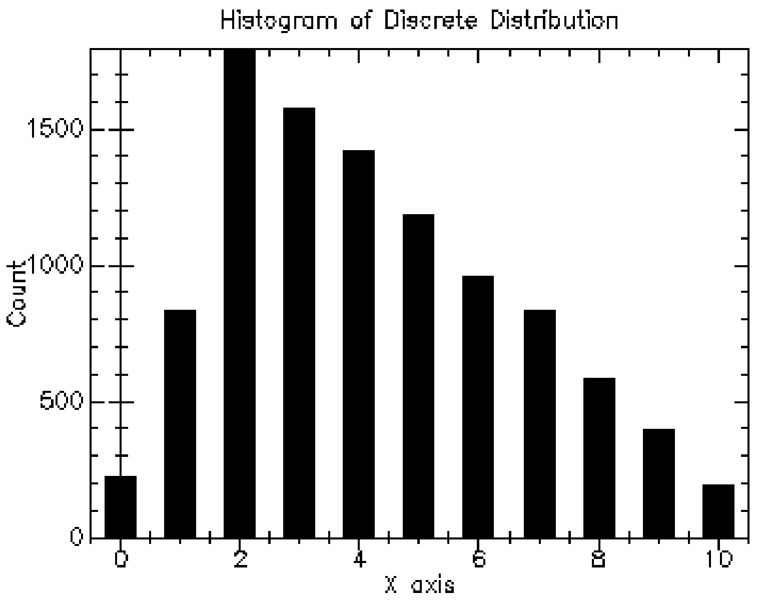

Figure 7.41: Histogram of Random Variates from a Discrete Distribution

7.19.3 Discrete Distribution Density Functions

`discrete-pdf`

<u>Function</u>: `(discrete-pdf d k)`
<u>Contract</u>: `(-> discrete? integer? (real-in 0.0 1.0))`

This function returns the probability density, $p(k)$, at k for the discrete distribution d.

`discrete-cdf`

<u>Function</u>: `(discrete-cdf d k)`
<u>Contract</u>: `(-> discrete? integer? (real-in 0.0 1.0)`

This function returns the cumulative density, $d(k)$, at k for the discrete distribution d.

7.19.4 General Discrete Distribution Graphics

The general discrete distribution graphics functions are defined in the file `discrete-graphics.ss` in the random-distributions sub-collection of the science collection and are made available using the form:

```
(require (planet "discrete-graphics.ss" ("williams" "science.plt" 2 0)
                 "random-distributions"))
```

`discrete-plot`

Function: `(discrete-plot d)`
Contract: `(-> discrete? any)`

This function returns a plot of the probability density of the general distribution *d*. The plot is produced by the plot collection provided with PLT Scheme (PLoT Scheme).

Example: Plot of probability density of the Poisson distribution with mean 10.0.

```
(require (planet "discrete.ss" ("williams" "science.plt" 2 0)
                 "random-distributions"))
(require (planet "discrete-graphics.ss" ("williams" "science.plt" 2 0)
                 "random-distributions"))

(let ((d (make-discrete #(.1 .4 .9 .8 .7 .6 .5 .4 .3 .2 .1))))
  (discrete-plot d))
```

Figure 7.42 shows the resulting plot.

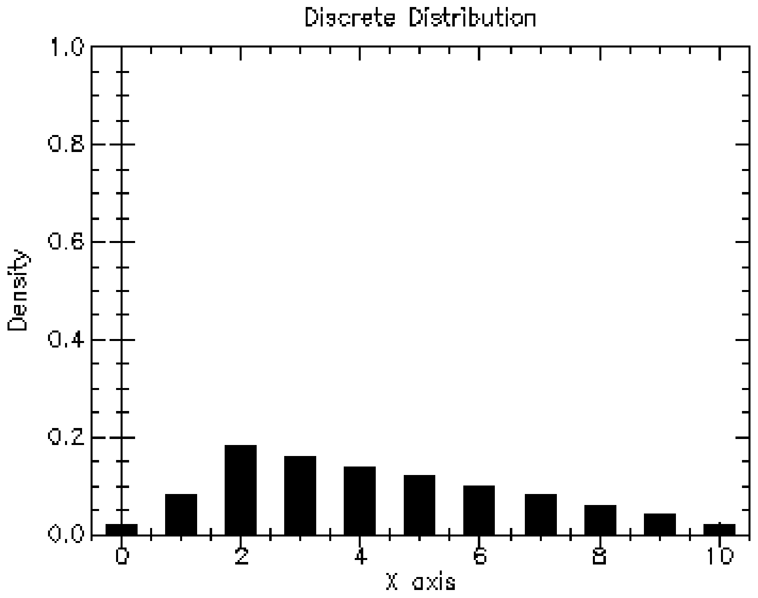

Figure 7.42: Plot of Probability Density for a General Discrete Distribution

Chapter 8

Statistics

This chapter describes the statistical functions provided by the PLT Scheme Science Collection. The basic statistical functions include functions to compute the mean, variance, and standard deviation. More advanced functions allow you to calculate absolute deviation, skewness, and kurtosis, as well as the median and arbitrary percentiles. The algorithms use recurrance relations to compute average quantities in a stable way, without large intermediate values that might overflow.

The functions described in this chapter are defined in the `statistics.ss` file in the science collection and are made available ising the following form:

```
(require (planet "statistics.ss" ("williams" "science.plt" 2 0)))
```

8.1 Mean, Standard Deviation, and Variance

`mean`

Function: `(mean data)`
Contract: `(-> (vectorof real?) real?)`

This function returns the arithmetic mean of *data*.

`variance`

```
Function:  (variance data mu)
Function:  (variance data)
Contract:  (case->
              (-> (vector-of real?) real? (>=/c 0.0))
              (-> (vector-of real?) (>=/c 0.0)))
```

This function returns the sample variance of *data*. If *mu* is not provided, it is calculated by a call to `(mean data)`.

`standard-deviation`

Function: `(standard-deviation data mu)`
Function: `(standard-deviation data)`
Contract:
```
(case->
   (-> (vector-of real?) real? (>=/c 0.0))
   (-> (vector-of real?) (>=/c 0.0)))
```

The *standard deviation* is defined as the square root of the variance. This function returns the square root of the corresponding `variance` function above.

`variance-with-fixed-mean`

Function: `(variance-with-fixed-mean data mu)`
Contract: `(-> (vector-of real?) real? (>=/c 0.0))`

This function returns an unbiased estimate of the variance of *data* when the population mean, *mu*, of the underlying distribution is known *a priori*.

`standard-deviation-with-fixed-mean`

Function: `(standard-deviation-with-fixed-mean data mu)`
Contract: `(-> (vector-of real?) real> (>=/c 0.0))`

This function returns the standard deviation of *data* with a fixed population mean, *mu*. The result is the square root of the `variance-with-fixed-mean` function.

8.2 Absolute Deviation

`absolute-deviation`

Function: `(absolute-deviation data mu)`
Function: `(absolute-deviation data)`
Contract:
```
(case->
   (-> (vector-of real?) real? (>=/c 0.0))
   (-> (vector-of real?) (>=/c 0.0)))
```

This function returns the absolute deviation of *data* relative to the given value of the mean, *mu*. If *mu* is not provided, it is calculated by a call to `(mean data)`. This function is also useful if you want to calculate the absolute deviation relative to any value other than the mean, such as zero or the median.

8.3 Higher Moments (Skewness and Kurtosis)

`skew`

Function: `(skew data mu sigma)`
Function: `(skew data)`
Contract:
```
(case->
   (-> (vector-of real?) real? (>=/c 0.0) real?)
   (-> (vector-of real?) real?)))
```

The *shewness* measures the symmetry of the tails of a distribution. This function returns the skewness of *data* using the given values of the mean, *mu* and standard deviation, *sigma*. If *mu* and *sigma* are not provided, they are calculated by calls to `(mean data)` and `(standard-deviation data mu)`.

`kurtosis`

```
Function:   (kurtosis data mu sigma)
Function:   (kurtosis data)
Contract:   (case->
               (-> (vector-of real?) real? (>=/c 0.0) real?)
               (-> (vector-of real?) real?)))
```

The *kurtosis* measures how sharply peaked a distribution is relative to its width. This function returns the kurtosis of *data* using the given values of the mean, *mu* and standard deviation, *sigma*. If *mu* and *sigma* are not provided, they are calculated by calls to `(mean data)` and `(standard-deviation data mu)`.

8.4 Autocorrelation

`lag-1-autocorrelation`

```
Function:   (lag-1-autocorrelation data mu)
Function:   (lag-1-autocorrelation data)
Contract:   (case->
               (-> non-empty-vector-of-reals real? real?)
               (-> non-empty-vector-of-reals real?))
```

This function returns the lag-1 autocorrelation of *data* using the given value of the mean, *mu*. If *mu* is not provided, it is calculated by a call to `(mean data)`.

8.5 Covariance

`covariance`

Function: `(covariance data1 data 2 mu1 mu2)`
Function: `(covariance data1 data2)`
Contract:
```
(case->
   (->r ((data1 (vectorof real?))
         (data2 (and/c (vectorof real?)
                       (lambda (x)
                         (= (vector-length data1)
                            (vector-length data2)))))
         (mu1 real?)
         (mu2 real?))
        real?)
   (->r ((data1 (vectorof real?))
         (data2 (and/c (vectorof real?)
                       (lambda (x)
                         (= (vector-length data1)
                            (vector-length data2))))))
        real?))
```

This function returns the covariance of *data1* and *data2*, which must both be the same length, using the given values of the means, *mu1* and *mu2*. If the values of *mu1* and *mu2* are not given, they are calculated using calls to `(mean data1)` and `(mean data2)`, respectively.

`covariance-with-fixed-means`

Function: `(covariance-with-fixed-means data1 data2)`
Contract:
```
(->r ((data1 (vectorof real?))
      (data2 (and/c (vectorof real?)
                    (lambda (x)
                      (= (vector-length data1)
                         (vector-length data2)))))
      (mu1 real?)
      (mu2 real?))
     real?)
```

This function returns the covariance of *data1* and *data2*, which must both be the same length, when the population means, *mu1* and *mu2*, of the underlying distributions are known *a priori*.

8.6 Weighted Samples

`weighted-mean`

```
Function:   (weighted-mean w data)
Contract:   (->r ((w (vectorof real?))
                  (data (and/c (vectorof real?)
                               (lambda (x)
                                 (= (vector-length w)
                                    (vector-length data))))))
                 real?)
```

This function returns the weighted mean of *data* using weights *w*.

`weighted-variance`

```
Function:   (weighted-variance w data wmu)
Function:   (weighted-variance w data)
Contract:   (case->
               (->r ((w (vectorof real?))
                     (data (and/c (vectorof real?)
                                  (lambda (x)
                                    (= (vector-length w)
                                       (vector-length data)))))
                     (mu real?)
                    (>=/c 0.0))
               (->r ((w (vectorof real?))
                     (data (and/c (vectorof real?)
                                  (lambda (x)
                                    (= (vector-length w)
                                       (vector-length data)))))
                    (>=/c 0.0)))
```

This function returns the weighted variance of *data* using weights *w* and the given weighted mean, *wmu*. If *wmu* is not provided, it is calculated by a call to `(weighted-mean w data)`.

weighted-standard-deviation

Function: `(weighted-standard-deviation w data wmu)`
Function: `(weighted-standard-deviation w data)`
Contract:
```
(case->
   (->r ((w (vectorof real?))
         (data (and/c (vectorof real?)
                      (lambda (x)
                        (= (vector-length w)
                           (vector-length data)))))
         (mu real?)
        (>=/c 0.0))
   (->r ((w (vectorof real?))
         (data (and/c (vectorof real?)
                      (lambda (x)
                        (= (vector-length w)
                           (vector-length data)))))
        (>=/c 0.0)))
```

The *standard deviation* is defined as the square root of the variance. This function returns the square root of the corresponding `weighted-variance` function above.

weighted-variance-with-fixed-mean

Function: `(weighted-variance-with-fixed-mean w data wmu)`
Contract:
```
(->r ((w (vectorof real?))
      (data (and/c (vectorof real?)
                   (lambda (x)
                     (= (vector-length w)
                        (vector-length data)))))
      (mu real?)
     (>=/c 0.0))
```

This function returns an unbiased estimate of the weighted variance of *data* using weights *w* when the weighted population mean, *wmu*, of the underlying distribution is known *a priori*.

weighted-standard-deviation-with-fixed-mean

Function: `(weighted-standard-deviation-with-fixed-mean w data wmu)`
Contract:
```
(->r ((w (vectorof real?))
      (data (and/c (vectorof real?)
                   (lambda (x)
                     (= (vector-length w)
                        (vector-length data)))))
      (mu real?)
     (>=/c 0.0))
```

This function returns the weighted standard deviation of *data* using weights *w* with a fixed weighted population mean, *wmu*. The result is the square root of the `weighted-variance-with-fixed-mean` function.

weighted-absolute-deviation

```
Function:  (weighted-absolute-deviation w data wmu)
Function:  (weighted-absolute-deviation w data)
Contract:  (case->
              (->r ((w (vectorof real?))
                    (data (and/c (vectorof real?)
                                 (lambda (x)
                                   (= (vector-length w)
                                      (vector-length data)))))
                    (mu real?)
                   (>=/c 0.0))
              (->r ((w (vectorof real?))
                    (data (and/c (vectorof real?)
                                 (lambda (x)
                                   (= (vector-length w)
                                      (vector-length data)))))
                   (>=/c 0.0)))
```

This function returns the weighted absolute deviation of *data* using weights *s* relative to the given value of the weighted mean, *wmu*. If *wmu* is not provided, it is calculated by a call to `(weighted-mean w data)`. This function is also useful if you want to calculate the weighted absolute deviation relative to any value other than the weighted mean, such as zero or the weighted median.

weighted-skew

```
Function:  (weighted-skew w data wmu wsigma)
Function:  (weighted-skew w data)
Contract:  (case->
              (->r ((w (vectorof real?))
                    (data (and/c (vectorof real?)
                                 (lambda (x)
                                   (= (vector-length w)
                                      (vector-length data)))))
                    (mu real?)
                    (sigma (>=/c 0.0)))
                   real?)
              (->r ((w (vectorof real?))
                    (data (and/c (vectorof real?)
                                 (lambda (x)
                                   (= (vector-length w)
                                      (vector-length data)))))
                   real?))
```

The *shewness* measures the symmetry of the tails of a distribution. This function returns the weighted skewness of *data* using weights *w* and the given values of the weighted mean, *wmu* and weighted standard deviation, *wsigma*. If *wmu* and *wsigma* are not provided, they are calculated by calls to `(weighted-mean w data)` and `(weighted-standard-deviation w data wmu)`.

`weighted-kurtosis`

```
Function:   (weighted-kurtosis w data wmu wsigma)
Function:   (weighted-kurtosis w data)
Contract:   (case->
               (->r ((w (vectorof real?))
                     (data (and/c (vectorof real?)
                                  (lambda (x)
                                    (= (vector-length w)
                                       (vector-length data)))))
                     (mu real?)
                     (sigma (>=/c 0.0)))
                    real?)
               (->r ((w (vectorof real?))
                     (data (and/c (vectorof real?)
                                  (lambda (x)
                                    (= (vector-length w)
                                       (vector-length data)))))
                    real?))
```

The *kurtosis* measures how sharply peaked a distribution is relative to its width. This function returns the weighted kurtosis of *data* using weights *w* and the given values of the weighted mean, *wmu* and weighted standard deviation, *wsigma*. If *wmu* and *wsigma* are not provided, they are calculated by calls to `(weighted-mean w data)` and `(weighted-standard-deviation w data wmu)`.

8.7 Maximum and Minimum

`maximum`

```
Function:   (maximum data
Contract:   (-> non-empty-vector-of-reals? real?)
```

This function returns the maximum value in *data*.

`minimum`

```
Function:   (minimun data
Contract:   (-> non-empty-vector-of-reals? real?)
```

This function returns the minimum value in *data*.

`minimum-maximum`

```
Function:   (minimun-maximum data
Contract:   (-> non-empty-vector-of-reals? (values real? real?))
```

This function returns the minimum and maximum values in *data* as multiple values.

`maximum-index`

Function: `(maximum-index data`
Contract: `(-> non-empty-vector-of-reals? natural-number?)`

This function returns the index of the maximum value in *data*. When there are several equal maximum elements, the index of the first one is chosen.

`minimum-index`

Function: `(minimun-index data`
Contract: `(-> non-empty-vector-of-reals? natural-number?)`

This function returns the index of the minimum value in *data*. When there are several equal minimum elements, the index of the first one is chosen.

`minimum-maximum-index`

```
Function:   (minimun-maximum-index data
Contract:   (-> non-empty-vector-of-reals?
                (values natural-number? number?))
```

This function returns the indices of the minimum and maximum values in *data* as multiple values. When there are several equal minimum or maximum elements, the index of the first ones are chosen.

8.8 Median and Percentiles

The median and percentile functions described in this section operate on sorted data. The contracts for these functions enforce this. Also, for convenience we use quantiles measured on a scale of 0 to 1 instead of percentiles (which use a scale of 0 to 100).

`median-from-sorted-data`

```
Function:   (median-from-sorted-data sorted-data
Contract:   (-> (and/c non-empty-vector-of-reals? sorted?)
                real?)
```

This function returns the median value of *sorted-data*. When the dataset has an odd number of elements, the median is the value of element $(n-1)/2$. When the dataset has an even number of elements, the median is the mean of the two nearest middle values, elements $(n-1)/2$ and $n/2$.

`quantile-from-sorted-data`

```
Function:   (qualtile-from-sorted-data sorted-data f
Contract:   (-> (and/c non-empty-vector-of-reals? sorted?)
                (real-in 0.0 1.0) real?)
```

This function returns a quantile calue of *sorted-data*. The quantile is determined by the value *f*, a fraction between 0 and 1. For example, to compute the value of the 75^{th} percentile, f should have the value 0.75.

The quantile is found by interpolation using the formula

$$quantile = 1 - delta(x[i]) + delta(x[i+1])$$

where i is $floor((n-1)f)$ and $delta$ is $(n-1)f - 1$.

8.9 Example

This example generates two vectors of data from a unit Gaussian distribution and a vector of cosine squared weighting data. All of the vectors are of length 1,000. These data are used to test all of the statistics functions.

```
(require (planet "gaussian.ss" ("williams" "science.plt" 2 0)
                 "random-distributions"))
(require (planet "statistics.ss" ("williams" "science.plt" 2 0)))
(require (planet "math.ss" ("williams" "science.plt" 2 0)))

(define (naive-sort! data)
  (let loop ()
    (let ((n (vector-length data))
          (sorted? #t))
      (do ((i 1 (+ i 1)))
          ((= i n) data)
        (if (< (vector-ref data i)
               (vector-ref data (- i 1)))
            (let ((t (vector-ref data i)))
              (vector-set! data i (vector-ref data (- i 1)))
              (vector-set! data (- i 1) t)
              (set! sorted? #f))))
        (if (not sorted?)
            (loop)))))

(let ((data1 (make-vector 1000))
      (data2 (make-vector 1000))
      (w     (make-vector 1000)))
  (do ((i 0 (+ i 1)))
      ((= i 1000) (void))
    ;; Random data from unit gaussian
    (vector-set! data1 i (random-unit-gaussian))
    (vector-set! data2 i (random-unit-gaussian))
    ;; Cos^2 weighting
    (vector-set! w i
      (expt (cos (- (* 2.0 pi (/ i 1000.0)) pi)) 2)))
  (printf "Statistics Example~n")
  (printf "                                mean = ~a~n"
          (mean data1))
  (printf "                            variance = ~a~n"
          (variance data1))
  (printf "                  standard deviation = ~a~n"
          (standard-deviation data1))
```

```
(printf "                   variance from 0.0 = ~a~n"
        (variance-with-fixed-mean data1 0.0))
(printf "         standard deviation from 0.0 = ~a~n"
        (standard-deviation-with-fixed-mean data1 0.0))
(printf "                  absolute deviation = ~a~n"
        (absolute-deviation data1))
(printf "         absolute deviation from 0.0 = ~a~n"
        (absolute-deviation data1 0.0))
(printf "                                skew = ~a~n"
        (skew data1))
(printf "                            kurtosis = ~a~n"
        (kurtosis data1))
(printf "               lag-1 autocorrelation = ~a~n"
        (lag-1-autocorrelation data1))
(printf "                          covariance = ~a~n"
        (covariance data1 data2))
(printf "                       weighted mean = ~a~n"
        (weighted-mean w data1))
(printf "                   weighted variance = ~a~n"
        (weighted-variance w data1))
(printf "         weighted standard deviation = ~a~n"
        (weighted-standard-deviation w data1))
(printf "          weighted variance from 0.0 = ~a~n"
        (weighted-variance-with-fixed-mean w data1 0.0))
(printf "weighted standard deviation from 0.0 = ~a~n"
        (weighted-standard-deviation-with-fixed-mean w data1 0.0))
(printf "         weighted absolute deviation = ~a~n"
        (weighted-absolute-deviation w data1))
(printf "weighted absolute deviation from 0.0 = ~a~n"
        (weighted-absolute-deviation w data1 0.0))
(printf "                       weighted skew = ~a~n"
        (weighted-skew w data1))
(printf "                   weighted kurtosis = ~a~n"
        (weighted-kurtosis w data1))
(printf "                             maximum = ~a~n"
        (maximum data1))
(printf "                             minimum = ~a~n"
        (minimum data1))
(printf "              index of maximum value = ~a~n"
        (maximum-index data1))
(printf "              index of minimum value = ~a~n"
        (minimum-index data1))
(naive-sort! data1)
(printf "                              median = ~a~n"
        (median-from-sorted-data data1))
(printf "                        10% quantile = ~a~n"
        (quantile-from-sorted-data data1 .1))
(printf "                        20% quantile = ~a~n"
        (quantile-from-sorted-data data1 .2))
(printf "                        30% quantile = ~a~n"
         (quantile-from-sorted-data data1 .3))
  (printf "                        40% quantile = ~a~n"
         (quantile-from-sorted-data data1 .4))
  (printf "                        50% quantile = ~a~n"
         (quantile-from-sorted-data data1 .5))
  (printf "                        60% quantile = ~a~n"
         (quantile-from-sorted-data data1 .6))
  (printf "                        70% quantile = ~a~n"
         (quantile-from-sorted-data data1 .7))
  (printf "                        80% quantile = ~a~n"
         (quantile-from-sorted-data data1 .8))
  (printf "                        90% quantile = ~a~n"
         (quantile-from-sorted-data data1 .9))
)
```

Produces the following output:

```
Statistics Example
                                mean = 0.03457693091555611
                            variance = 1.0285343857083422
                  standard deviation = 1.0141668431320077
                   variance from 0.0 = 1.028701415474174
         standard deviation from 0.0 = 1.014249188056946
                  absolute deviation = 0.7987180852601665
         absolute deviation from 0.0 = 0.7987898146946209
                                skew = 0.043402934671178436
                            kurtosis = 0.17722452271704014
               lag-1 autocorrelation = 0.0029930889831972143
                          covariance = 0.005782911085590894
                       weighted mean = 0.05096139259270008
                   weighted variance = 1.0500293763787367
         weighted standard deviation = 1.0247094107007786
          weighted variance from 0.0 = 1.0510513958491579
weighted standard deviation from 0.0 = 1.0252079768755011
         weighted absolute deviation = 0.8054378524718832
weighted absolute deviation from 0.0 = 0.8052440544958938
                       weighted skew = 0.046448729539282155
                   weighted kurtosis = 0.3050060704791675
                             maximum = 3.731148814104969
                             minimum = -3.327265864298485
              index of maximum value = 502
              index of minimum value = 476
                              median = 0.019281803306206644
                        10% quantile = -1.243869878615807
                        20% quantile = -0.7816243947573505
                        30% quantile = -0.4708703241429585
                        40% quantile = -0.2299309332835332
                        50% quantile = 0.019281803306206644
                        60% quantile = 0.30022966479982344
                        70% quantile = 0.5317978807508836
                        80% quantile = 0.832291888537874
                        90% quantile = 1.3061151234700463
```

Chapter 9

Histograms

This chapter describes the functions for creating and using histograms provided by the PLT Scheme Science Collection. Histograms provide a convenient way of summarizing the distribution of a set of data. A histogram contains a vector of bins that count the number of events falling into a given range. The bins of a histogram can be used to record both integer and non-integer distributions.

The ranges of the bins can be either continuous or discrete over a range. For continuous ranges, the width of these ranges can be either fixed or arbitrary. Also, for continuous ranges, both one and two dimensional histograms are supported.

9.1 1D Histograms

The 1D histogram functions described in this section are defined in the `histogram.ss` file in the science collection and are made available using the following form:

```
(require (planet "histogram.ss" ("williams" "science.plt" 2 0)))
```

`histogram?`

Function: `(histogram? x)`
Contract: `(-> any? boolean?)`

This function returns true, `#t`, if x is a histogram and false, `#f` otherwise.

9.1.1 Creating 1D Histograms

`make-histogram`

Function: `(make-histogram n)`
Contract: `(-> (integer-in 1 +inf.0) histogram?)`

This function returns a new, empty histogram with n bins and $n + 1$ range entries. The range entries must be set with a subsequent call to `set-histogram-ranges!` or `set-histogram-ranges-uniform!`.

`make-histogram-with-ranges-uniform`

Function: `(make-histogram-with-ranges-uniform n x-min x-max)`
Contract:
```
(->r ((n (integer-in 1 +inf.0)
      (x-min real?)
      (x-max (>/c x-min)))
     histogram?)
```

This function returns a new, empty histogram with n bins. The $n + 1$ range entries are initialized to provide n uniform width bins from *x-min* to *x-max*.

9.1.2 Updating and Accessing 1D Histogram Elements

`histogram-n`

Function: `(histogram-n h)`
Contract: `(-> histogram? (integer-in 1 +inf.0))`

This function returns the number of bins in the histogram h.

`histogram-ranges`

Function: `(histogram-ranges h)`
Contract: `(-> histogram? (vectorof real?))`

This function returns the vector of ranges for the histogram h. The length of the vector is equal to the number of bins in h plus one.

`set-histogram-ranges!`

Function: `(set-histogram-ranges! h ranges)`
Contract: `(-> histogram? (vectorof real?) void?)`

This function sets the ranges for the histogram h according to the given *ranges*. The length of the *ranges* vector must be equal to the number of bins in h plus one. The bins in h are also reset.

`set-histogram-ranges-uniform!`

Function: `(set-histogram-ranges-uniform! h x-min x-max)`
Contract:
```
(->r ((h histogram?)
      (x-min real?)
      (x-max (>/c x-min)))
     void?)
```

This function sets the ranges for the histogram h uniformly from *x-min* to *x-max*. The bins in h are also reset.

`histogram-bins`

Function: `(histogram-bins h)`
Contract: `(-> histogram? (vectorof real?))`

This functions returns the vector of bins for the histogram h.

`histogram-increment!`

Function: `(histogram-increment! h x)`
Contract: `(-> histogram? real? void?)`

This function increments the bin in the histogram *h* containing *x*. The bin value is incremented by one.

`histogram-accumulate!`

Function: `(histogram-accumulate! h x weight)`
Contract: `(-> histogram? real (>-/c 0.0) void?)`

This function increments the bin in the histogram *h* containing *x* by the specified *weight*.

`histogram-get`

Function: `(histogram-get h i)`
Contract: `(-> histogram? natural-number? (>=/c 0.0))`

This functions returns the contents of the i^{th} bin of the histogram *h*.

`histogram-get-range`

Function: `(histogram-get-range h i)`
Contract: `(-> histogram? natural-number? (values real? real?))`

This function returns the upper and lower range limits for the i^{th} bin of the histogram *h*. The upper and lower range limits are returned as multiple values.

9.1.3 1D Histogram Statistics

`histogram-max`

Function: `(histogram-max h)`
Contract: `(-> histogram? (>=/c 0.0))`

This function returns the maximum bin value in the histogram *h*. Since in this implementation bin values are non-negative, the maximum value is also non-negative.

`histogram-min`

Function: `(histogram-min h)`
Contract: `(-> histogram? (>=/c 0.0))`

This function returns the minimum bin value in the histogram *h*. Since in this implementation bin values are non-negative, the minimum value is also non-negative.

`histogram-mean`

Function: `(histogram-mean h)`
Contract: `(-> histogram? (>=/c 0.0))`

This function returns the mean of the data in the histogram *h*.

`histogram-sigma`

Function: `(histogram-sigma h)`
Contract: `(-> histogram? (>=/c 0.0))`

This function returns the standard deviation of the data in the histogram *h*.

`histogram-sum`

Function: `(histogram-sum h)`
Contract: `(-> histogram? (>=/c 0.0))`

This function returns the sum of the data in the histogram *h*.

9.1.4 1D Histogram Graphics

The histogram graphics functions are defined in the file `histogram-graphics.ss` in the science collection and are made available using the following form:

```
(require (planet "histogram-graphics.ss" ("williams" "science.plt" 2 0)))
```

`histogram-plot`

```
Function:  (histogram-plot h title)
Function:  (histogram-plot h)
Contract:  (case->
              (-> histogram? string? any)
              (-> histogram? any))
```

This function returns a plot of the histogram *h* with the specified *title*. If *title* is not specified, `"Histogram"` is used. The plot is scaled to the maximum bin value. The plot is produced by the histogram plotting extension to the plot collection provided with PLT Scheme (PLoT Scheme).

`histogram-plot-scaled`

```
Function:  (histogram-plot-scaled h title)
Function:  (histogram-plot-scaled h)
Contract:  (case->
              (-> histogram? string? any)
              (-> histogram? any))
```

This function returns a plot of the histogram *h* with the specified *title*. If *title* is not specified, `"Histogram"` is used. The plot is scaled to the sum of the bin values. It is most useful for a small number of bin - generally, ten or less. The plot is produced by the histogram plotting extension to the plot collection provided with PLT Scheme (PLoT Scheme).

9.1.5 Examples

Example: Plot of histogram of random variates from the unit Gaussian (normal) distribution.

```
(require (planet "gaussian.ss" ("williams" "science.plt" 2 0)
                 "random-distributions"))
(require (planet "histogram-with-graphics.ss"
                 ("williams" "science.plt" 2 0)))

(let ((h (make-histogram-with-ranges-uniform 40 -3.0 3.0)))
  (do ((i 0 (+ i 1)))
      ((= i 10000) (void))
    (histogram-increment! h (random-unit-gaussian)))
  (histogram-plot h "Histogram of Unit Gaussian Distribution"))
```

Figure 9.1 shows the resulting histogram.

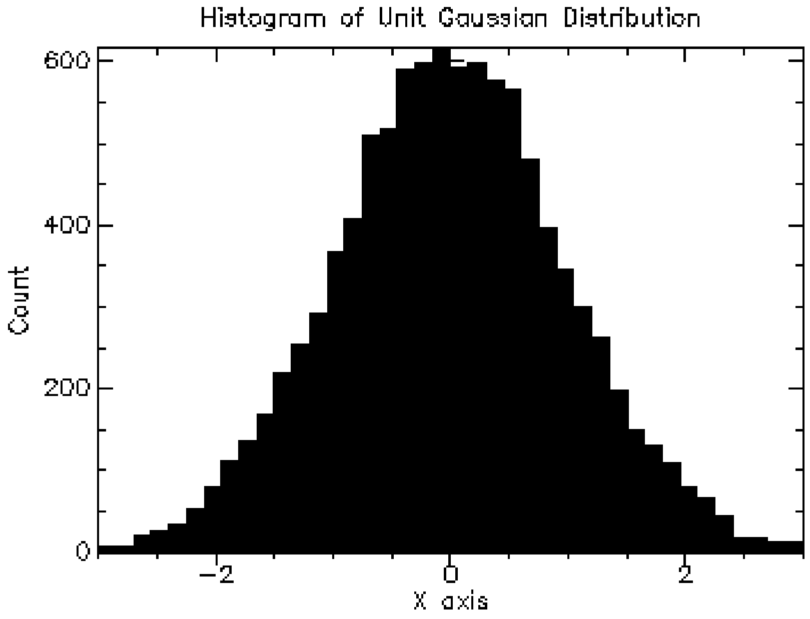

Figure 9.1: Histogram of Random Variates from Unit Gaussian (Normal)

Example: Scaled plot of histogram of random variates from the exponential distribution with mean 1.0.

```
(require (planet "exponential.ss" ("williams" "science.plt" 2 0)
                 "random-distributions"))
(require (planet "histogram-with-graphics.ss"
                 ("williams" "science.plt" 2 0)))
```

```
(let ((h (make-histogram-with-ranges-uniform 10 0.0 8.0)))
  (do ((i 0 (+ i 1)))
      ((= i 10000) (void))
    (histogram-increment! h (random-exponential 1.0)))
  (histogram-plot-scaled
    h "Scaled Histogram of Exponential Distribution"))
```

Figure 9.2 shows the resulting histogram.

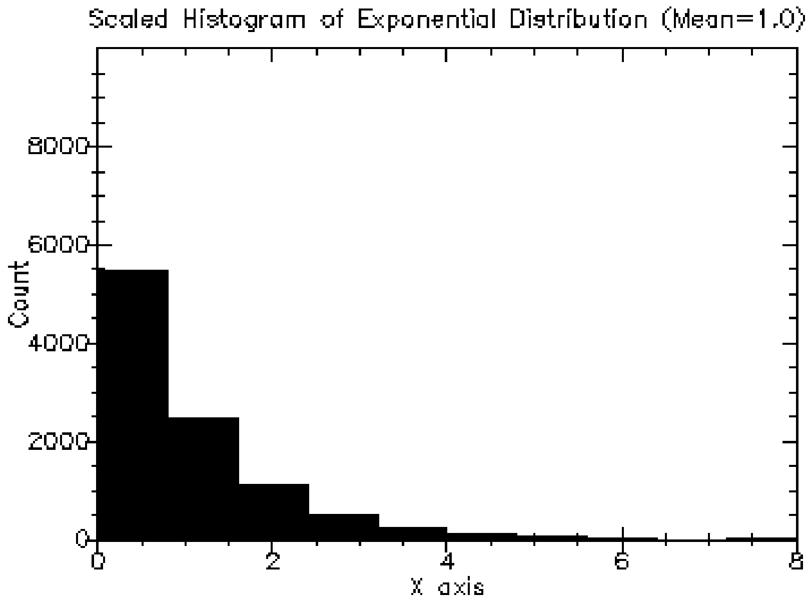

Figure 9.2: Scaled Histogram of Random Variates from Exponential (1.0))

9.2 2D Histograms

The 2D histogram functions described in this chapter are defined in the `histogram-2d.ss` file in the science collection and are made available using the following form:

```
(require (planet "histogram-2d.ss" ("williams" "science.plt" 2 0)))
```

`histogram-2d?`

Function: `(histogram-2d? x)`
Contract: `(-> any? boolean?)`

This function returns true, `#t`, if x is a 2D histogram and false, `#f` otherwise.

9.2.1 Creating 2D Histograms

`make-histogram-2d`

```
Function:   (make-histogram-2d nx ny)
Contract:   (-> (integer-in 1 +inf.0) (integer-in 1 +inf.0) histogram-2d?)
```

This function returns a new, empty 2D histogram with *nx* bins in the x direction and *ny* bins in the y direction and $nx + 1$ range entries in the x direction and $ny + 1$ range entries in the y direction. The range entries must be set with a subsequent call to `set-histogram-2d-ranges!` or `set-histogram-2d-ranges-uniform!`.

`make-histogram-2d-with-ranges-uniform`

```
Function:   (make-histogram-2d-with-ranges-uniform
               nx ny x-min x-max y-min y-max)
Contract:   (->r ((nx (integer-in 1 +inf.0))
                  (ny (integer-in 1 +inf.0))
                  (x-min real?)
                  (x-max (>/c x-min))
                  (y-min real?)
                  (y-max (>/c y-min)))
                 histogram-2d?)
```

This function returns a new, empty 2D histogram with *nx* bins in the x direction and *ny* bins in the y direction. The $nx + 1$ range entries in the x direction are initialized to provide *nx* uniform width bins from *x-min* to *x-max*. The $ny + 1$ range entries in the y direction are initialized to provide *ny* uniform width bins from *y-min* to *y-max*.

9.2.2 Updating and Accessing 2D Histogram Elements

`histogram-2d-nx`

```
Function:   (histogram-2d-nx h)
Contract:   (-> histogram-2d? (integer-in 1 +inf.0))
```

This function returns the number of bins in the x direction in the 2D histogram *h*.

`histogram-2d-ny`

```
Function:   (histogram-2d-ny h)
Contract:   (-> histogram-2d? (integer-in 1 +inf.0))
```

This function returns the number of bins in the y direction in the 2D histogram *h*.

`histogram-2d-x-ranges`

```
Function:   (histogram-2d-x-ranges h)
Contract:   (-> histogram-2d? (vectorof real?))
```

This function returns the vector of ranges in the x direction for the 2D histogram *h*. The length of the vector is equal to the number of bins in the x direction in *h* plus one.

`histogram-2d-y-ranges`

Function: `(histogram-2d-y-ranges h)`
Contract: `(-> histogram-2d? (vectorof real?))`

This function returns the vector of ranges in the y direction for the 2D histogram *h*. The length of the vector is equal to the number of bins in the y direction in *h* plus one.

`set-histogram-2d-ranges!`

Function: `(set-histogram-2d-ranges! h x-ranges y-ranges)`
Contract: `(-> histogram-2d? (vectorof real?) (vectorof real?) void?)`

This function sets the ranges for the 2D histogram *h* according to the given *x-ranges* and *y-ranges*. The length of the *x-ranges* vector must be equal to the number of bins in the x direction in *h* plus one. The length of the *y-ranges* vector must be equal to the number of bins in the y direction in *h* plus one. The bins in *h* are also reset.

`set-histogram-2d-ranges-uniform!`

Function: `(set-histogram-2d-ranges-uniform! h x-min x-max y-min y-max)`
Contract:
```
(->r ((h histogram-2d?)
      (x-min real?)
      (x-max (>/c x-min))
      (y-min real?)
      (y-max (>/c y-max)))
     void?)
```

This function sets the ranges for the 2D histogram *h* uniformly from *x-min* to *x-max* in the x direction and uniformly from *y-min* to *y-max* in the y direction. The bins in *h* are also reset.

`histogram-2d-bins`

Function: `(histogram-bins h)`
Contract: `(-> histogram-2d? (vectorof real?))`

This functions returns the vector of bins for the 2D histogram *h*. The length of the vector is $nx * ny$. The $(i, j)^{th}$ index is computed as $(i * ny) + j$.

`histogram-2d-increment!`

Function: `(histogram-increment! h x y)`
Contract: `(-> histogram-2d? real? real? void?)`

This function increments the bin in the 2D histogram *h* containing (x, y). The bin value is incremented by one.

`histogram-2d-accumulate!`

Function: `(histogram-2daccumulate! h x y weight)`
Contract: `(-> histogram-2d? real? real? (>-/c 0.0) void?)`

This function increments the bin in the 2D histogram *h* containing (x, y) by the specified *weight*.

`histogram-2d-get`

Function: `(histogram-2d-get h i j)`
Contract: `(-> histogram-2d? natural-number? natural-number? (>=/c 0.0))`

This functions returns the contents of the $(i, j)^{th}$ bin of the 2D histogram *h*.

`histogram-2d-get-x-range`

Function: `(histogram-2d-get-x-range h i j)`
Contract:
```
(-> histogram-2d? natural-number? natural-number?
    (values real? real?))
```

This function returns the upper and lower range limits in the x direction for the $(i, j)^{th}$ bin of the 2D histogram *h*. The upper and lower range limits are returned as multiple values.

`histogram-2d-get-y-range`

Function: `(histogram-2d-get-y-range h i j)`
Contract:
```
(-> histogram-2d? natural-number? natural-number?
    (values real? real?))
```

This function returns the upper and lower range limits in the y direction for the $(i, j)^{th}$ bin of the 2D histogram *h*. The upper and lower range limits are returned as multiple values.

9.2.3 2D Histogram Statistics

`histogram-2d-max`

Function: `(histogram-2d-max h)`
Contract: `(-> histogram-2d? (>=/c 0.0))`

This function returns the maximum bin value in the 2D histogram *h*. Since in this implementation bin values are non-negative, the maximum value is also non-negative.

`histogram-2d-min`

Function: `(histogram-2d-min h)`
Contract: `(-> histogram-2d? (>=/c 0.0))`

This function returns the minimum bin value in the 2D histogram *h*. Since in this implementation bin values are non-negative, the minimum value is also non-negative.

`histogram-2d-sum`

Function: `(histogram-2d-sum h)`
Contract: `(-> histogram-2d? (>=/c 0.0))`

This function returns the sum of the data in the 2D histogram *h*.

`histogram-2d-x-mean`

Function: `(histogram-2d-x-mean h)`
Contract: `(-> histogram-2d? (>=/c 0.0))`

This function returns the mean in the x direction of the data in the 2D histogram *h*.

`histogram-2d-y-mean`

Function: `(histogram-2d-y-mean h)`
Contract: `(-> histogram-2d? (>=/c 0.0))`

This function returns the mean in the y direction of the data in the 2D histogram *h*.

`histogram-2d-x-sigma`

Function: `(histogram-2d-x-sigma h)`
Contract: `(-> histogram-2d? (>=/c 0.0))`

This function returns the standard deviation in the x direction of the data in the 2D histogram *h*.

`histogram-2d-y-sigma`

Function: `(histogram-2d-y-sigma h)`
Contract: `(-> histogram-2d? (>=/c 0.0))`

This function returns the standard deviation in the y direction of the data in the 2D histogram *h*.

`histogram-2d-covariance`

Function: `(histogram-2d-covariance h)`
Contract: `(-> histogram-2d? (>=/c 0.0))`

This function returns the covariance of the data in the 2D histogram *h*.

9.2.4 2D Histogram Graphics

The 2D histogram graphics functions are defined in the file `histogram-2d-graphics.ss` in the science collection and are made available using the following form:

```
(require (planet "histogram-2d-graphics.ss"
                 ("williams" "science.plt" 2 0)))
```

`histogram-2d-plot`

Function: `(histogram-2d-plot h title)`
Function: `(histogram-2d-plot h)`
Contract:
```
(case->
   (-> histogram-2d? string? any)
   (-> histogram-2d? any))
```

This function returns a plot of the 2D histogram *h* with the specified *title*. If *title* is not specified, `"Histogram"` is used. The plot is scaled to the maximum bin value. The plot is produced by the 2D histogram plotting extension to the plot collection provided with PLT Scheme (PLoT Scheme).

9.2.5 Example

Example: Plot of 2D histogram of random variates from the bivariate Gaussian distribution standard deviations 1.0 and 1.0 in the x and y directions and correlation coefficient 0.0.

```
(require (planet "bivariate-gaussian.ss" ("williams" "science.plt" 2 0)
                 "random-distributions"))
(require (planet "histogram-2d-with-graphics.ss"
                 ("williams" "science.plt" 2 0)))

(let ((h (make-histogram-2d-with-ranges-uniform
          20 20 -3.0 3.0 -3.0 3.0)))
  (do ((i 0 (+ i 1)))
      ((= i 10000) (void))
    (let-values (((x y) (random-bivariate-gaussian 1.0 1.0 0.0)))
      (histogram-2d-increment! h x y)))
  (histogram-2d-plot h "Histogram of Bivariate Gaussian Distribution"))
```

Figure 9.3 shows the resulting 2D histogram.

9.3 Discrete Histograms

The discrete histogram functions described in this section are defined in the `discrete-histogram.ss` file in the science collection and are made available using the following form:

```
(require (planet "discrete-histogram.ss" ("williams" "science.plt" 2 0)))
```

`discrete-histogram?`

Function: `(discrete-histogram? x)`
Contract: `(-> any? boolean?)`

This function returns true, `#t`, if *x* is a discrete histogram and false, `#f` otherwise.

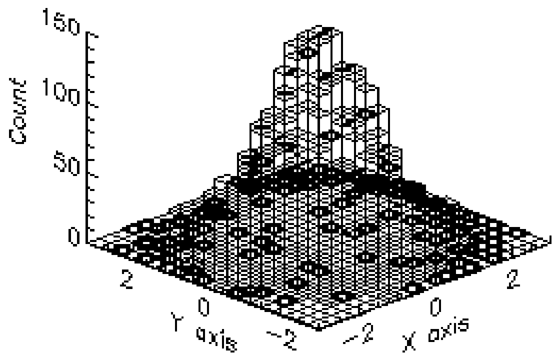

Figure 9.3: Histogram of Random Variates from Bivariate Gaussian (1.0, 1.0, 0.0)

9.3.1 Creating Discrete Histograms

`make-discrete-histogram`

```
Function:   (make-discrete-histogram n1 n2 dynamic?)
Function:   (make-discrete-histogram n1 n2)
Function:   (make-discrete-histogram)
Contract:   (case-> (->r ((n1 integer?)
                          (n2 (and/c integer? (>=/c n1)))
                          (dynamic? boolean?))
                         discrete-histogram?)
                    (->r ((n1 integer?)
                          (n2 (and/c integer? (>=/c n1))))
                         discrete-histogram?)
                    (->r discrete-histogram?)))
```

This function returns a new, empty discrete histogram with range *n1* to *n2*. If *dynamic?* is `#t` or `make-discrete-histogram` is called with no arguments, the resulting discrete histogram will grow dynamically to accomodate subsequent data points.

9.3.2 Updating and Accessing Discrete Histogram Elements

`discrete-histogram-n1`

```
Function:   (discrete-histogram-n1 h)
Contract:   (-> discrete-histogram? integer?)
```

This function returns the lower range of the discrete histogram *h*.

`discrete-histogram-n2`

```
Function:   (discrete-histogram-n2 h)
Contract:   (-> discrete-histogram? integer?)
```

This function returns the upper range of the discrete histogram *h*.

`discrete-histogram-dynamic?`

```
Function:   (discrete-histogram-dynamic? h)
Contract:   (-> discrete-histogram? boolean?)
```

This function returns `#t` if the discrete histogram *h* is dynamic and `#f` otherwise.

`discrete-histogram-bins`

```
Function:   (discrete-histogram-bins h)
Contract:   (-> discrete-histogram? (vectorof real?))
```

This functions returns the vector of bins for the discrete histogram *h*.

`discrete-histogram-increment!`

Function: `(discrete-histogram-increment! h i)`
Contract: `(-> discrete-histogram? integer? void?)`

This function increments the bin in the discrete histogram *h* containing *i*. The bin value is incremented by one.

`discrete-histogram-accumulate!`

Function: `(discrete-histogram-accumulate! h i weight)`
Contract: `(-> discrete-histogram? integer? (>-/c 0.0) void?)`

This function increments the bin in the discrete histogram *h* containing *i* by the specified *weight*.

`discrete-histogram-get`

Function: `(discrete-histogram-get h i)`
Contract: `(-> discrete-histogram? integer? (>=/c 0.0))`

This functions returns the contents of the bin of the discrete histogram *h* containing *i*.

9.3.3 Discrete Histogram Statistics

`discrete-histogram-max`

Function: `(discrete-histogram-max h)`
Contract: `(-> discrete-histogram? (>=/c 0.0))`

This function returns the maximum bin value in the discrete histogram *h*. Since in this implementation bin values are non-negative, the maximum value is also non-negative.

`discrete-histogram-min`

Function: `(discrete-histogram-min h)`
Contract: `(-> discrete-histogram? (>=/c 0.0))`

This function returns the minimum bin value in the discrete histogram *h*. Since in this implementation bin values are non-negative, the minimum value is also non-negative.

`discrete-histogram-sum`

Function: `(discrete-histogram-sum h)`
Contract: `(-> discrete-histogram? (>=/c 0.0))`

This function returns the sum of the data in the discrete histogram *h*.

9.3.4 Discrete Histogram Graphics

The discrete histogram graphics functions are defined in the file `discrete-histogram-graphics.ss` in the science collection and are made available using the following form:

```
(require (planet "discrete-histogram-graphics.ss"
                 ("williams" "science.plt" 2 0)))
```

`discrete-histogram-plot`

Function: `(discrete-histogram-plot h title)`
Function: `(discrete-histogram-plot h)`
Contract:
```
(case->
   (-> discrete-histogram? string? any)
   (-> discrete-histogram? any))
```

This function returns a plot of the discrete histogram *h* with the specified *title*. If *title* is not specified, `"Histogram"` is used. The plot is scaled to the maximum bin value. The plot is produced by the discrete histogram plotting extension to the plot collection provided with PLT Scheme (PLoT Scheme).

`discrete-histogram-plot-scaled`

Function: `(discrete-histogram-plot-scaled h title)`
Function: `(discrete-histogram-plot-scaled h)`
Contract:
```
(case->
   (-> discrete-histogram? string? any)
   (-> discrete-histogram? any))
```

This function returns a plot of the discrete histogram *h* with the specified *title*. If *title* is not specified, `"Histogram"` is used. The plot is scaled to the sum of the bin values. It is most useful for a small number of bin - generally, ten or less. The plot is produced by the discrete histogram plotting extension to the plot collection provided with PLT Scheme (PLoT Scheme).

9.3.5 Examples

Example: Plot of discrete histogram of random variates from the Poisson distribution with mean 10.0.

```
(require (planet "poisson.ss" ("williams" "science.plt" 2 0)
                 "random-distributions"))
(require (planet "discrete-histogram-with-graphics.ss"
                 ("williams" "science.plt" 2 0)))

(let ((h (make-discrete-histogram)))
  (do ((i 0 (+ i 1)))
      ((= i 10000) (void))
    (discrete-histogram-increment! h (random-poisson 10.0)))
  (discrete-histogram-plot h "Histogram of Poisson Distribution"))
```

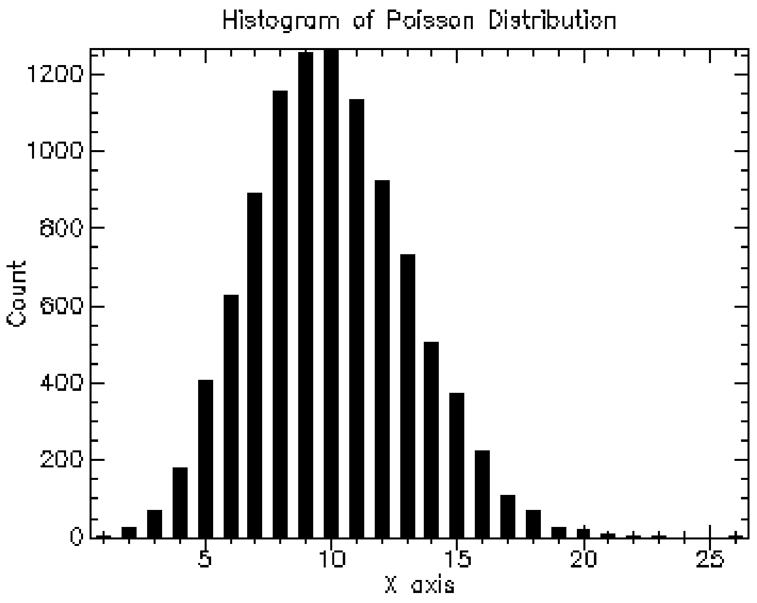

Figure 9.4: Histogram of Random Variates from Poisson (10.0)

Figure 9.4 shows the resulting histogram.

Example: Scaled plot of discrete histogram of random variates from the logarithmic distribution with probability 0.5.

```
(require (planet "logarithmic.ss" ("williams" "science.plt" 2 0)
                 "random-distributions"))
(require (planet "discrete-histogram-with-graphics.ss"
                 ("williams" "science.plt" 2 0)))

(let ((h (make-discrete-histogram)))
  (do ((i 0 (+ i 1)))
      ((= i 10000) (void))
    (discrete-histogram-increment! h (random-logarithmic 0.5)))
  (discrete-histogram-plot-scaled
    h "Histogram of Logarithmic Distribution"))
```

Figure 9.5 shows the resulting histogram.

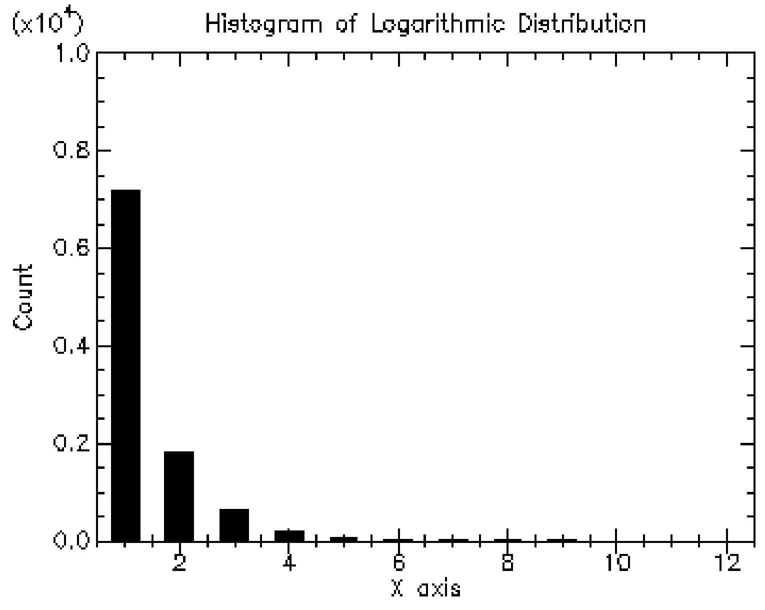

Figure 9.5: Histogram of Random Variates from Logarithmic (0.5)

Chapter 10

Ordinary Differential Equations

This chapter describes funtions for solving ordinary differential equation (ODE) initial value problems. The PLT Scheme Science Collection provides a variety of low-level methods, such as Runge-Kutta and Bulirsch-Stoer routines, and higher-level components for adaptive step-size control. The components can be combined by the user to achieve the desired solution, with full access to any intermediate steps. The functions described in this chapter are defined in the `ode-initval.ss` file in the science collection and are made available using the form:

```
(require (planet "ode-initval.ss" ("williams" "science.plt" 2 0)))
```

10.1 Defining the ODE System

The routines solve the general n-dimensional first-order system,

$$dy_i(t)/dt = f_i(t, y_i(t), ..., y_n(t))$$

for $i = 1, \ldots, n$. The stepping functions rely on the vector of derivatives f_i and the Jacobian matrix, $J_{ij} = df_i(t, y(t))/dy_j$. A system of equations is defined using the `ode-system` structure.

`ode-system`

Structure: `ode-system`

This structure defines a general ODE system with arbitrary parameters.

`function` – A function `(lambda (t y dydt params) ...)`. This function should store the elements of $f_i(t, y, params)$ in the vector *dydt*, for arguments *(t, y)* and parameters *params*.

`jacobian` – A function `(lambda (t y dfdy dfdt params) ...`. This function should store the elements $df_i(t, y, params)/dt$ in the vector *dfdt* and the

Jacobian matrix J_{ij} in the vector *dfdy* as a row-ordered matrix `J(i,j) = dfdy(i * dimension + j)` where *dimension* is the dimension of the system. Some of the simplier solver algorithms do not make use of the Jacobian matrix, so it is not always strictly necessary to provide it (the `jacobian` field of the structure can be replace by `#f` for those algorithms). However, it is useful to provide the Jacobian to allow the solver algorithms to be interchanged. The best algorithms make use of the Jacobian.

`dimension` – This is the dimension of the system of equations.

`params` – This is a list of the arbitrary parameters of the system.

10.2 Stepping Functions

The lowest level components are the *stepping functions* that advance a solution from time t to $t + h$ for a fixed step size h and estimate the resulting local error.

`make-ode-step`

Function: `(make-ode-step step-type dim)`
Contract: `(-> ode-step-type? natural? ode-step?)`

This function returns a newly created instance of a stepping function of type *step-type* for a system of *dim* dimensions.

`ode-step-reset`

Function: `(ode-step-reset step)`
Contract: `(-> ode-step? void)`

This function resets the stepping function *step*. It should be used whenever the next use of *step* will not be a continuation of a previous step.

`ode-step-name`

Function: `(ode-step-name step)`
Contract: `(-> ode-step? string?)`

This function returns the name of the stepping function *step* as a string.

`ode-step-order`

Function: `(ode-step-order step)`
Contract: `(-> ode-step? natural?)`

This function returns the order of the stepping function *step* on the previous step. This order can vary if the stepping function itself is adaptive.

`ode-step-apply`

Function:
```
(ode-step-apply step t h y y-err dydt-in dydt-out dydt)
```
Contract:
```
(-> ode-step? real? real?
    (vector-of real?) (vector-of real?)
    (vector-of real?) (vector-of real?)
    ode-system? void)
```

This function applies the stepping function *step* to the system of equations defined by *dydt*, using the step size h to advance the system from time t and state y to time $t+h$. The new state of the system is stored in y on output, with an estimate of the absolute error in each component stored in *y-err*. If the argument *dydt-in* is not `#f`, it should be a vector containing the derivatives for the system at time t on input. This is optional as the derivatives will be computed internally if they are not provided, but allows the reuse of existing derivative information. On output the new derivatives of the system at time $t+h$ will be stored in the vector *dydt-out*, if it is not `#f`.

The following stepping algorithms are available.

Step Type: `ode-step-rk2`

Embedded Runge-Kutta (2,3) method.

Step Type: `ode-step-rk4`

4^{th} order (classical) Runge-Kutta.

Step Type: `ode-step-rkf45`

Embedded Runge-Kutta-Fehlberg (4,5) method. This method is a good general-purpose integrator.

10.3 Adaptive Step-Size Control

The control function examines the proposed change to the solution and its error estimate produced by a stepping function and attempts to determine the optimal step-size for a user-specified level of error.

`ode-control-standard-new`

Function: `(ode-control-standard-new eps-abs eps-rel a-y a-dydt)`
Contract: `(-> real? real? real? real?)`

The standard control object is a four parameter heuristic based on absolute and relative errors *eps-abs* and *eps-rel*, and scaling factors *a-y* and *a-dydt* for the system stste *y(t)* and derivatives *y'(t)*, respectively.

The step size adjustment procedure for this method begins by computing the desired error level D_i for each component,

$$D_i = eps_{abs} + eps_{rel} * (a_y|y_i| + a_{dydt}h|y'_i|)$$

and comparing it with the observed error $E_i = |yerr_i|$. If the observed error E exceeds the desired error level D by more than 10

$$h_{new} = h_{old} * S * (E/D)^{-1/q)}$$

where q is the consistency order of the method (e.g. q=4 for 4(5) embedded RK), and S is a safety factor of 0.9. The ratio E/D is taken to be the maximum of the ratios E_i/D_i.

If the observed error E is less than 50% of the desired level D for the maximum ratio E_i/D_I, then the algorithm takes the opportunity to increase the step size to bring the error in line with the desired level,

$$h_{new} = h_{old} * S * (E/D)^{(-1/(q+1))}$$

This encompasses all the standard scaling methods. To avoid uncontrolled changes in the step size, the overall scaling factor is limited to the range 1/5 to 5.

`ode-control-y-new`

<u>Function</u>: `(ode-control-y-new eps-abs eps-rel)`
<u>Contract</u>: `(-> real? real?)`

This function creates a new control object that will keep the local error within an absolute error of *eps-abs* and relative error *eps-rel* with respect to the solution $y_i(t)$. This is equivalent to the standard control object with *a-y* = 1 and *a-dydt* = 0.

`ode-control-yp-new`

<u>Function</u>: `(ode-control-yp-new eps-abs eps-rel)`
<u>Contract</u>: `(-> real? real?)`

This function creates a new control object that will keep the local error within an absolute error of *eps-abs* and relative error *eps-rel* with respect to the derivatives of the solution $y'_i(t)$. This is equivalent to the standard control object with *a-y* = 0 and *a-dydt* = 1.

`ode-control-scaled-new`

```
Function:  (ode-control-scaled-new eps-abs eps-rel a-y a-dydt
                                   scale-abs dim)
Contract:  (-> real? real? real? real? (vector-of real?) natural?)
```

This function creates a new control object that uses the same algorithm as `ode-control-standard-new`, but with an absolute error that is scaled for each component by the array *scale-abs*. The formula for D_i for this control object is,

$$D_i = eps_{abs} * s_i + eps_{rel} * (a_y|y_i| + a_{dydt}h|y'_i|)$$

where s_i is the i^{th} component of the array *scale-abs*. The same error control heuristic is used by the Matlab ODE suite.

`ode-control`

Function: `(make-ode-control control-type)`
Contract: `(-> ode-control-type? ode-control?)`

This function returns a new instance of a control function of type *control-type*. This function is only needed for defining new types of control functions. For most purposes, the standard control functions described above should be sufficient.

`ode-control-init`

Function: `(ode-control-init control eps-abs eps-rel a-y a-dydt)`
Contract: `(-> ode-control? real? real? real? real? any)`

This function initializes the control function *control* with the parameters *eps-abs* (absolute error), *eps-rel* (relative error), *a-y* (scaling factor for y), and *a-dydt* scaling factor for derivatives.

`ode-control-h-adjust`

Function: `(ode-control-h-adjust control step y y-err dydt h)`
Contract:
```
(-> ode-control? ode-step
    (vectorof real?) (vectorof real?)
     (vectorof real?) box? any)
```

This function adjusts the step size h using the control function *control* and the current values of *y*, *y-err*, and *dydt*. The stepping function *step* is also needed to determine the order of the method. If the error in the y valyes *y-err* is found to be too large, then the step-size h is reduced and the function returns -1. If the error is sufficiently small, then h may be increased and 1 is returned. The function returns 0 if the step-size is unchanged. The goal of the function is to estimate the largest step-size that satisfies the user-specified accuracy requirement for the current point.

`ode-control-name`

Function: `(ode-control-name control)`
Contract: `(-> ode-control? string?)`

This function returns a pointer to the name of the control function. For eaxmple,

```
(printf ("control method is '~a'~n"
         (ode-control-name control)))
```

would print something like `control mathod is 'standard'`.

10.4 Evolution

The highest-level of the system is the evolution function that combines the results of a stepping function and control function to reliably advance the solution forward over an interval (t_0, t_1). If the control function signals that the step size should be descreased, the evolution function backs out of the current step and tries the proposed smaller step size. This process is continued until an acceptable step size is found.

`make-ode-evolve`

Function: `(make-ode-evolve dim)`
Contract: `(-> positive? ode-evolve?)`

This function returns a new instance of an evolution function for a system of *dim* dimensions.

`ode-evolve-apply`

```
Function:  (ode-evolve-apply evolve control step system
                             t t1 h y)
Contract:  (-> ode-evolve? ode-control? ode-step? ode-system?
               box? real? box? (vectorof real?) any)
```

This function evolves the system from time *t* and position *y* using the stepping function *step*. The new time and position are stored in *t* and *y* on output. The initial step size is taken as *h*, but this may be modified using the control function *control* to achieve the appropriate bound, if necessary. The routine may make several calls to *step* in order to determine the optimum step size. If the step size has been changed, the value of *h* will be modified on output. The maximum time *t1* is guaranteed not to be exceeded by the time step. On the final time step, the value of *t* will be set to *t1* exactly.

`ode-evolve-reset`

Function: `(ode-evolve-reset evolve)`
Contract: `(-> ode-evolve? any)`

This function resets the evolution function *evolve*. It should be used used whenever the next use of *evolve* will not be a continuation of a previous step.

10.5 Examples

Example: The following programs solve the second-order nonlinear Van der Pol oscillator equation,

$$x"(t) + \mu x'(t)(x(t)^2 - 1) + x(t) = 0$$

This can be converted into a first order system suitable for use with the routines described in this chapter by introducing a separate variable for the velocity, y = x'(t),

$$x' = y$$

$$y' = -x + \mu y(1 - x^2)$$

The following example integrates the above system of equations from $t = 0.0 to 100.0$ in increments of 0.01 using a 4^{th} order Runge-Kutta stepping function.

```
(require (planet "ode-initval.ss" ("williams" "science.plt" 2 0)))
(require (lib "plot.ss" "plot"))

(define (func t y f params)
  (let ((mu (car params))
        (y0 (vector-ref y 0))
        (y1 (vector-ref y 1)))
    (vector-set! f 0 y1)
    (vector-set! f 1 (- (- y0) (* mu y1 (- (* y0 y0) 1.0))))))

(define (main)
  (let* ((type rk4-ode-type)
         (step (make-ode-step type 2))
         (mu 10.0)
         (system (make-ode-system func #f 2 (list mu)))
         (t 0.0)
         (t1 100.0)
         (h 1.0e-2)
         (y #(1.0 0.0))
         (y-err (make-vector 2))
         (dydt-in (make-vector 2))
         (dydt-out (make-vector 2))
         (y0-values '())
         (y1-values '()))
    (ode-system-function-eval system t y dydt-in)
    (let loop ()
      (if (< t t1)
          (begin
            (ode-step-apply step t h
                              y y-err
                              dydt-in
                              dydt-out
                              system)
            ;(printf "~a ~a ~a~n" t (vector-ref y 0) (vector-ref y 1))
            (set! y0-values
                 (cons (vector t (vector-ref y 0)) y0-values))
            (set! y1-values
                 (cons (vector t (vector-ref y 1)) y1-values))
            (vector-set! dydt-in 0 (vector-ref dydt-out 0))
            (vector-set! dydt-in 1 (vector-ref dydt-out 1))
            (set! t (+ t h))
            (loop))))
    (printf "~a~n" (plot (points (reverse y0-values))
                         (x-min 0.0)
                         (x-max 100.0)
                         (y-min -2.0)
                         (y-max 2.0)))
    (printf "~a~n" (plot (points (reverse y1-values))
                         (x-min 0.0)
                         (x-max 100.0)))))
```

Figures 10.1 and 10.2 show the resulting output plots of y0 and y1.

Example: The following example evolves the above system of equations from $t = 0.0 to 100.0$ maintaining an error in the y value of 1.0e-6 using a 4^{th} order Runge-Kutta stepping function.

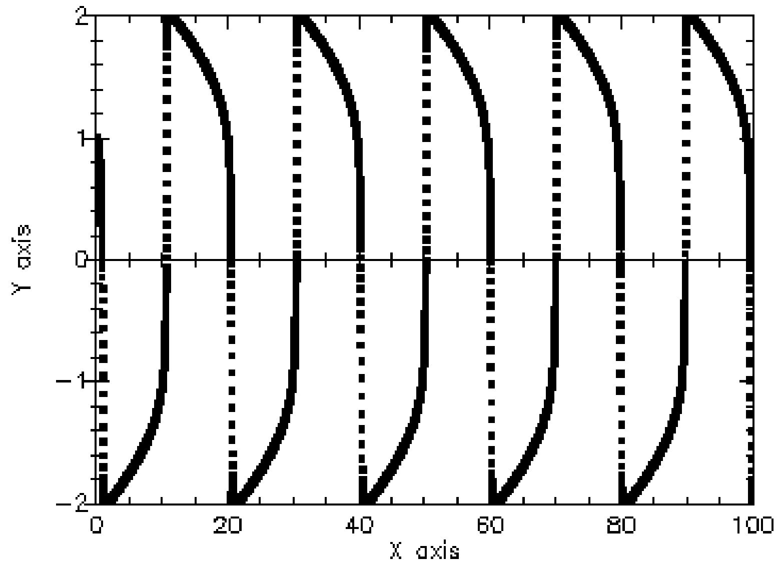

Figure 10.1: ODE Example 1 Plot of y0

```
(require (planet "ode-initval.ss" ("williams" "science.plt" 2 0)))
(require (lib "plot.ss" "plot"))

(define (func t y f params)
  (let ((mu (car params))
        (y0 (vector-ref y 0))
        (y1 (vector-ref y 1)))
    (vector-set! f 0 y1)
    (vector-set! f 1 (- (- y0) (* mu y1 (- (* y0 y0) 1.0))))))

(define (main)
  (let* ((type rk4-ode-type)
         (step (make-ode-step type 2))
         (control (control-y-new 1.0e-6 0.0))
         (evolve (make-ode-evolve 2))
         (mu 10.0)
         (system (make-ode-system func #f 2 (list mu)))
         (t (box 0.0))
         (t1 100.0)
         (h (box 1.0e-6))
         (y #(1.0 0.0))
         (y0-values '())
```

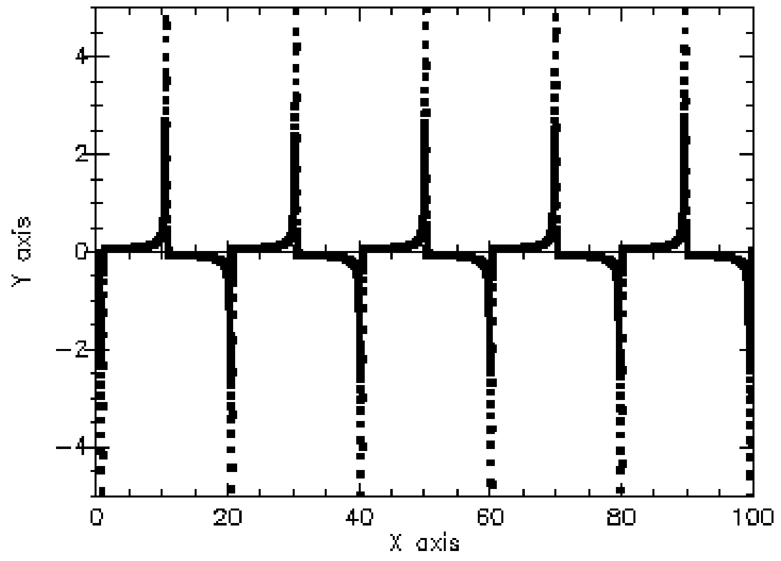

Figure 10.2: ODE Example 1 Plot of y1

```
     (y1-values '()))
(let loop ()
  (if (< (unbox t) t1)
      (begin
        (ode-evolve-apply
         evolve control step system
         t t1 h y)
        ;(printf "~a ~a ~a~n"
        ;        (unbox t) (vector-ref y 0) (vector-ref y 1))
        (set! y0-values
             (cons (vector (unbox t) (vector-ref y 0)) y0-values))
        (set! y1-values
             (cons (vector (unbox t) (vector-ref y 1)) y1-values))
        (loop))))
(printf "Number of iterations   = ~a~n"
        (ode-evolve-count evolve))
(printf "Number of failed steps = ~a~n"
        (ode-evolve-failed-steps evolve))
(printf "~a~n" (plot (points (reverse y0-values))
                     (x-min 0.0)
                     (x-max 100.0)
                     (y-min -2.0)
                     (y-max 2.0)))
```

```
    (printf "~a~n" (plot (points (reverse y1-values))
                         (x-min 0.0)
                         (x-max 100.0)))))
```

When run, it prints the following:

```
Number of iterations   = 84575
Number of failed steps = 352
```

Figures 10.3 and 10.4 show the resulting output plots of y0 and y1.

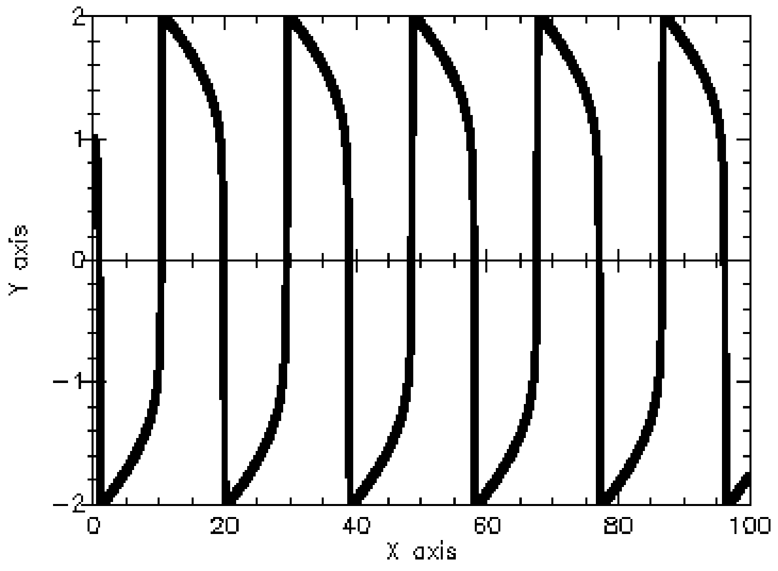

Figure 10.3: ODE Example 2 Plot of y0

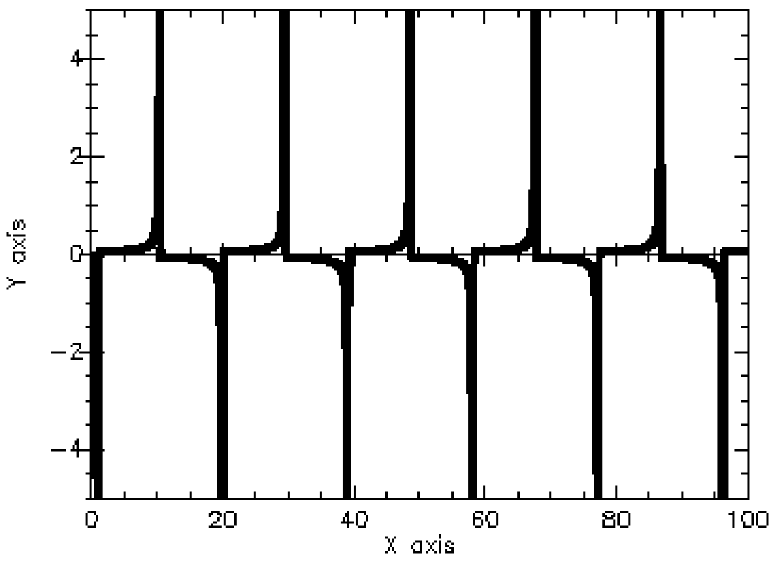

Figure 10.4: ODE Example 2 Plot of y1

Chapter 11

Chebyshev Approximations

This chapter describes the routines for computing Chebyshev approximations to univariate functions provided by the PLT Scheme Science Collection. A Chebyshev approximation is a trucvation of the series

$$f(x) = \sum c_n T_n(x)$$

where the Chebyshev polymonials $T_n(x) = \cos(n \arccos x)$ provides an orthogonal basis of polynomials in the interval $[-1, 1]$ with the weight function $1/\sqrt{1-x^2}$. The first few Chebyshev ploymonials are, $T_0(x) = 1$, $T_1(x) = x$, $T_2(x) = 2x^2 - 1$. For more information see Abramowitz and Stegan, Chapter 22.

The functions described in this chapter are defined in the `chebyshev.ss` file in the science collection and are made available using the form:

```
(require (planet "chebyshev.ss" ("williams" "science.plt" 2 0)))
```

11.1 The `chebyshev-series` Structure

`chebyshev-series`

```
Structure:  chebyshev-series
Contract:   (struct chebyshev-series
                    ((coefficient (vectorof real?))
                     (order natural-number?)
                     (lower real?)
                     (upper real?)))
```

This structure defines a Chebyshev Series.

`coefficients` – a vector of length *order* containing the coefficients for the Chebyshev series.

`order` – The order of the Chebyshev series.

`lower` – The lower bound on the interval over which the Chebyshev series is defined.

`upper` – The upper bound on the interval over which the Chebyshev series is defined.

The approximations are made over the range [*lower*, *upper*] using *order* + 1 terms, including *coefficient*[0]. The series is computed using the following convention,

$$f(x) = (c_0/2) + \sum_{n=1} c_n T_n(x)$$

which is needed when accessing the coefficients directly.

11.2 Creation and Calculation of Chebyshev Series

`make-chebyshev-series-order`

Function: `(make-chebyshev-series-order order)`
Contract: `(-> natural-number? chebyshev-series?)`

This function returns a newly created Chebyshev series with the given *order*.

`chebyshev-series-init`

Function: `(chebyshev-series-init cs func a b)`
Contract: `(-> chebyshev-series? procedure? real? real? void?)`

This function computes the Chebyshev approximation *cs* for the function *func* over the range (a, b) to the previously specified order. The computation of the Chebyshev approximation is an O(n^2) process and requires n function evaluations.

11.3 Chebyshev Series Evaluation

`chebyshev-eval`

Function: `(chebyshev-eval cs x)`
Contract: `(-> chebyshev-series? real? real?)`

This function evaluates the Chebyshev series *cs* at the given point x.

`chebyshev-eval-n`

Function: `(chebyshev-eval-n cs n x)`
Contract: `(-> chebyshev-series? natural-number? real? real?)`

This function evaluates the Chebyshev series *cs* at the given point x to (at most) the given order n.

11.4 Examples

Example: The following program computes Chebyshev approximations to a step function. This is an extremely difficult approximation to make, due to the discontinuity, and was chosed as an example where approximation error is visible. For smooth functions the Chebyshev approximation converges extremely rapidly and errors would not be visible.

```
(require (planet "chebyshev.ss" ("williams" "science.plt" 2 0)))
(require (lib "plot.ss" "plot"))

(define (f x)
  (if (< x 0.5) .25 .75))

(define (chebyshev-example n)
  (let ((cs (make-chebyshev-series-order 40))
        (y-values '())
        (y-cs-10-values '())
        (y-cs-40-values '()))
    (chebyshev-series-init cs f 0.0 1.0)
    (do ((i 0 (+ i 1)))
        ((= i n) (void))
      (let* ((x (exact->inexact (/ i n)))
             (y (f x))
             (y-cs-10 (chebyshev-eval-n cs 10 x))
             (y-cs-40 (chebyshev-eval cs x)))
        ;(printf "~a ~a ~a ~a\n"
        ;        x y y-cs-10 y-cs-40)
        (set! y-values (cons (vector x y) y-values))
        (set! y-cs-10-values
              (cons (vector x y-cs-10) y-cs-10-values))
        (set! y-cs-40-values
              (cons (vector x y-cs-40) y-cs-40-values))))
    (printf "~a~n" (plot (mix (points (reverse y-values))
                              (points (reverse y-cs-10-values)))
                         (x-min 0) (x-max 1)
                         (y-min 0) (y-max 1)
                         (title "Chebyshev Series Order 10")))
    (printf "~a~n" (plot (mix (points (reverse y-values))
                              (points (reverse y-cs-40-values)))
                         (x-min 0) (x-max 1)
                         (y-min 0) (y-max 1)
                         (title "Chebyshev Series Order 40")))))

(chebyshev-example 100)
```

Figures 11.1 and 11.2 show the resulting output plots.

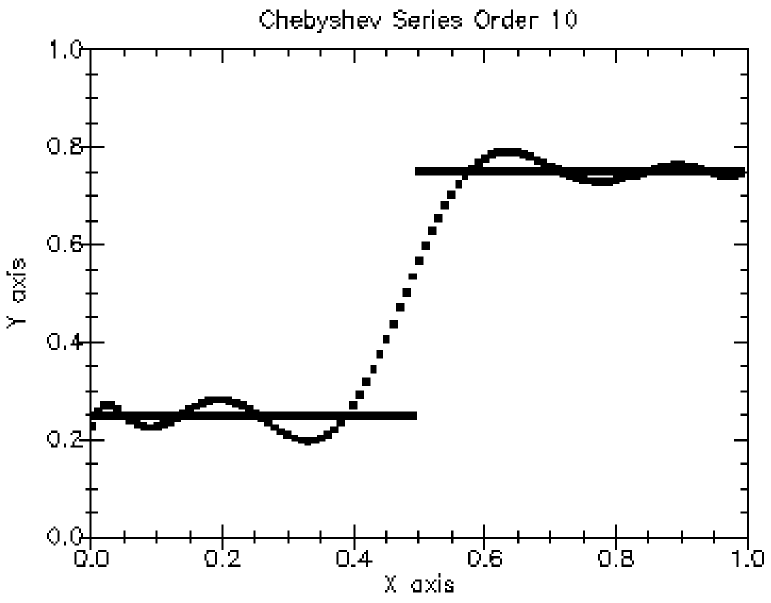

Figure 11.1: Chebyshev Series Order 10

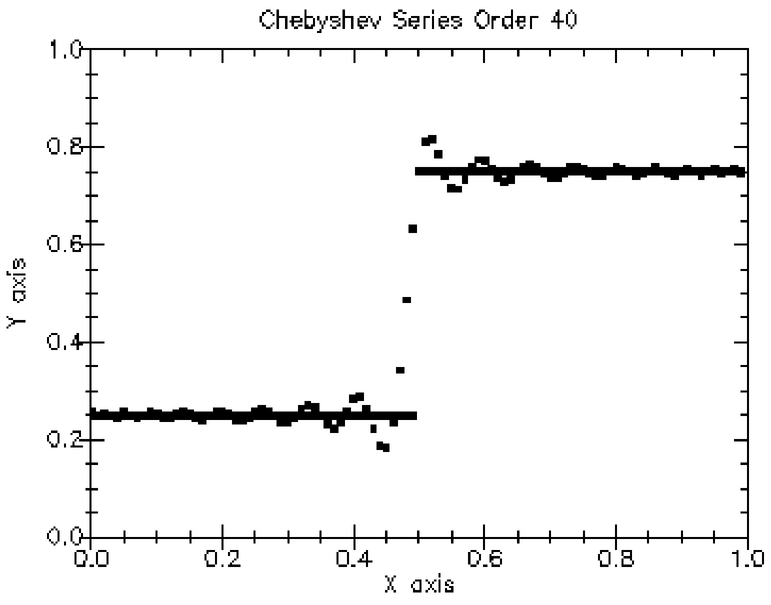

Figure 11.2: Chebyshev Series Order 40

Appendix A

GNU Lesser General Public License (LGPL)

it in new free programs; and that you are informed that you can do these things.

To protect your rights, we need to make restrictions that forbid distributors to deny you these rights or to ask you to surrender these rights. These restrictions translate to certain responsibilities for you if you distribute copies of the library or if you modify it.

For example, if you distribute copies of the library, whether gratis or for a fee, you must give the recipients all the rights that we gave you. You must make sure that they, too, receive or can get the source code. If you link other code with the library, you must provide complete object files to the recipients, so that they can relink them with the library after making changes to the library and recompiling it. And you must show them these terms so they know their rights.

We protect your rights with a two-step method: (1) we copyright the library, and (2) we offer you this license, which gives you legal permission to copy, distribute and/or modify the library.

To protect each distributor, we want to make it very clear that there is no warranty for the free library. Also, if the library is modified by someone else and passed on, the recipients should know that what they have is not the original version, so that the original author's reputation will not be affected by problems that might be introduced by others.

Finally, software patents pose a constant threat to the existence of any free program. We wish to make sure that a company cannot effectively restrict the users of a free program by obtaining a restrictive license from a patent holder. Therefore, we insist that any patent license obtained for a version of the library must be consistent with the full freedom of use specified in this license.

Most GNU software, including some libraries, is covered by the ordinary GNU General Public License. This license, the GNU Lesser General Public License, applies to certain designated libraries, and is quite different from the ordinary General Public License. We use this license for certain libraries in order to permit linking those libraries into non-free programs.

When a program is linked with a library, whether statically or using a shared library, the combination of the two is legally speaking a combined work, a derivative of the original library. The ordinary General Public License therefore permits such linking only if the entire combination fits its criteria of freedom. The Lesser General Public License permits more lax criteria for linking other code with the library.

We call this license the "Lesser" General Public License because it does Less to protect the user's freedom than the ordinary General Public License. It also provides other free software developers Less of an advantage over competing non-free programs. These disadvantages are the reason we use the ordinary General Public License for many libraries. However, the Lesser license provides advantages in certain special circumstances.

For example, on rare occasions, there may be a special need to encourage the widest possible use of a certain library, so that it becomes a de-facto standard. To achieve this, non-free programs must be allowed to use the library. A more frequent case is that a free library does the same job as widely used non-free libraries. In this case, there is little to gain by limiting the free library to free software only, so we use the Lesser General Public License.

In other cases, permission to use a particular library in non-free programs enables a greater number of people to use a large body of free software. For example, permission to use the GNU C Library in non-free programs enables many more people to use the whole GNU operating system, as well as its variant, the GNU/Linux operating system.

Although the Lesser General Public License is Less protective of the users' freedom, it does ensure that the user of a program that is linked with the Library has the freedom and the wherewithal to run that program using a modified version of the Library.

The precise terms and conditions for copying, distribution and modification follow. Pay close attention to the difference between a "work based on the library" and a "work that uses the library". The former contains code derived from the library, whereas the latter must be combined with the library in order to run.

GNU LESSER GENERAL PUBLIC LICENSE
TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License Agreement applies to any software library or other program which contains a notice placed by the copyright holder or other authorized party saying it may be distributed under the terms of this Lesser General Public License (also called "this License"). Each licensee is addressed as "you".

A "library" means a collection of software functions and/or data prepared so as to be conveniently linked with application programs (which use some of those functions and data) to form executables.

The "Library", below, refers to any such software library or work which has been distributed under these terms. A "work based on the Library" means either the Library or any derivative work under copyright law: that is to say, a work containing the Library or a portion of it, either verbatim or with modifications and/or translated straightforwardly into another language. (Hereinafter, translation is included without limitation in the term "modification".)

"Source code" for a work means the preferred form of the work for making modifications to it. For a library, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the library.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running a program using the Library is not restricted, and output from such a program is covered only if its contents constitute a work based on the Library (independent of the use of the Library in a tool for writing it). Whether that is true depends on what the Library does and what the program that uses the Library does.

1. You may copy and distribute verbatim copies of the Library's complete source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and distribute a copy of this License along with the Library.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Library or any portion of it, thus forming a work based on the Library, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) The modified work must itself be a software library.

b) You must cause the files modified to carry prominent notices stating that you changed the files and the date of any change.

c) You must cause the whole of the work to be licensed at no charge to all third parties under the terms of this License.

d) If a facility in the modified Library refers to a function or a table of data to be supplied by an application program that uses the facility, other than as an argument passed when the facility is invoked, then you must make a good faith effort to ensure that, in the event an application does not supply such function or table, the facility still operates, and performs whatever part of its purpose remains meaningful.

```
    (For example, a function in a library to compute square roots has
    a purpose that is entirely well-defined independent of the
    application.  Therefore, Subsection 2d requires that any
    application-supplied function or table used by this function must
    be optional: if the application does not supply it, the square
    root function must still compute square roots.)

These requirements apply to the modified work as a whole.  If
identifiable sections of that work are not derived from the Library,
and can be reasonably considered independent and separate works in
themselves, then this License, and its terms, do not apply to those
sections when you distribute them as separate works.  But when you
distribute the same sections as part of a whole which is a work based
on the Library, the distribution of the whole must be on the terms of
this License, whose permissions for other licensees extend to the
entire whole, and thus to each and every part regardless of who wrote
it.

Thus, it is not the intent of this section to claim rights or contest
your rights to work written entirely by you; rather, the intent is to
exercise the right to control the distribution of derivative or
collective works based on the Library.

In addition, mere aggregation of another work not based on the Library
with the Library (or with a work based on the Library) on a volume of
a storage or distribution medium does not bring the other work under
the scope of this License.

  3. You may opt to apply the terms of the ordinary GNU General Public
License instead of this License to a given copy of the Library.  To do
this, you must alter all the notices that refer to this License, so
that they refer to the ordinary GNU General Public License, version 2,
instead of to this License.  (If a newer version than version 2 of the
ordinary GNU General Public License has appeared, then you can specify
that version instead if you wish.)  Do not make any other change in
these notices.

  Once this change is made in a given copy, it is irreversible for
that copy, so the ordinary GNU General Public License applies to all
subsequent copies and derivative works made from that copy.

  This option is useful when you wish to copy part of the code of
the Library into a program that is not a library.

  4. You may copy and distribute the Library (or a portion or
derivative of it, under Section 2) in object code or executable form
under the terms of Sections 1 and 2 above provided that you accompany
it with the complete corresponding machine-readable source code, which
must be distributed under the terms of Sections 1 and 2 above on a
```

```
medium customarily used for software interchange.

  If distribution of object code is made by offering access to copy
from a designated place, then offering equivalent access to copy the
source code from the same place satisfies the requirement to
distribute the source code, even though third parties are not
compelled to copy the source along with the object code.

  5. A program that contains no derivative of any portion of the
Library, but is designed to work with the Library by being compiled or
linked with it, is called a "work that uses the Library".  Such a
work, in isolation, is not a derivative work of the Library, and
therefore falls outside the scope of this License.

  However, linking a "work that uses the Library" with the Library
creates an executable that is a derivative of the Library (because it
contains portions of the Library), rather than a "work that uses the
library".  The executable is therefore covered by this License.
Section 6 states terms for distribution of such executables.

  When a "work that uses the Library" uses material from a header file
that is part of the Library, the object code for the work may be a
derivative work of the Library even though the source code is not.
Whether this is true is especially significant if the work can be
linked without the Library, or if the work is itself a library.  The
threshold for this to be true is not precisely defined by law.

  If such an object file uses only numerical parameters, data
structure layouts and accessors, and small macros and small inline
functions (ten lines or less in length), then the use of the object
file is unrestricted, regardless of whether it is legally a derivative
work.  (Executables containing this object code plus portions of the
Library will still fall under Section 6.)

  Otherwise, if the work is a derivative of the Library, you may
distribute the object code for the work under the terms of Section 6.
Any executables containing that work also fall under Section 6,
whether or not they are linked directly with the Library itself.

  6. As an exception to the Sections above, you may also combine or
link a "work that uses the Library" with the Library to produce a
work containing portions of the Library, and distribute that work
under terms of your choice, provided that the terms permit
modification of the work for the customer's own use and reverse
engineering for debugging such modifications.

  You must give prominent notice with each copy of the work that the
Library is used in it and that the Library and its use are covered by
this License.  You must supply a copy of this License.  If the work
during execution displays copyright notices, you must include the
```

copyright notice for the Library among them, as well as a reference directing the user to the copy of this License. Also, you must do one of these things:

a) Accompany the work with the complete corresponding machine-readable source code for the Library including whatever changes were used in the work (which must be distributed under Sections 1 and 2 above); and, if the work is an executable linked with the Library, with the complete machine-readable "work that uses the Library", as object code and/or source code, so that the user can modify the Library and then relink to produce a modified executable containing the modified Library. (It is understood that the user who changes the contents of definitions files in the Library will not necessarily be able to recompile the application to use the modified definitions.)

b) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (1) uses at run time a copy of the library already present on the user's computer system, rather than copying library functions into the executable, and (2) will operate properly with a modified version of the library, if the user installs one, as long as the modified version is interface-compatible with the version that the work was made with.

c) Accompany the work with a written offer, valid for at least three years, to give the same user the materials specified in Subsection 6a, above, for a charge no more than the cost of performing this distribution.

d) If distribution of the work is made by offering access to copy from a designated place, offer equivalent access to copy the above specified materials from the same place.

e) Verify that the user has already received a copy of these materials or that you have already sent this user a copy.

For an executable, the required form of the "work that uses the Library" must include any data and utility programs needed for reproducing the executable from it. However, as a special exception, the materials to be distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

It may happen that this requirement contradicts the license restrictions of other proprietary libraries that do not normally accompany the operating system. Such a contradiction means you cannot use both them and the Library together in an executable that you distribute.

7. You may place library facilities that are a work based on the Library side-by-side in a single library together with other library facilities not covered by this License, and distribute such a combined library, provided that the separate distribution of the work based on the Library and of the other library facilities is otherwise permitted, and provided that you do these two things:

a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities. This must be distributed under the terms of the Sections above.

b) Give prominent notice with the combined library of the fact that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

8. You may not copy, modify, sublicense, link with, or distribute the Library except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, link with, or distribute the Library is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

9. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Library or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Library (or any work based on the Library), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Library or works based on it.

10. Each time you redistribute the Library (or any work based on the Library), the recipient automatically receives a license from the original licensor to copy, distribute, link with or modify the Library subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties with this License.

11. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Library at all. For example, if a patent

license would not permit royalty-free redistribution of the Library by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Library.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply, and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

12. If the distribution and/or use of the Library is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Library under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

13. The Free Software Foundation may publish revised and/or new versions of the Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Library does not specify a license version number, you may choose any version ever published by the Free Software Foundation.

14. If you wish to incorporate parts of the Library into other free programs whose distribution conditions are incompatible with these, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status

Appendix B

SRFI 27: Sources of Random Bits

Title

SRFI 27: Sources of Random Bits

Author

Sabastan Egner

Status

This SRFI is currently in “final” status. To see an explanation of each status that a SRFI can hold, see here. You can access previous messages via the archive of the mailing list.

- Draft: 2002/02/12-2002/04/12
- Revised: 2002/04/04
- Revised: 2002/04/10
- Revised: 2002/04/10
- Final: 2002/06/03

Abstract

This document specifies an interface to sources of random bits, or ”random sources” for brevity. In particular, there are three different ways to use the interface, with varying demands on the quality of the source and the amout of control over the production process:

- The ”no fuss” interface specifies that (`random-source` n) produces the next random integer in $\{0, ..., n-1\}$ and (`random-real`) produces the next random real number between zero and one. The details of how these random values are

produced may not be very relevant, as long as they appear to be sufficiently random.

- For simulation purposes, on the contrary, it is usually necessary to know that the numbers are produced deterministically by a pseudo random number generator of high quality and to have explicit access to its state. In addition, one might want to use several independent sources of random numbers at the same time and it can be useful to have some simple form of randomization.
- For security applications a serious form of true randomization is essential, in the sense that it is difficult for an adversary to exploit or introduce imperfections into the distribution of random bits. Moreover, the linear complexity of the stream of random bits is more important than its statistical properties. In these applications, an entropy source (producing truely random bits at a low rate) is used to randomize a pseudo random number generator to increase the rate of available bits.

Once random sources provide the infrastructure to obtain random bits, these can be used to construct other random deviates. Most important are floating point numbers of various distributions and random discrete structures, such as permutations or graphs. As there is an essentially unlimited number of such objects (with limited use elsewhere), we do not include them in this SRFI. In other words, this SRFI is *not* about making all sorts of random objects—it is about obtaining random bits in a portable, flexible, reliable, and efficient way.

Rationale

This SRFI defines an interface for sources of random bits computed by a pseudo random number generator. The interface provides range-limited integer and real numbers. It allows accessing the state of the underlying generator. Moreover, it is possible to obtain a large number of independent generators and to invoke a mild form of true randomization.

The design aims at sufficient flexibility to cover the usage patterns of many applications as diverse as discrete structures, numerical simulations, and cryptographic protocols. At the same time, the interface aims at simplicity, which is important for occasional use. As there is no "one size fits all" random number generator, the design necessarily represents some form of compromise between the needs of the various applications.

Although strictly speaking not part of the specification, the emphasis of this proposal is on *high quality* random numbers and on *high performance*. As the state of the art in pseudo random number generators is still advancing considerably, the choice of method for the reference implementation should essentially be considered preliminary.

Specification

(`random-integer` n) $\rightarrow x$

The next integer x in $\{0, ..., n-1\}$ obtained from `default-random-source`. Subsequent results of this procedure appear to be independent uniformly distributed over the range $\{0, ..., n-1\}$. The argument n must be a positive integer, otherwise an error is signalled.

(`random-real`) *to x*

The next number $0 < x < 1$ obtained from `default-random-source`. Subsequent results of this procedure appear to be independent uniformly distributed. The numerical type of the results and the quantization of the output range depend on the implementation; refer to `random-source-make-reals` for details.

`default-random-source`

A random source from which `random-integer` and `random-real` have been derived using `random-source-make-integers` and `random-source-make-reals`. Note that an assignment to `default-random-source` does not change `random` or `random-real`; it is also strongly recommended not to assign a new value.

(`make-random-source`) *to s*

Creates a new random source s. Implementations may accept additional, optional arguments in order to create different types of random sources. A random source created with `make-random-source` represents a deterministic stream of random bits generated by some form of pseudo random number generator. Each random source obtained as (`make-random-source`) generates the same stream of values, unless the state is modified with one of the procedures below.

(`random-source?` *obj*) $\rightarrow$ *bool*

Tests if *obj* is a random source. Objects of type random source are distinct from all other types of objects.

(`random-source-state-ref` *s*) *to state*
(`random-source-state-set!` *s state*)

Get and set the current state of a random source s. The structure of the object state depends on the implementation; the only portable use of it is as argument to `random-source-state-set!`. It is, however, required that a state possess an external representation.

(`random-source-randomize!` *s*)

Makes an effort to set the state of the random source s to a truly random state. The actual quality of this randomization depends on the implementation but it can at least be assumed that the procedure sets s to a different state for each subsequent run of the Scheme system.

(`random-source-pseudo-randomize!` *s i j*)

Changes the state of the random source s into the initial state of the $(i, j)^t h$ independent random source, where i and j are non-negative integers. This procedure provides a mechanism to obtain a large number of independent random sources (usually all derived from the same backbone generator), indexed by two integers. In contrast to `random-source-randomize!`, this procedure is entirely deterministic.

(`random-source-make-integers` *s*) $\rightarrow$ *rand*

Obtains a procedure *rand* to generate random integers using the random source s. *Rand* takes a single argument n, which must be a positive integer, and returns the next uniformly distributed random integer from the interval $\{0, ..., n-1\}$ by advancing the state of the source s.

If an application obtains and uses several generators for the same random source s, a call to any of these generators advances the state of s. Hence, the generators *do not* produce the same sequence of random integers each but rather share a state. This also holds for all other types of generators derived from a fixed random sources. Implementations that support concurrency make sure that the state of a generator is properly advanced.

(`random-source-make-reals` *s*) $\rightarrow$ *rand*
(`random-source-make-reals` *s unit*) $\rightarrow$ *rand*

Obtains a procedure *rand* to generate random real numbers $0 < x < 1$ using the random source s. The procedure *rand* is called without arguments.

The optional parameter *unit* determines the type of numbers being produced by *rand* and the quantization of the output. *Unit* must be a number such that $0 < unit < 1$. The numbers created by *rand* are of the same numerical type as *unit* and the potential output values are spaced by at most *unit*. One can imagine *rand* to create numbers as $x * unit$ where x is a random integer in $\{1, ..., floor(1/unit) - 1\}$. Note, however, that this need not be the way the values are actually created and that the actual resolution of *rand* can be much higher than *unit*. In case *unit* is absent it defaults to a reasonably small value (related to the width of the mantissa of an efficient number format).

Design Rationale

Why not combine `random-integer` and `random-real`?

The two procedures are not combined into a single variable-arity procedures to save a little time and space during execution. Although some Scheme systems can deal with variable arity as efficiently as with fixed arity this is not always the case and time efficiency is very important here.

Why not some object-oriented interface?

There are many alternatives to the interface as specified in this SRFI. In particular, every framework for object-orientation can be used to define a class for random sources and specify the interface for the methods on random sources. However, as the object-oriented frameworks differ considerably in terms of syntax and functionality, this SRFI does not make use of any particular framework.

Why is there not just a generator with a fixed range?

A bare fixed-range generator is of very limited use. Nearly every application has to add some functionality to make use of the random numbers. The most fundamental task in manipulating random numbers is to change the range and quantization. This is exactly what is provided by `random-integer` and `random-real`. In addition, is saves the user from the pitfall of changing the range with a simple `modulo`-computation which may substantially reduce the quality of the numbers being produced.

The design of the interface is based on three prototype applications:

1. Repeatedly choose from relatively small sets: As the size of the set is likely to vary from call to call, `random-integer` accepts a range argument n in every call. The implementation should try to avoid boxing/unboxing of values if the ranges fit into immediate integers.
2. Generate a few large integers with a fixed number of bits: As generating the random number itself is expensive, passing the range argument in every call does not hurt performance. Hence, the same interface as in the first application can be used.
3. Generate real numbers: Unlike the choose-from-set case, the range and the quantization is constant over a potentially very large number of calls. In addition, there are usually just a few distinct instances of quantization and number type, most likely corresponding to underlying `float` and `double` representations. Therefore, `random-real` does not accept any parameters but the procedure `random-source-make-reals` creates a properly configured `random-real` procedure.

Why bother about floating point numbers at all?

A proper floating point implementation of a random number generator is potentially much more efficient that an integer implementation because it can use more powerful arithmetics hardware. If in addition the application needs floating point random numbers it would be an intolerable waste to run an integer generator to produce floating point random numbers. A secondary reason is to save the user from the 'not as easy as it seems' task of converting an integer generator into a real generator.

Why are zero and one excluded from `random-real`?

The procedure random-real does not return $x = 0$ or $x = 1$ in order to allow `(log x)` and `(log (- 1 x))` without the danger of a numerical exception.

Implementatiom

Choice of generator

The most important decision about the implementation is the choice of the random number generator. The basic principle here is: *Let quality prevail!* In the end, a performance penalty of a better generator may be a cheap price to pay for some avoided catastrophes. It may be unexpected, but I have also seen many examples where the better generator was also the faster. Simple linear congruential generator cannot be recommended as they tend to be ill-behaved in several ways. For this reason, my initial proposal was George Marsaglia's COMBO generator, which is the combination of a 32-bit multiplicative lagged Fibonacci-generator with a 16-bit multiply with carry generator. The COMBO generator passes all tests of Marsaglia's DIEHARD testsuite for random number generators and has a period of order 2^60.

As an improvement, Brad Lucier suggested suggested Pierre L'Ecuyer's MRG32k3a generator which is combination of two recursive generators of degree three, both of which fit into 54-bit arithmetics. The MRG32k3a generator also passes DIEHARD and in addition, has desireable spectral properties and a period in the order of 2^191. As a matter of fact, multiple recursive generators (MRGs) are theoretically much better understood than special constructions as the COMBO generator. This is the reason why the implementations provided here implements the MRG32k3a generator. When implemented in floating point arithmetics with sufficient mantissa-width, this generator is also very fast.

Choice of arithmetics

The next important decision about the implementation is the type of arithmetics to be used. The choice is difficult and depends heavily on the underlying Scheme system and even on the underlying hardware platform and architecture. For the MRG32k3a generator, use 64-bit arithmetics if you really have it. If not, use a floating point ALU if it gives you 54 or more bits of mantissa. And if you do not have floats either, then at least try to make sure you work with immediate integers (instead of allocated objects). Unfortunately, there is no portable way in Scheme to find out about native and emulated arithmetics.

As performance is critical to many applications, one might want to implement the actual generator itself in native code. For this reason, I provide three different implementations of the backbone generator as a source of inspiration. See the code below.

Data Type for Random Sources

An important aspect of the specification in this SRFI is that random sources are objects of a distinct type. Although this is straight-forward and available in nearly every Scheme implementation, there is no portable way to do this at present. One way to define the record type is to use SRFI-9.

The reference implementations below define a record type to contain the exported procedures. The actual state of the generator is stored in the binding time environment of `make-random-source`. This has the advantage that access to the state is fast even if the record type would be slow (which need not be the case).

Entropy Source for Randomization

Another problematic part of the specification with respect to portability is `random-source-randomize!` as it needs access to a real entropy source. A reasonable choice for such as source is to use the system clock in order to obtain a value for randomization, for example in the way John David Stone recommends (see reference below). This is good enough for most applications with the notable exception of security related programs. One way to obtain the time in Scheme is to use SRFI-19.

Implementation of the specified interface

Once the portability issues are resolved, one can provide the remaining functionality as specified in this SRFI document. For the reference implementation, a relatively large part of the code deals with the more advanced features of the MRG32k3a generator, in particular `random-source-pseudo-randomize!`. This code is inspired by Pierre L'Ecuyer's own implementation of the MRG32k3a generator.

Another part of this generic code deals with changing the range and quantization of the random numbers and with error checking to detect common mistakes and abuses.

Implementation Examples

Here are three alternative implementations of the SRFI. (Here are all files, tar-gzipped, 13020 bytes.) Keep in mind that a SRFI is a "request for implementation", which means these implementations are merely examples to illustrate the specification and inspire people to implement it better and faster. The performance figures below are rough indications measured on a Pentium3, 800 Mhz, Linux; x int/s, y real/s means `(random-integer 2)` can be computed about x times a second and `(random-real)` about y times a second. The implementations are

1. for Scheme 48 0.57, using 54-bit `integer` only. This implementation aims at portability, not at performance (30000 ints/s, 3000/s reals/s).
2. for Scheme 48 0.57 with the core generator being implemented in C using `(double)`-arithmetics. The generator is made available in Scheme 48 via the C/Scheme interface. The performance of this generator is good (160000 ints/s, 180000 reals/s).
3. for Gambit 3.0, using `flonum` and 54-bit `integer`. This code is inspired by a program by Brad Lucier as posted to the discussion archive of this SRFI. The performance of this generator is good when compiled (5000 ints/s, 25000/s reals/s when interpreted, 200000 ints/s, 400000/s reals/s when compiled; see acknowledgements).

In addition to the implementations there is a small collection of confidence tests for the interface specified. The tests merely check a few assertions expressed by the specification. It is not the intention to provide a complete test of the interface here. It is even less the intention to provide statistical tests of the generator itself. However, there is a function to write random bits from the generators to a file in a way readable by the *DIEHARD* testsuite. This makes it easier for implementors to find out about their favorite generators and check their implementation.

Recommended Usage Patterns

Unless the functionality defined in this SRFI is sufficient, an application has to implement more procedures to construct other random deviates. This section contains some recommendation on how to do this technically by presenting examples of increasing difficulty with respect to the interface. Note that the code below is not part of the specification, it is merely meant to illustrate the spirit.

Generating Random Permutations

The following code defines procedures to generate random permutations of the set $\{0, ..., n-1\}$. Such a permutation is represented by a `vector` of length n for the images of the points.

Observe that the implementation first defines the procedure `random-source-make-permutations` to turn a random source s into a procedure to generate permutations of given degree n. In a second step, this is applied to the default source to define a ready-to-use procedure for permutations: (`random-permutation` n) constructs a random permutation of degree n.

```
(define (random-source-make-permutations s)
  (let ((rand (random-source-make-integers s)))
    (lambda (n)
      (let ((x (make-vector n 0)))
(do ((i 0 (+ i 1)))
    ((= i n))
  (vector-set! x i i))
(do ((k n (- k 1)))
    ((= k 1) x)
  (let* ((i (- k 1))
 (j (rand k))
 (xi (vector-ref x i))
 (xj (vector-ref x j)))
    (vector-set! x i xj)
    (vector-set! x j xi)))))))

(define random-permutation
  (random-source-make-permutations default-random-source))
```

For the algorithm refer to Knuth's "The Art of Computer Programming", Vol. II, 2nd ed., Algorithm P of Section 3.4.2.

Generating Exponentially-Distributed Random Numbers

The following code defines procedures to generate exponentially Exp(mu)-distributed random numbers. The technical difficulty of the interface addressed here is how to pass optional arguments to `random-source-make-reals`.

```
(define (random-source-make-exponentials s . unit)
  (let ((rand (apply random-source-make-reals s unit)))
    (lambda (mu)
      (- (* mu (log (rand)))))))

(define random-exponential
  (random-source-make-exponentials default-random-source))
```

The algorithm is folklore. Refer to Knuth's "The Art of Computer Programming", Vol. II, 2nd ed., Section 3.4.1.D.

Generating Normally-Distributed Random Numbers

The following code defines procedures to generate normal N(mu, sigma)-distributed real numbers using the polar method. The technical difficulty of the interface addressed here is that the polar method generates two results per computation. We return one of the result and store the second one to be returned by the next call to the procedure. Note that this implies that `random-source-state-set!` (and the other procedures modifying the state) does not necessarily affect the output of `random-normal` immediately!

```
(define (random-source-make-normals s . unit)
  (let ((rand (apply random-source-make-reals s unit))
(next #f))
```

```
    (lambda (mu sigma)
      (if next
  (let ((result next))
    (set! next #f)
    (+ mu (* sigma result)))
  (let loop ()
    (let* ((v1 (- (* 2 (rand)) 1))
   (v2 (- (* 2 (rand)) 1))
   (s (+ (* v1 v1) (* v2 v2))))
      (if (>= s 1)
  (loop)
  (let ((scale (sqrt (/ (* -2 (log s)) s))))
    (set! next (* scale v2))
    (+ mu (* sigma scale v1))))))))))

(define random-normal
  (random-source-make-normals default-random-source))
```

For the algorithm refer to Knuth's "The Art of Computer Programming", Vol. II, 2nd ed., Algorithm P of Section 3.4.1.C.

Acknowledgements

I would like to thank all people who have participated in the discussion, in particular Brad Lucier and Pierre l'Ecuyer. Their contributions have greatly improved the design of this SRFI. Moreover, Brad has optimized the Gambit implementation quite substantially.

References

1. G. Marsaglia: Diehard – Testsuite for Random Number Generators. stat.fsu.edu/~geo/diehard.html (Also contains some enerators that do pass Diehard.)
2. D. E. Knuth: The Art of Computer Programming; Volume II Seminumerical Algorithms. 2nd ed. Addison-Wesley, 1981. (The famous chapter on random number generators.)
3. P. L'Ecuyer: "Software for Uniform Random Number Generation: Distinguishing the Good and the Bad", Proceedings of the 2001 Winter Simulation Conference, IEEE Press, Dec. 2001, 95–105. www.iro.umontreal.ca/~lecuyer/myftp /papers/wsc01rng.pdf (Profound discussion of random number generators.)
4. P. L'Ecuyer: "Good Parameter Sets for Combined Multiple Recursive Random Number Generators", Shorter version in Operations Research, 47, 1 (1999), 159–164. www.iro.umontreal.ca/~lecuyer/myftp/papers/combmrg2.ps (Actual numbers for good generators.)
5. P. L'Ecuyer: "Software for Uniform Random Number Generation: Distinguishing the Good and the Bad", Proceedings of the 2001 Winter Simulation Conference, IEEE Press, Dec. 2001, 95–105.
6. MIT Scheme v7.6: random flo:random-unit *random-state* make-random-state random-state? `http://www.swiss.ai.mit.edu/projects/scheme/documenta`

tion/scheme_5.html#SEC53 (A mechanism to run a fixed unspecified generator.)

7. A. Jaffer: SLIB 2d2 with (require 'random): random *random-state* copy-random-state seed-¿random-state make-random-state random:uniform random: exp random:normal-vector! random-hollow-sphere! random:solid-sphere! http://www.swiss.ai.mit.edu/$\sim$jaffer/slib_4.html#SEC92 (Based on the MIT Scheme mechanism.)
8. R. Kelsey, J. Rees: Scheme 48 v0.57 'random.scm': make-random make-random-vector (Internal procedures of Scheme48; a fixed 28-bit generator.)
9. M. Flatt: PLT MzScheme Version 200alpha1: random random-seed current-pseudo-random-generator make-pseudo-random-generator pseudo-random-generator? http://download.plt-scheme.org/doc/200alpha1/html/mzscheme/mzscheme-Z-H-3.html#_idx_144 (A mechanism to run a generator and to exchange the generator.)
10. H. Abelson, G. J. Sussmann, J. Sussman: Structure and Interpretation of Computer Programs. http://mitpress.mit.edu/sicp/full-text/book/book-Z-H-20.html#_idx_2934 (The rand-example shows a textbook way to define a random number generator.)
11. John David Stone: A portable random-number generator. http://www.math.grin.edu/~stone/events/scheme-workshop/random.html (An implementation of a linear congruental generator in Scheme.)
12. Network Working Group: RFC1750: Randomness Recommendations for Security. http://www.cis.ohio-state.edu/htbin/rfc/rfc1750.html (A serious discussion of serious randomness for serious security.)
13. http://www.random.org/essay.html
14. http://www.taygeta.com/random/randrefs.html (Resources on random number generators and randomness.)

Copyright

Bibliography

[1] Free Software Foundation, *GNU Lesser General Public License*, Version 2.1, February 1999

[2] M. Galassi et al, *GNU Scientific Library Reference Manual (2nd Ed.)*, ISBN 0954161734

[3] M. Flatt et al, *PLT MzScheme: Reference Manual*, http://plt-scheme.org/

[4] M. Flatt et al, *PLT MzLib: Reference Manual*, http://plt-scheme.org/

[5] M. Flatt et al, *PLT MrEd: Graphical Toolbox Manual*, http://plt-scheme.org/

[6] *PLaneT Package Repository*, http://planet.plt-scheme.org/

[7] *PLoT Scheme*, http://plt-scheme.org/

[8] Williams, M. Douglas, *PLT Scheme Simulation Collection Reference Manual*

Index

absolute-deviation, 95
acosh, 12
asinh, 12
atanh, 12

bernoulli-cdf, 78
bernoulli-pdf, 78
bernoulli-plot, 79
beta-pdf, 38
beta-plot, 38
binomial-pdf, 81
binomial-plot, 81
bivariate-gaussian-pdf, 41
bivariate-gaussian-plot, 42

chebyshev-eval, 136
chebyshev-eval-n, 136
chebyshev-series, 135
chebyshev-series-init, 136
chi-squared-pdf, 44
chi-squared-plot, 44
choose, 21
covariance, 97
covariance-with-fixed-means, 97
current-random-source, 33

default-random-source, 31, 153
discrete-cdf, 92
discrete-histogram-accumulate!, 120
discrete-histogram-bins, 119
discrete-histogram-dynamic?, 119
discrete-histogram-get, 120
discrete-histogram-increment!, 120
discrete-histogram-max, 120
discrete-histogram-min, 120
discrete-histogram-n1, 119
discrete-histogram-n2, 119
discrete-histogram-plot, 121
discrete-histogram-plot-scaled, 121
discrete-histogram-sum, 120
discrete-histogram?, 117
discrete-pdf, 92
discrete-plot, 93
discrete?, 91
double-fact, 21

erf, 14
erfc, 15
eta, 29
eta-int, 28
expm1, 12
exponential-cdf, 46
exponential-pdf, 46
exponential-plot, 47

f-distribution-pdf, 49
f-distribution-plot, 50
factorial, 21
fcmp, 13
finite?, 11
flat-cdf, 52
flat-pdf, 52
flat-plot, 53
frexp, 12

gamma, 18
gamma-inv, 20
gamma-pdf, 54
gamma-plot, 55
gamma-xmax, 18
gammastar, 20
gaussian-cdf, 59
gaussian-pdf, 59
gaussian-plot, 60
gaussian-tail-pdf, 63
gaussian-tail-plot, 65
geometric-pdf, 84
geometric-plot, 85

hazard, 17
histogram-2d-accumulate!, 115

histogram-2d-bins, 114
histogram-2d-covariance, 116
histogram-2d-get, 115
histogram-2d-get-x-range, 115
histogram-2d-get-y-range, 115
histogram-2d-increment!, 114
histogram-2d-max, 115
histogram-2d-min, 115
histogram-2d-nx, 113
histogram-2d-ny, 113
histogram-2d-plot, 117
histogram-2d-sum, 116
histogram-2d-x-mean, 116
histogram-2d-x-ranges, 113
histogram-2d-x-sigma, 116
histogram-2d-y-mean, 116
histogram-2d-y-ranges, 114
histogram-2d-y-sigma, 116
histogram-2d?, 112
histogram-accumulate!, 109
histogram-bins, 108
histogram-get, 109
histogram-get-range, 109
histogram-increment!, 109
histogram-max, 109
histogram-mean, 109
histogram-min, 109
histogram-n, 108
histogram-plot, 110
histogram-plot-scaled, 110
histogram-ranges, 108
histogram-sigma, 110
histogram-sum, 110
histogram?, 107
hypot, 12
hzeta, 27

infinite?, 10

kurtosis, 96

lag-1-autocorrelation, 96
ldexp, 12
lnchoose, 22
lndouble-fact, 21
lnfact, 21
lngamma, 19
lngamma-sgn, 19
log1p, 11
logarithmic-pdf, 86
logarithmic-plot, 87
lognormal-cdf, 67
lognormal-pdf, 67
lognormal-plot, 68

make-chebyshev-series-order, 136
make-discrete, 91
make-discrete-histogram, 119
make-histogram, 107
make-histogram-2d, 113
make-histogram-2d-with-ranges-uniform, 113
make-histogram-with-ranges-uniform, 108
make-ode-evolve, 129
make-ode-step, 125
make-random-source, 31, 153
make-random-source-vector, 34
maximum, 101
maximum-index, 102
mean, 94
median-from-sorted-data, 102
minimum, 101
minimum-index, 102
minimum-maximum, 101
minimum-maximum-index, 102

nan?, 10

ode-control, 128
ode-control-h-adjust, 128
ode-control-init, 128
ode-control-name, 128
ode-control-scaled-new, 127
ode-control-standard-new, 126
ode-control-y-new, 127
ode-control-yp-new, 127
ode-evolve-apply, 129
ode-evolve-reset, 129
ode-step-apply, 126
ode-step-name, 125
ode-step-order, 125
ode-step-reset, 125
ode-step-rk2, 126
ode-step-rk4, 126
ode-step-rkf45, 126
ode-system, 124

pareto-cdf, 71
pareto-pdf, 70

pareto-plot, 71
poisson-pdf, 89
poisson-plot, 90
psi, 22
psi-1, 23
psi-1-int, 23
psi-1piy, 22
psi-int, 22
psi-n, 24

quantile-from-sorted-data, 102

random-bernoulli, 78
random-beta, 38
random-binomial, 81
random-bivariate-gaussian, 40
random-chi-squared, 42
random-discrete, 91
random-exponential, 46
random-f-distribution, 48
random-flat, 50
random-gamma, 53
random-gaussian, 56
random-gaussian-ratio-method, 59
random-gaussian-tail, 61
random-geometric, 83
random-integer, 30, 152
random-logarithmic, 86
random-lognormal, 67
random-pareto, 69
random-poisson, 88
random-real, 30, 152
random-source-make-integers, 32, 153
random-source-make-reals, 32, 154
random-source-pseudo-randomize!, 32, 153
random-source-randomize!, 31, 153
random-source-state, 34
random-source-state-ref, 31, 153
random-source-state-set!, 31
random-source?, 31, 153
random-t-distribution, 71
random-triangular, 75
random-uniform, 33
random-uniform-int, 33
random-unit-gaussian, 57
random-unit-gaussian-ratio-method, 59
random-unit-gaussian-tail, 62

set-histogram-2d-ranges!, 114
set-histogram-2d-ranges-uniform!, 114
set-histogram-ranges!, 108
set-histogram-ranges-uniform!, 108
set-random-source-state!, 34
sign, 13
skew, 95
standard-deviation, 95
standard-deviation-with-fixed-mean, 95

t-distribution-pdf, 72
t-distribution-plot, 73
triangular-cdf, 76
triangular-pdf, 75
triangular-plot, 77

unit-gaussian-pdf, 59, 60
unit-gaussian-plot, 60
unit-gaussian-tail-pdf, 64
unit-gaussian-tail-plot, 65

variance, 94
variance-with-fixed-mean, 95

weighted-absolute-deviation, 100
weighted-kurtosis, 101
weighted-mean, 98
weighted-skew, 100
weighted-standard-deviation, 99
weighted-standard-deviation-with-fixed-mean, 99
weighted-variance, 98
weighted-variance-with-fixed-mean, 99
with-new-random-source, 33
with-random-source, 33

zeta, 26
zeta-int, 26
zetam1, 27
zetam1-int, 26