

PLT Scheme Knowledge-Based Simulation

Dr. Doug Williams
m.douglas.williams@saic.com

September 27, 2004

Motivation

- To recreate a previously available knowledge-based simulation environment capability
 - Previous implementation in Symbolics Common LISP
 - Reimplement in PLT Scheme
 - Availability – free download (www.drscheme.org)
 - Portability – Windows, Linux, UNIX, Mac OS X
- Extend previous work
 - Provide a better mathematical framework
 - Implement a process-based model of simulation
 - Investigate a process interaction model for simulation
 - Implement an efficient rule-based inference engine
 - Provide a framework for advancing knowledge-based simulation

Components

- **PLT Scheme Science Collection**
 - Provides the mathematical and analysis framework
 - Previously part of the simulation collection, but provides functionality that is useful outside of simulations
 - Inspired by the GNU Scientific Library (GSL)
- **PLT Scheme Simulation Collection**
 - Provides a process-based, discrete-event simulation engine
 - Designed to facilitate component-based simulation models
 - Designed to facilitate distributed simulation models
- **PLT Scheme Inference Collection**
 - Provides a rule-based inference engine

PLT Scheme Science Collection

- Machine Constants
 - Primarily for internal implementation use
 - Floating-point min, max, epsilon and related constants
- Mathematical Constants and Functions
- Special Functions
- Random Number Generation
- Random Distributions
- Statistics
- Histograms
- Chebyshev Approximations

Mathematical Constants and Functions

| | |
|----------|--------------------------------------|
| e | The base of exponentials |
| log2e | The base-2 logarithm of 2 |
| log10e | The base-10 logarithm of e |
| sqrt2 | The square root of two |
| sqrt1/2 | The square root of one half |
| sqrt3 | The square root of three |
| pi | The constant pi |
| pi/2 | Pi divided by 2 |
| pi/4 | Pi divided by 4 |
| sqrtpi | The square root of pi |
| 2/sqrtpi | Two divided by the square root of pi |
| 1/pi | The reciprocal of pi |
| 2/pi | Twice the reciprocal of pi |
| ln10 | The natural log of 10 |
| ln2 | The natural log of 2 |
| lnpi | The natural log of pi |
| euler | Euler's constant |

- **Infinities and Not-a-Number**

- (nan? x)
- (infinite? x)
- (finite? x)

- **Elementary Functions**

- (log1p x)
- (expm1 x)
- (hypot x)
- (acosh x)
- (asinh x)
- (atanh x)
- (ldexp x e)
- (frexp x)

- **Testing the Sign of Numbers**

- (sign x)

- **Approximate Comparison of Real Numbers**

- (fcmp x y epsilon)

Special Functions

- Error Functions
- Gamma Functions
- Psi (Digamma) Functions
- Zeta Functions

Discussion – Exception Handling

- PLT Scheme provides exception handling, for example
 - Application Type Errors
 - Application Mismatch Errors
- PLT Scheme provides `+inf.0`, `-inf.0`, `+nan.0`
- PLT Scheme provides contracts to define function parameter and result constraints
- Design Decisions
 - Use contracts for type checking, range checking, and parameter consistency checking
 - Check all function provided by modules (using `provide/contract`)
 - Use `+inf.0` for overflow, `-inf.0` for underflow
 - Use `+nan.0` for domain errors (or when the value can't be computed for any reason other than the above)

Demo – Contracts (1)

- Fixed number of arguments and single result

```
(mean (-> (vectorof real?) real?))
```

- Multiple lambda forms [case-lambda]

```
(variance  
  (case-> (-> (vectorof real?) real? (>=/c 0.0))  
          (-> (vectorof real?) (>=/c 0.0))))
```

- Interparameter constraints

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

Demo – Contracts (2)

- **New flat contract [flat-named-contract]**

```
(define nonempty-vector-of-reals?
  (flat-named-contract "nonempty-vector-of-reals?"
    (lambda (x)
      (and (vector? x)
           (> (vector-length x) 0)
           (let ((n (vector-length x)))
             (let/ec exit
               (do ((i 0 (+ i 1)))
                   ((= i n) #t)
                   (if (not (real? (vector-ref x i)))
                       (exit #f))))))))))
```

- **Usage**

```
(maximum (-> nonempty-vector-of-reals? real?))
```

Demo – Contracts (3)

- **Multiple values**

```
(minimum-maximum
```

```
  (-> nonempty-vector-of reals? (values real? real?)))
```

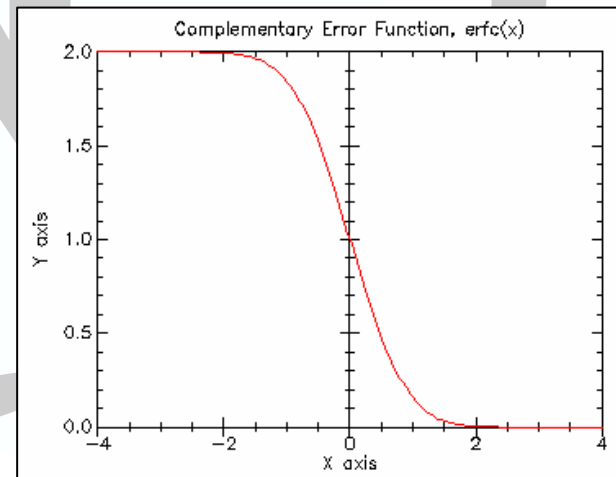
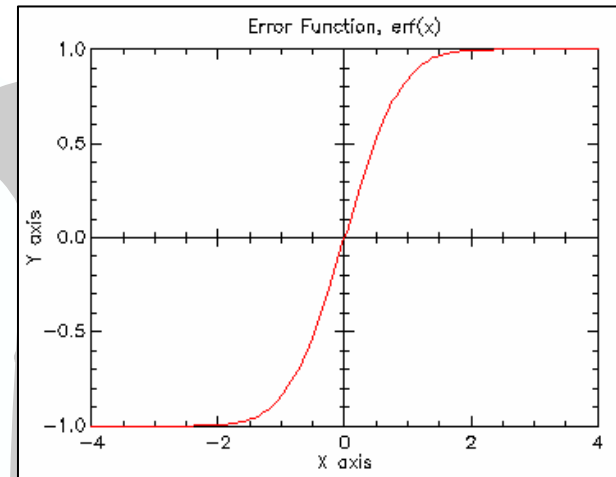
-or-

```
(minimum-maximum
```

```
  (->* (nonempty-vector-of-reals?)  
        (real? real?)))
```

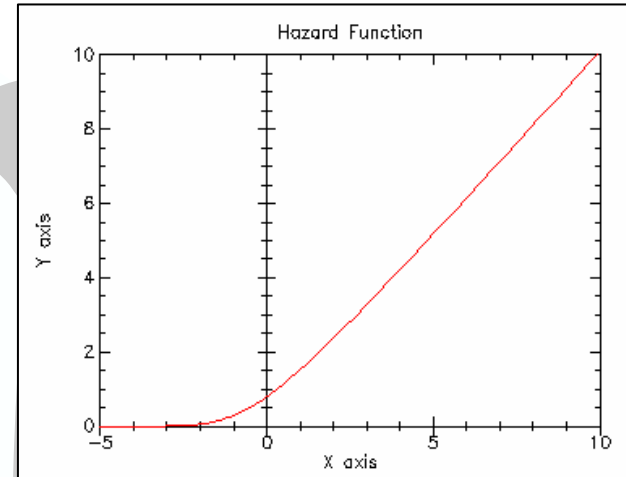
Error Functions (1)

- Error Function
(`erf x`)
- Complementary Error Function
(`erfc x`)
(`log-erfc x`)



Error Functions (2)

- Hazard Function
(hazard x)



Demo – PLoT Scheme

- The plots on the preceding slides were produced with the PLT Scheme plot package – PLoT Scheme

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

- Simple plot with default ranges

```
(require (lib "erfc.ss" "science" "special-functions"))  
(require (lib "plot.ss" "plot"))  
(plot (line erf))
```

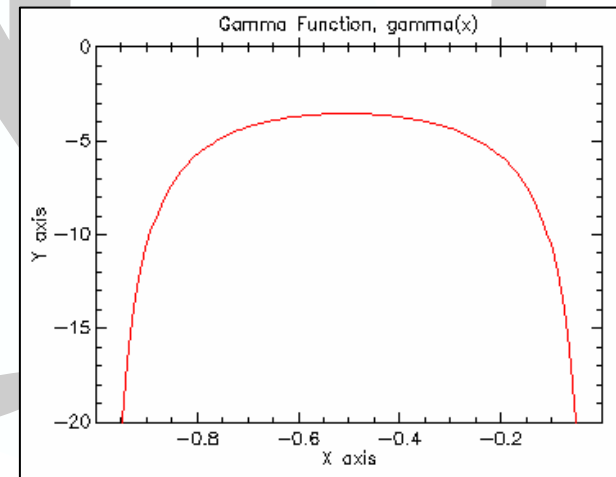
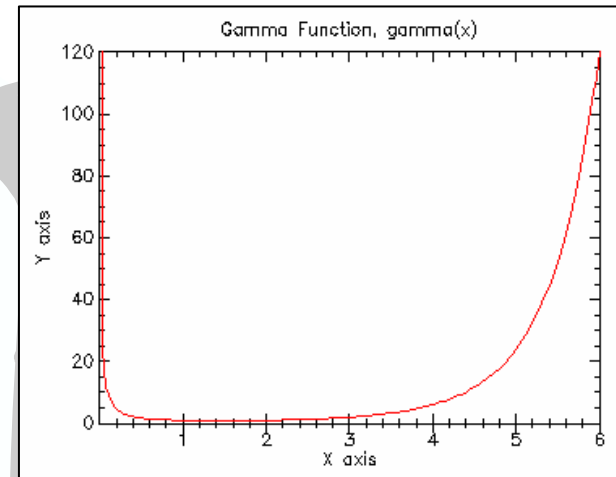
- Erf plot produced using

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

- Alexander Friedman and Jamie Raymond. "PLoT Scheme". *Scheme Workshop 2003*. November 2003
– <http://www.cs.utah.edu/techreports/2003/pdf/UUCS-03-023.pdf>

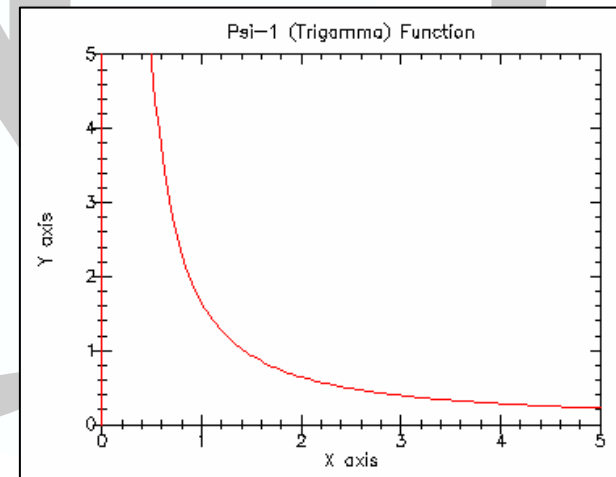
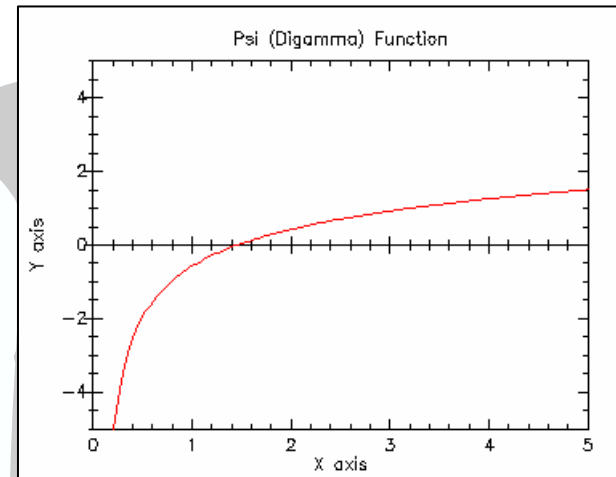
Gamma Functions

- **Gamma**
(gamma x)
- **Natural Logarithm of Gamma**
(lngamma x)
(lngamma-*sgn* x)
- **Regulated Gamma Function**
(gammastar x)
- **Reciprocal of Gamma**
(gammainv x)
- **Factorial, n!**
(fact n)
(lnfact n)
- **Double Factorial, n!!**
(double-fact n)
(lndouble-fact n)
- **Combinatorial Factor**
(choose n m)
(lnchoose n m)



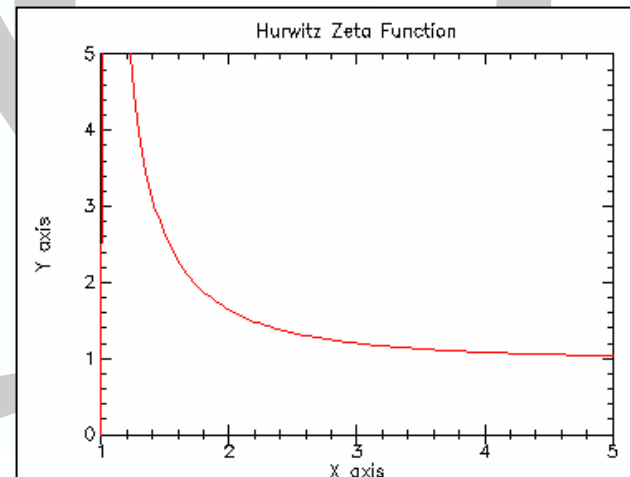
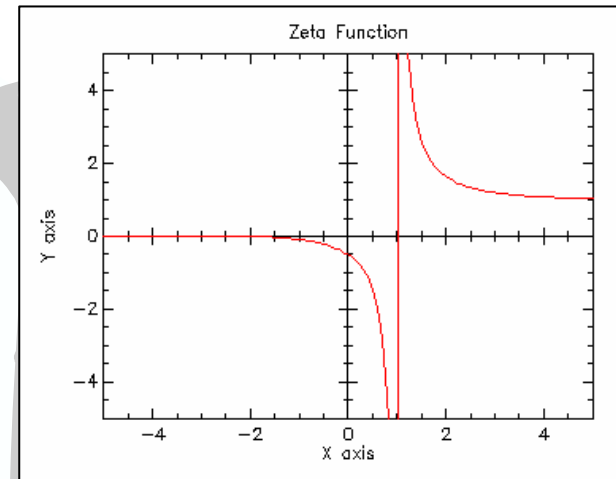
Psi (Digamma) Functions

- Digamma Function
(`psi-int n`)
(`psi x`)
(`psi-1pi y`)
- Trigamma Function
(`psi-1-int n`)
(`psi-1 x`)
- Polygamma Function
(`psi-n n x`)



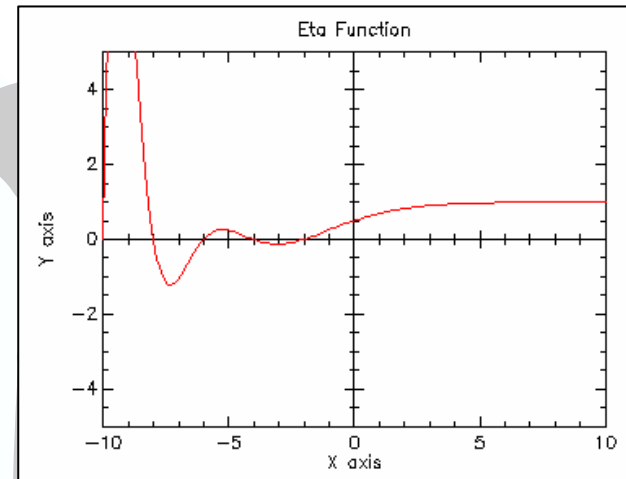
Zeta Functions (1)

- Riemann Zeta Function
(zeta-int n)
(zeta s)
- Reimann Zeta Function Minus One
(zetam1-int n)
(zetam1 n)
- Hurwitz Zeta Function
(hzeta s a)



Zeta Functions (2)

- Eta Function
(eta-int n)
(eta s)



Discussion – Mutually Dependent Modules

- The Gamma, Psi, and Zeta functions are mutually dependent
- Simply putting them into separate modules doesn't work
 - PLT Scheme doesn't allow circular dependencies between modules
 - Isn't a problem in Common LISP (or C for that matter) because the dependencies are limited to run-time function references
- Previous discussions in the PLT mailing list have suggested using units (instead of modules)
 - Module syntax is convenient for the end user
 - Units would require a different syntax for the end user for this (and maybe a few similar situations)
- Design Decision
 - Use separate files for the implementations
 - Create one module that includes the separate implementations
 - Create individual modules that provides specific visibility, if desired
- Any ideas?

Random Number Generation (1)

- Based on Pierre L'Ecuyer's 54-bit implementation of SRFI 27 – Sources of Random Bits
 - `(random-integer n)`
 - `(random-real)`
 - `default-random-source`
 - `(make-random-source)`
 - `(random-source-state-ref s)`
 - `(random-source-state-set! s state)`
 - `(random-source-randomize! s)`
 - `(random-source-pseudo-randomize! s i j)`
 - `(random-source-make-integers s)`
 - `(random-source-make-reals s)`
 - `(random-source-make-reals s unit)`

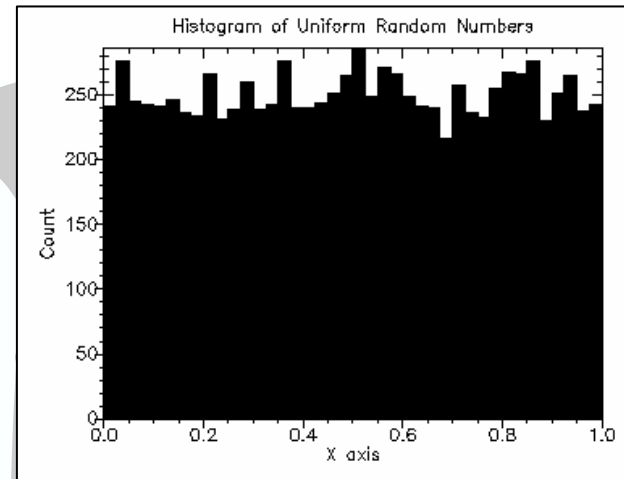
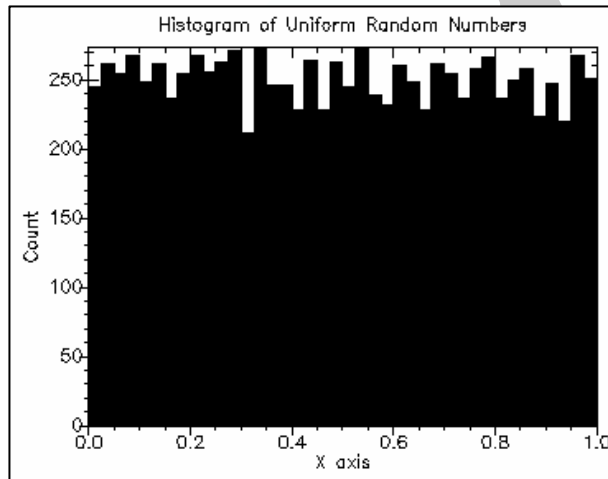
Random Number Generation (2)

- **The `current-random-source` Parameter**
 - Provides a per-thread random source reference
 - `(current-random-source)` returns the current random source
 - `(current-random-source s)` sets the current random source to `s`
 - `parameterize` provides a mechanism for dynamically binding parameters, including `current-random-source`
- **Macros `with-random-source` and `with-new-random-source` provide wrappers around `parameterize`**

Random Number Generation (3)

- Primitive uniform integer and real functions that understand `current-random-source`
 - `(random-uniform-int [s] n)`
 - `(random-uniform [s])`
- Miscellaneous – to make random sources better match PLT Scheme structures
 - `(random-source-state s)`
 - `(set-random-source-state! s state)`

Demo – Random Number Generation



- **Plot uniform random numbers**

```
(require (lib "random-source.ss" "science"))
(require (lib "histogram-with-graphics.ss" "science"))
(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")))
```

Random Distributions

- Continuous Distributions

- Beta
- Bivariate Gaussian
- Chi-squared
- Exponential
- F-distribution
- Flat (Uniform)
- Gamma
- Gaussian
- Unit Gaussian
- Gaussian Tail
- Unit Gaussian Tail
- Lognormal
- Pareto
- t-distribution
- Triangular

- Discrete Distributions

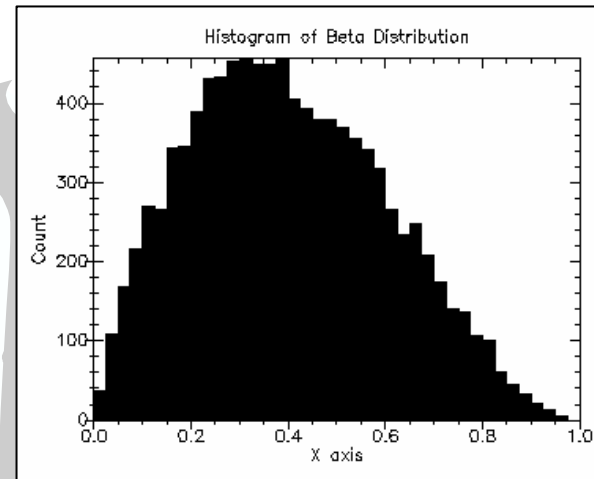
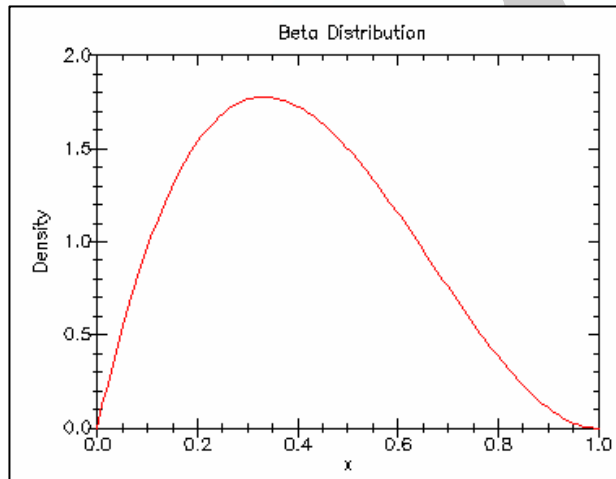
- Binomial
- Bernoulli
- Geometric
- Logarithmic
- Poisson

- General Discrete Distributions

Continuous Distributions

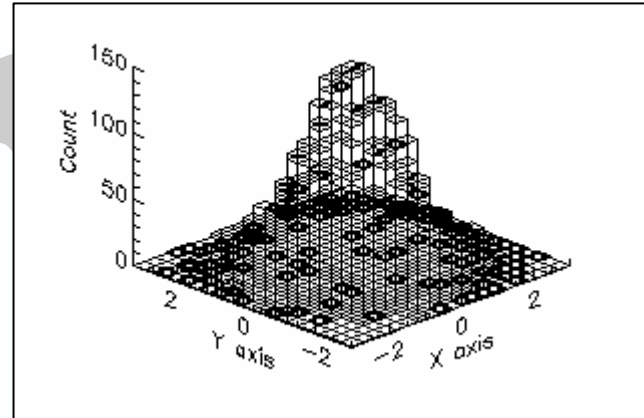
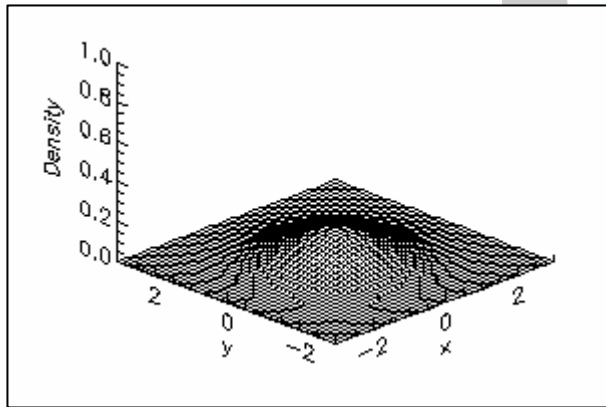
- Continuous Distributions
 - Beta
 - Bivariate Gaussian
 - Chi-squared
 - Exponential
 - F-distribution
 - Flat (Uniform)
 - Gamma
 - Gaussian
 - Unit Gaussian
 - Gaussian Tail
 - Unit Gaussian Tail
 - Lognormal
 - Pareto
 - t-distribution
 - Triangular

Beta



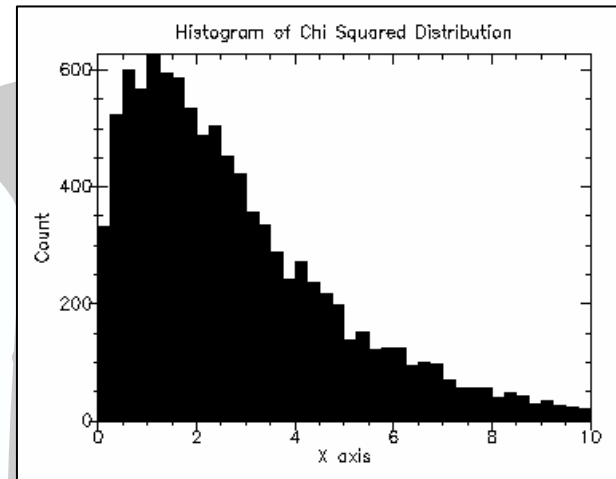
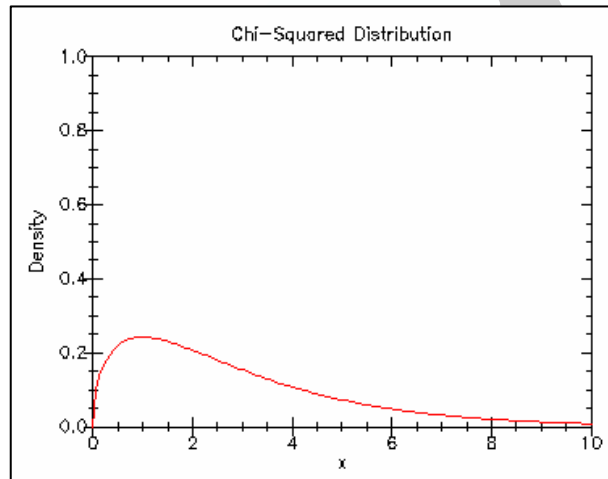
- **Beta Distribution**
(random-beta [s] nu)
(beta-pdf x nu)
- **Beta Graphics**
(beta-plot nu)

Bivariate Gaussian



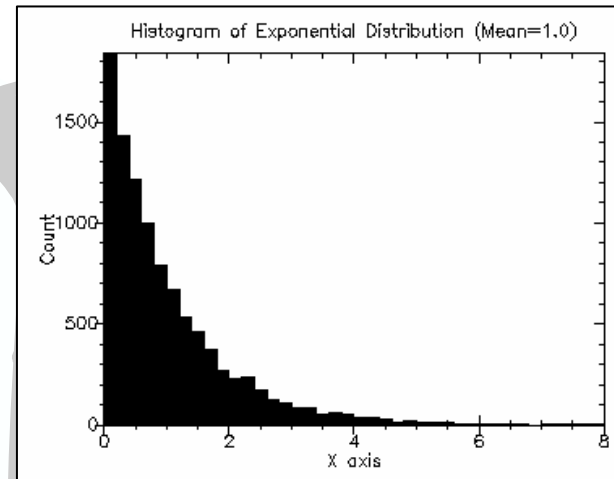
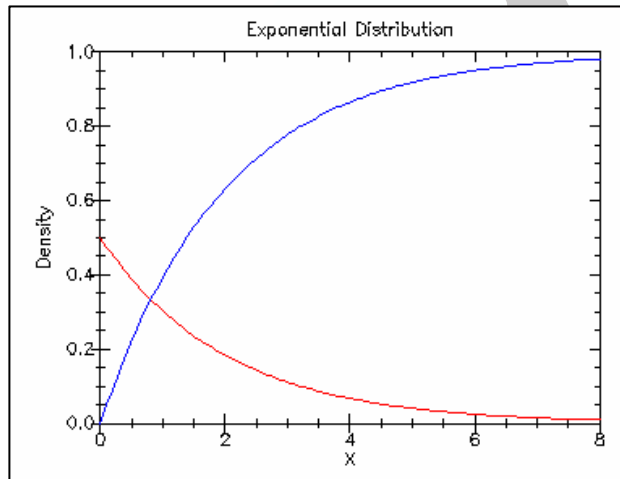
- **Bivariate Gaussian Distribution**
(random-bivariate-gaussian
[s] sigma-x sigma-y rho)
(bivariate-gaussian-pdf x y sigma-x sigma-y rho)
- **Bivariate-Gaussian Graphics**
(bivariate-gaussian-plot sigma-x sigma-y rho)

Chi-Squared



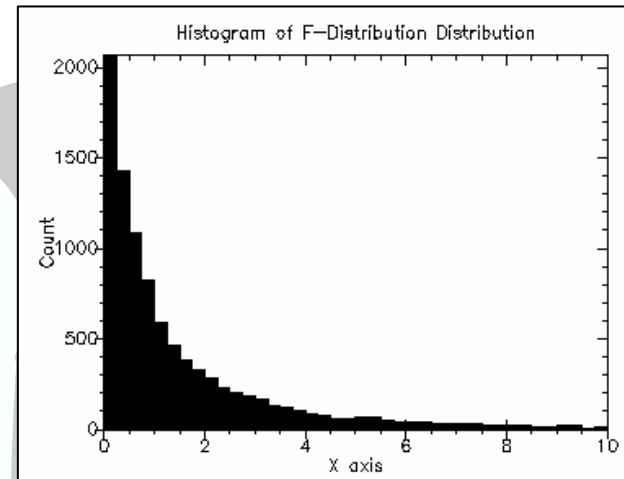
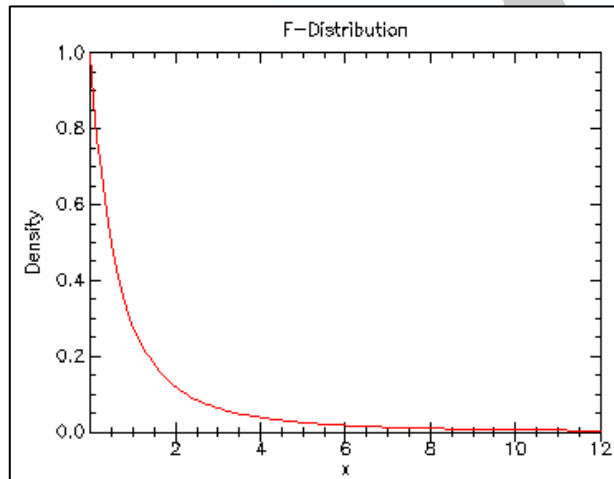
- **Chi-Squared Distribution**
(random-chi-squared [s] nu)
(chi-squared-pdf nu)
- **Chi-Squared Graphics**
(chi-squared-plot nu)

Exponential



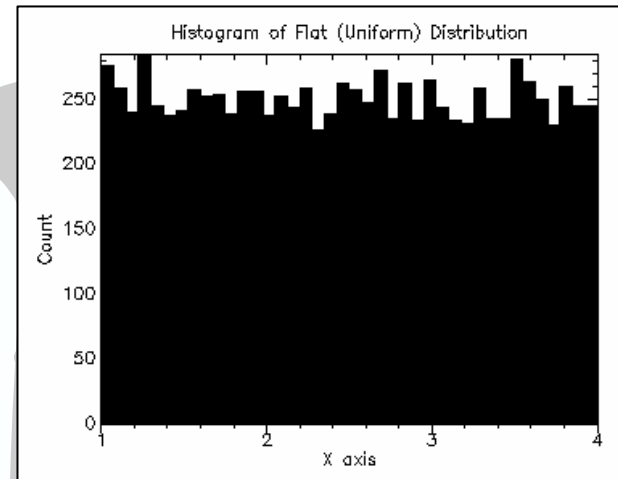
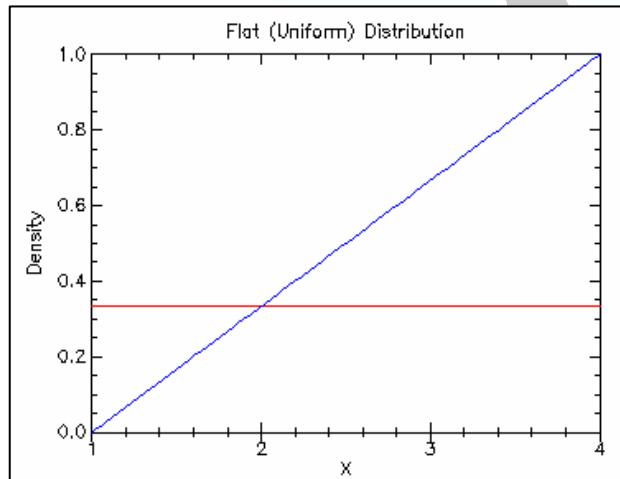
- **Exponential Distribution**
(random-exponential [s] mu)
(exponential-pdf x mu)
(exponential-cdf x mu)
- **Exponential Graphics**
(exponential-plot mu)

F-Distribution



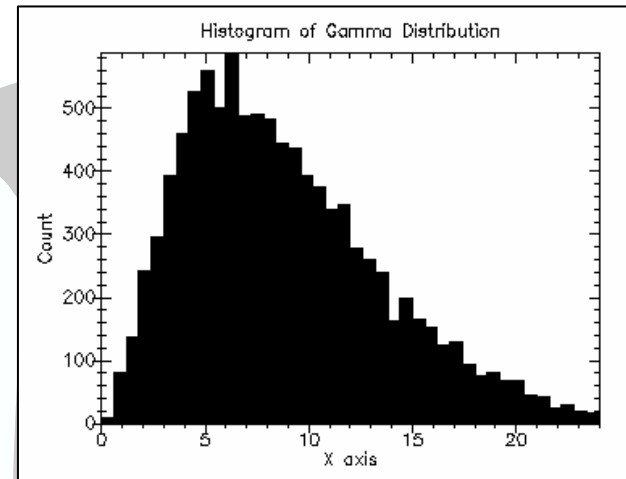
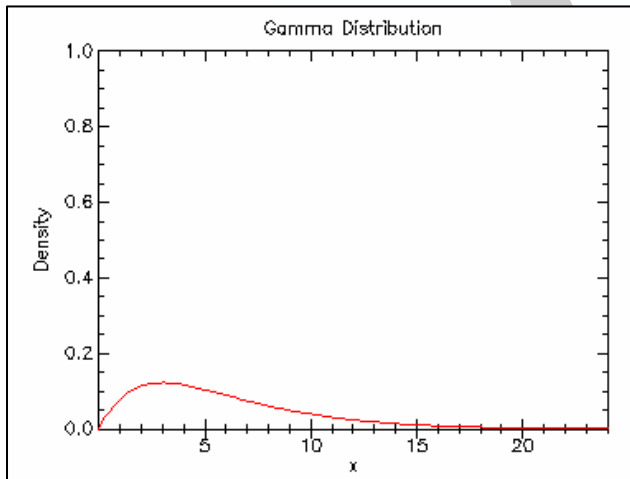
- **F-Distribution**
(random-f-distribution [s] nu1 nu2)
(f-distribution-pdf nu1 nu2)
- **F-Distribution Graphics**
(f-distribution-pdf nu1 nu2)

Flat (Uniform)



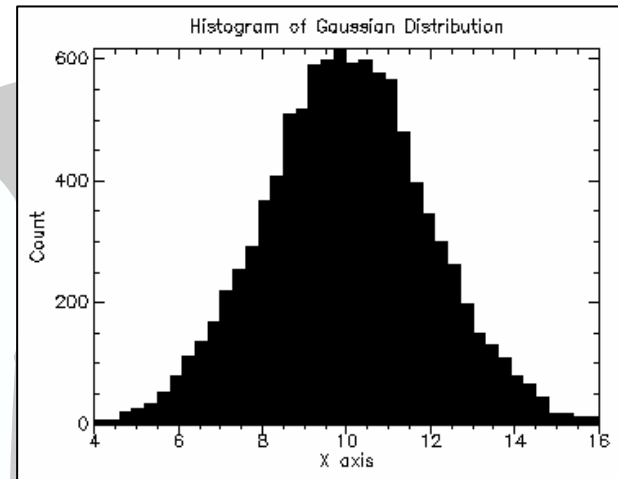
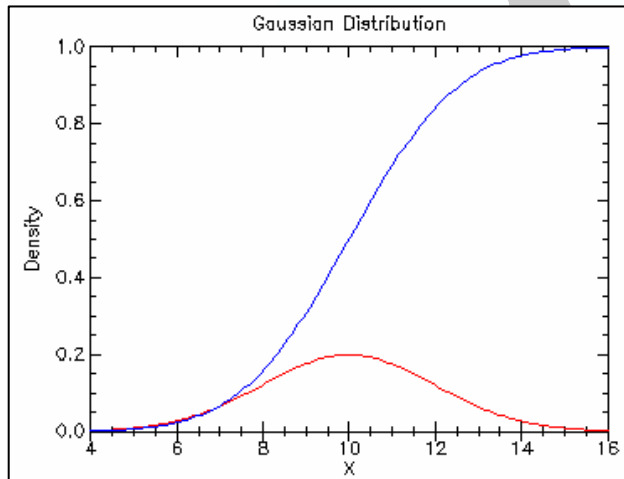
- Flat (Uniform) Distribution
(random-flat [s] a b)
(flat-pdf x a b)
(flat-cdf x a b)
- Flat (Uniform) Graphics
(flat-plot a b)

Gamma



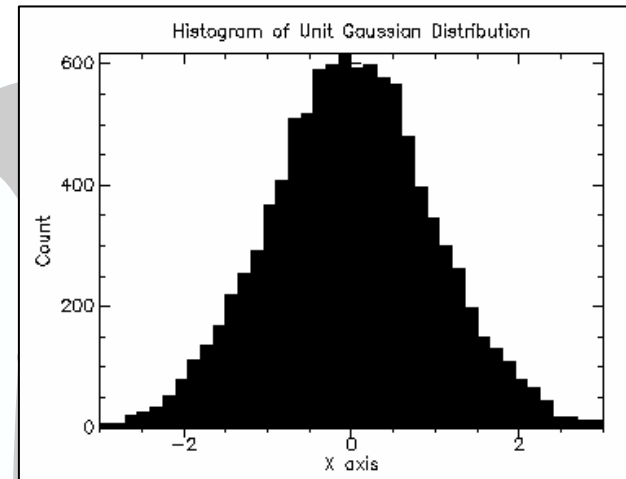
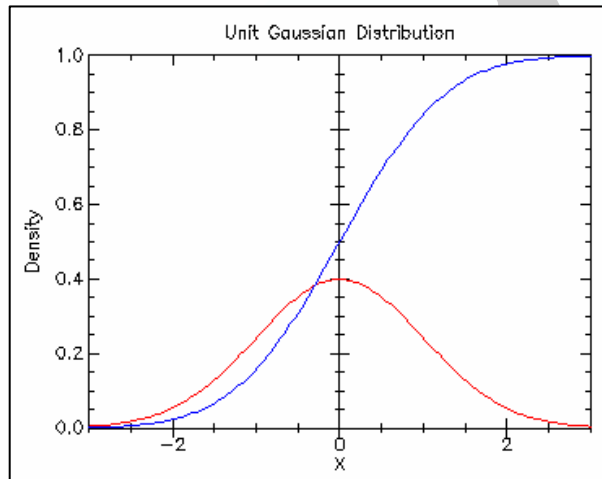
- **Gamma Distribution**
(`random-gamma [s] a b`)
(`gamma-pdf x a b`)
- **Gamma Graphics**
(`gamma-plot a b`)

Gaussian (Normal)



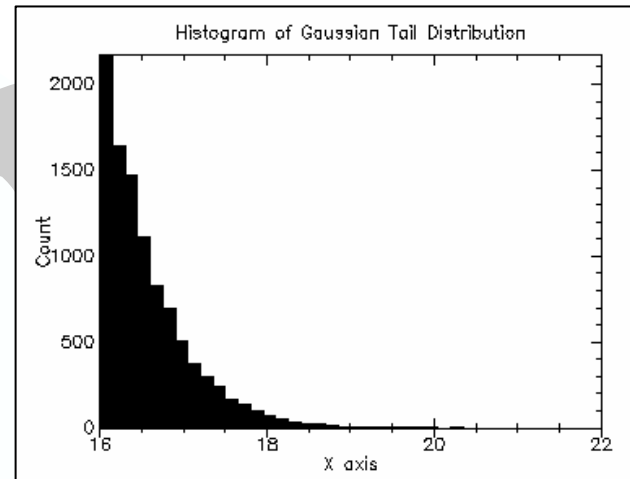
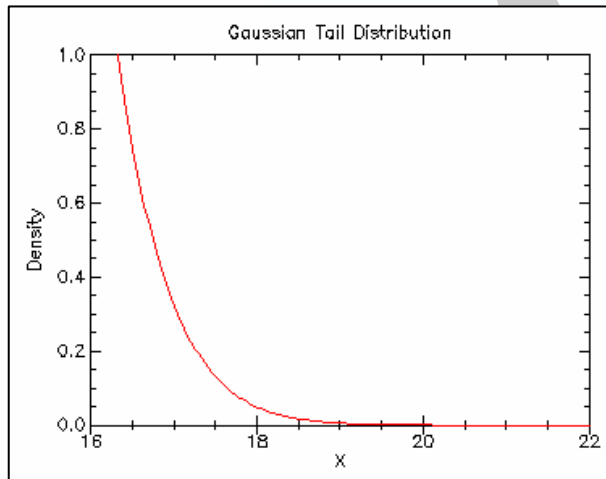
- **Gaussian (Normal) Distribution**
(random-gaussian [s] mu sigma)
(random-gaussian-ratio-method [s] mu sigma)
(gaussian-pdf x mu sigma)
(gaussian-cdf x mu sigma)
- **Gaussian Graphics**
(gaussian-plot mu sigma)

Unit Gaussian (Normal)



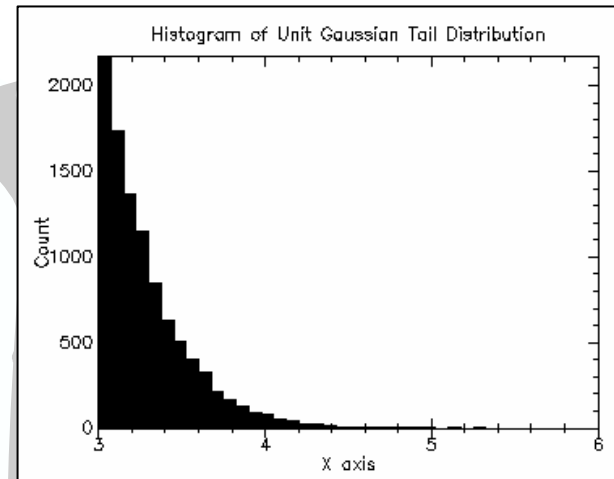
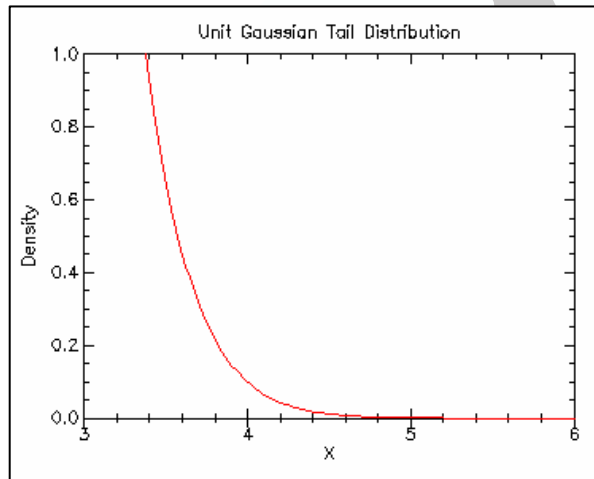
- **Unit Gaussian (Normal) Distribution**
(`random-unit-gaussian [s]`)
(`random-unit-gaussian-ratio-method [s]`)
(`unit-gaussian-pdf x`)
(`unit-gaussian-cdf x`)
- **Unit Gaussian (Normal) Graphics**
(`unit-gaussian-plot`)

Gaussian Tail



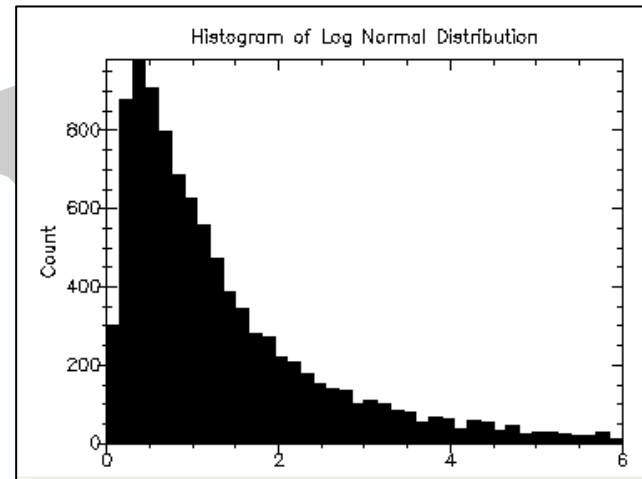
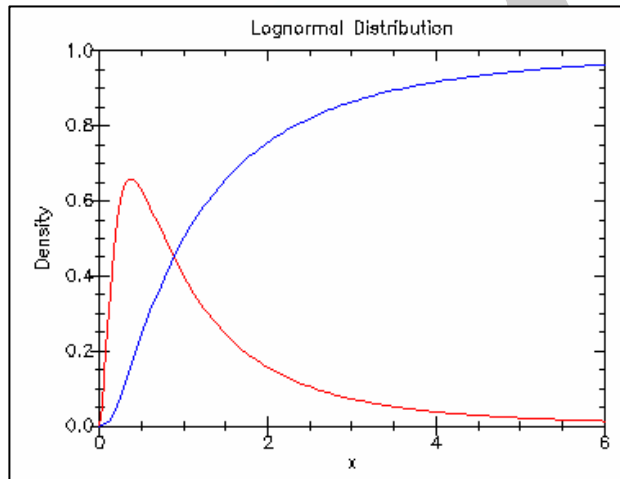
- **Gaussian Tail Distribution**
(random-gaussian-tail [s] a mu sigma)
(gaussian-tail-pdf x a mu sigma)
- Gaussian Tail Graphics**
(gaussian-tail-plot a mu sigma)

Unit Gaussian Tail



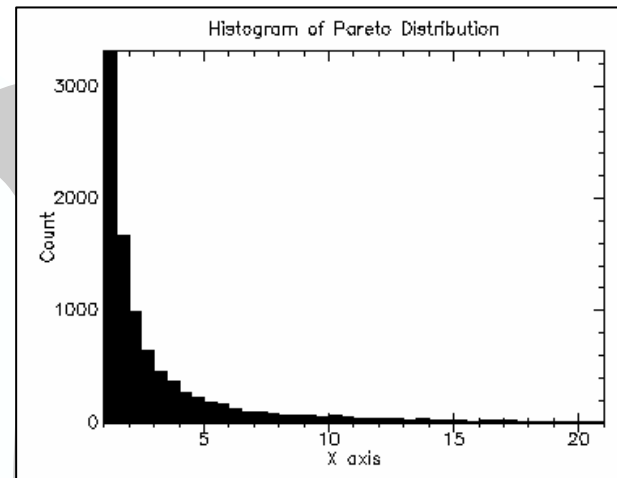
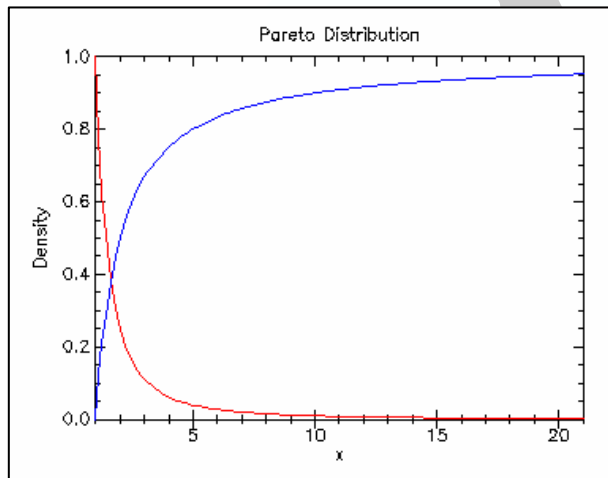
- Unit Gaussian Tail Distribution
(random-unit-gaussian-tail [s] a)
(unit-gaussian-pdf x a)
Unit Gaussian Tail Graphics
(unit-gaussian-tail-plot a)

Log Normal



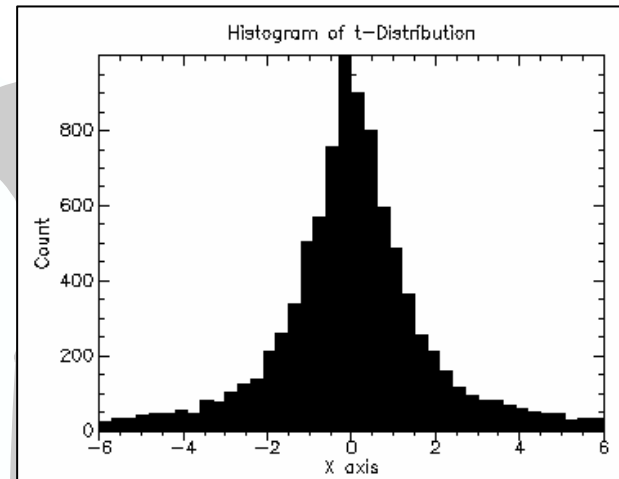
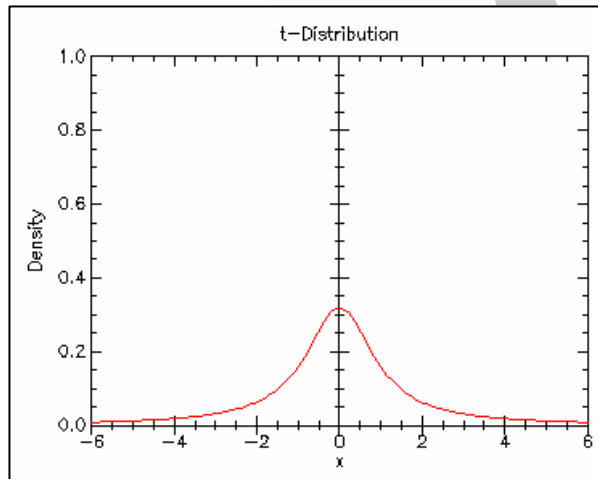
- **Log Normal Distribution**
(random-lognormal [s] mu sigma)
(lognormal-pdf x mu sigma)
(lognormal-cdf x mu sigma)
- **Log Normal Graphics**
(lognormal-plot mu sigma)

Pareto



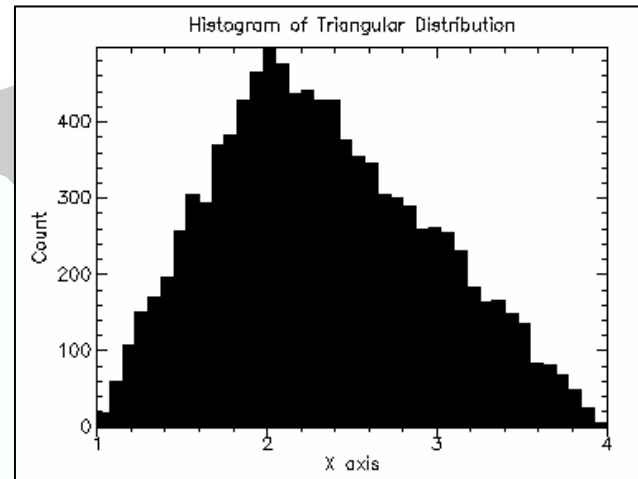
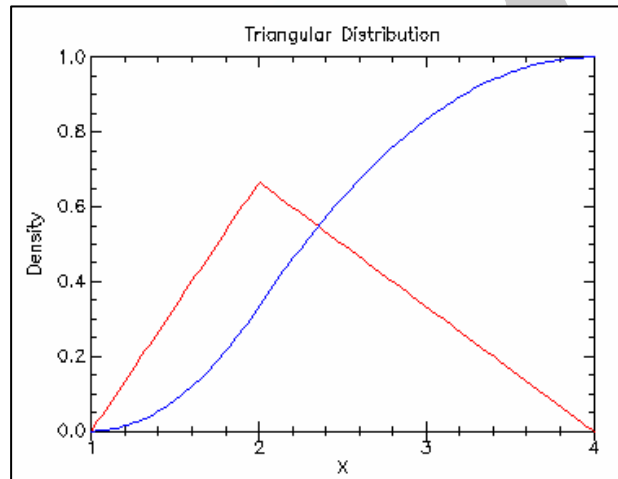
- Pareto Distribution
(random-pareto [s] a b)
(pareto-pdf x a b)
(pareto-cdf x a b)
- Pareto Graphics
(pareto-plot a b)

t-Distribution



- **t-Distribution**
(random-t-distribution [s] nu)
(t-distribution-pdf x nu)
- **t-Distribution Graphics**
(t-distribution-plot nu)

Triangular

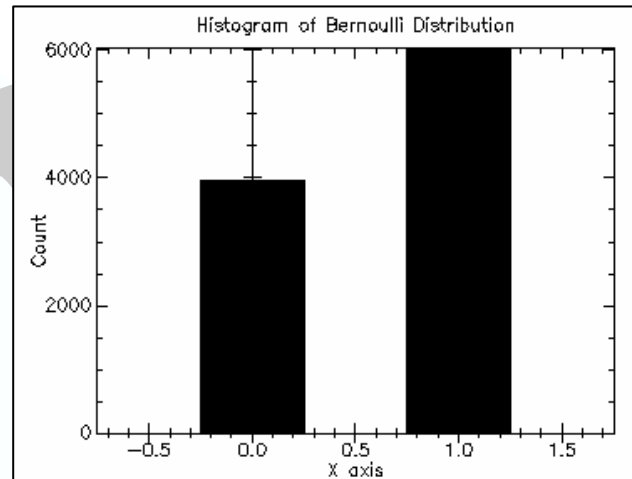
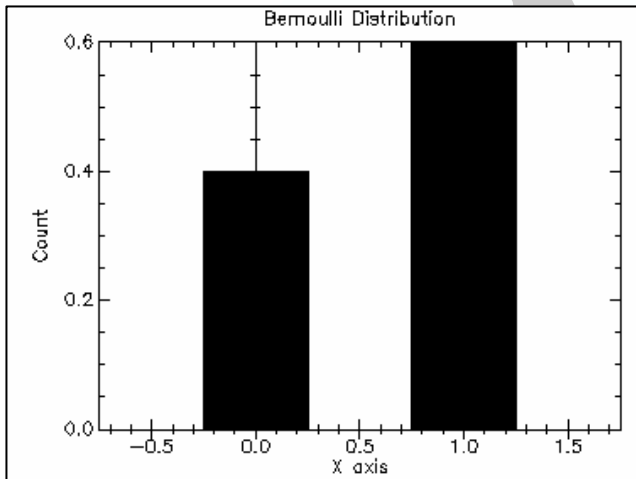


- **Triangular Distribution**
(random-triangular [s] a b c)
(triangular-pdf x a b c)
(triangular-cdf x a b c)
- **Triangular Graphics**
(triangular-plot a b c)

Discrete Distributions

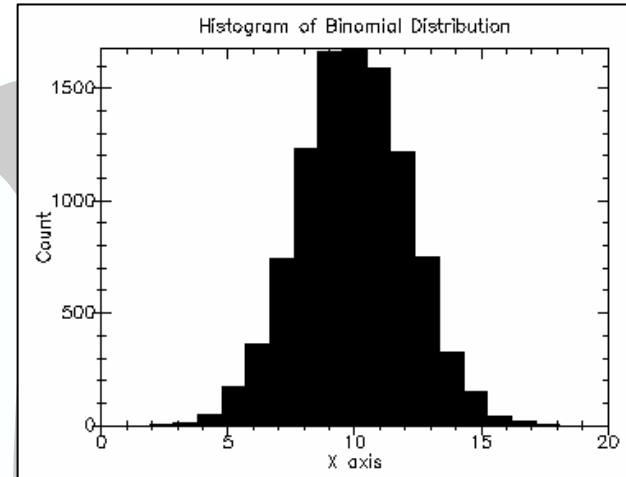
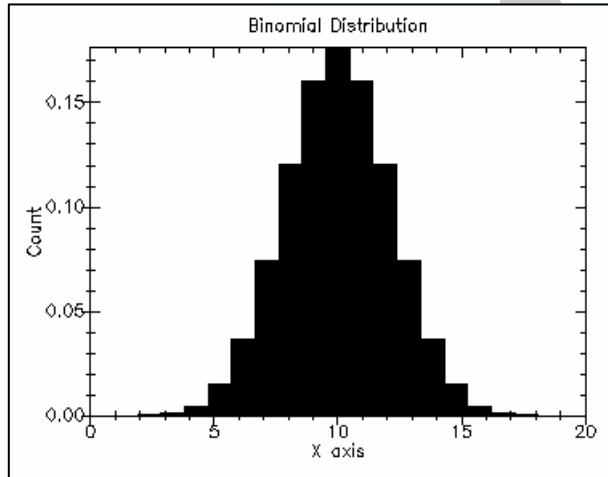
- Discrete Distributions
 - Binomial
 - Bernoulli
 - Geometric
 - Logarithmic
 - Poisson

Bernoulli



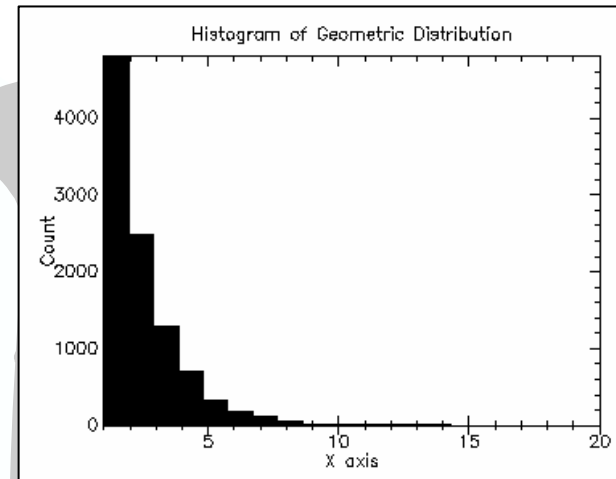
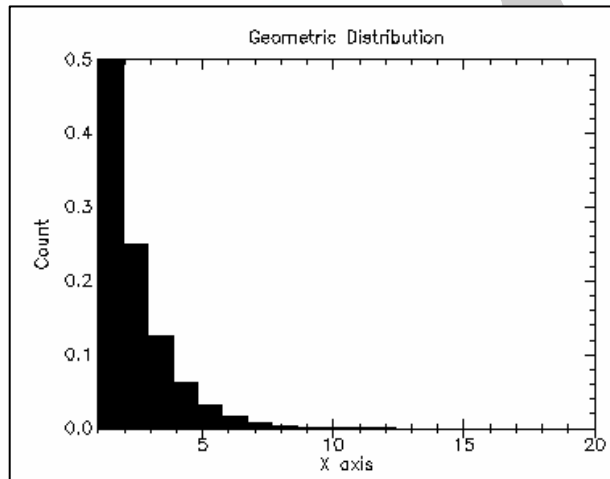
- **Bernoulli Distribution**
(random-bernoulli [s] p)
(bernoulli-pdf n p)
(bernoulli-cdf n p)
- **Bernoulli Graphics**
(bernoulli-plot p)

Binomial



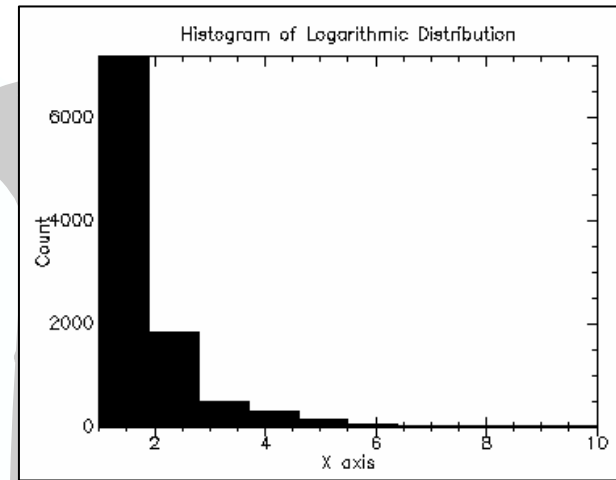
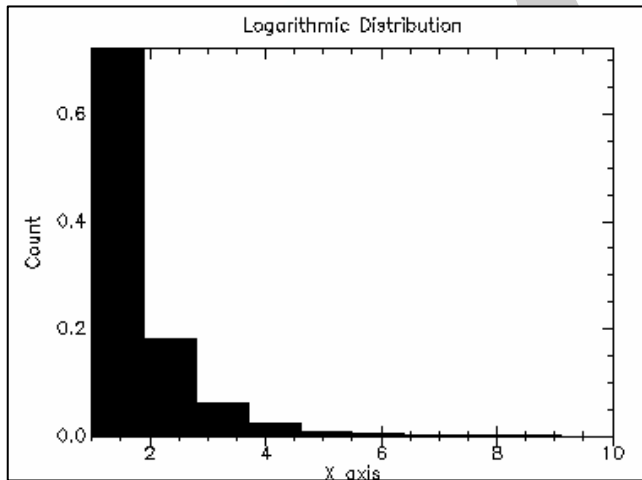
- **Binomial Distribution**
(random-binomial [s] p n)
(binomial-pdf k p n)
- **Binomial-Graphics**
(binomial-plot p n)

Geometric



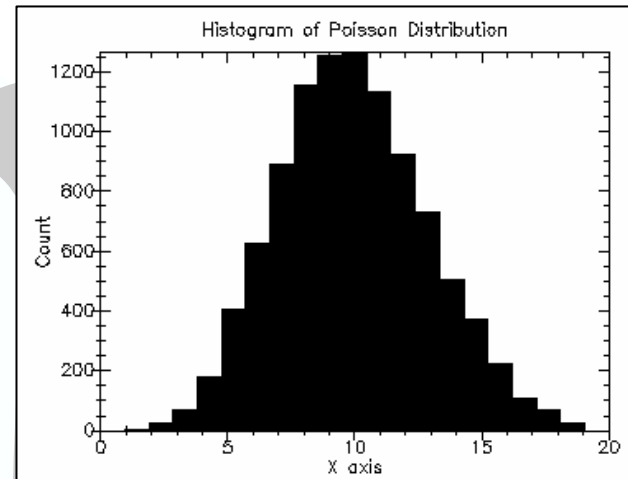
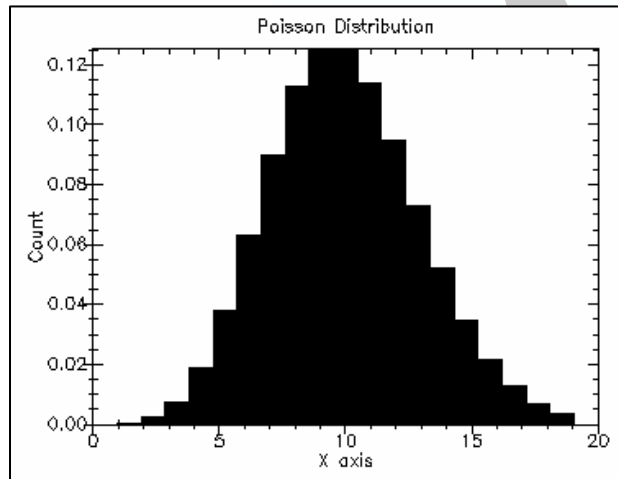
- **Geometric Distribution**
(`random-geometric [s] p`)
(`geometric-pdf k p`)
- **Geometric Graphics**
(`geometric-plot p`)

Logarithmic



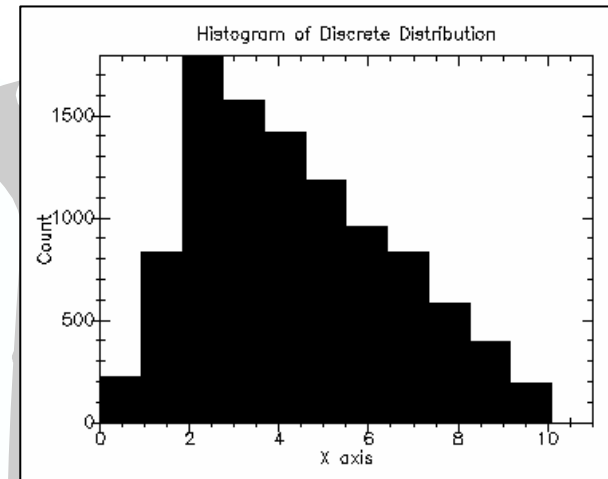
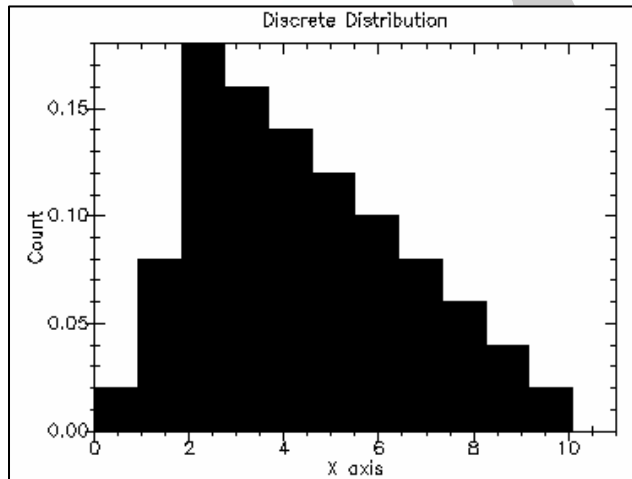
- **Logarithmic Distribution**
(`random-logarithmic [s] p`)
(`logarithmic-pdf k p`)
- **Logarithmic Graphics**
(`logarithmic-plot p`)

Poisson



- **Poisson Distribution**
(random-poisson [s] mu)
(poisson-pdf k mu)
- **Poisson Graphics**
(poisson-plot mu)

General Discrete Distributions



- **General Discrete Distribution**
(discrete? x)
(make-discrete w)
(random-discrete [s] d)
(discrete-pdf d k)
(discrete-cdf d k)
- **General Discrete Graphics**
(discrete-plot d)

Statistics (1)

- **Mean, Standard Deviation, and Variance**
 - (mean data)
 - (variance data [mean])
 - (standard-deviation data [mean])
 - (variance-with-fixed-mean data mean)
 - (standard-deviation-with-fixed-mean data mean)
- **Absolute Deviation**
 - (absolute-deviation data [mean])
- **Higher Moments (Skewness and Kurtosis)**
 - (skew data [mean sd])
 - (kurtosis data [mean sd])
- **Autocorrelation**
 - (lag-1-autocorrelation data [mean])
- **Covariance**
 - (covariance data1 data2 [mean1 mean2])
 - (covariance-with-fixed-means data1 data2 mean1 mean2)

Statistics (2)

- **Weighted Samples**

- (weighted-mean w data)
- (weighted-variance w data [mean])
- (weighted-standard-deviation w data [mean])
- (weighted-variance-with-fixed-mean
w data mean)
- (weighted-standard-deviation-with-fixed-mean
w data mean)
- (weighted-absolute-deviation w data [mean])
- (weighted-skew w data [mean sd])
- (weighted-kurtosis w data [mean sd])

Statistics (3)

- **Maximum and Minimum Values**
 - `(maximum data)`
 - `(minimum data)`
 - `(minimum-maximum data)`
 - `(maximum-index data)`
 - `(minimum-index data)`
 - `(minimum-maximum-index data)`
- **Median and Percentiles**
 - `(median-from-sorted-data sorted-data)`
 - `(quantile-from-sorted-data sorted-data f)`

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
```

Histograms

- Histograms provide a convenient way of summarizing the distribution of a set of data
- A histogram consists of a set of bins that count the number of events falling into a given range of a continuous variable, x
- The bins of a histogram contain real numbers so they can record both integer and non-integer distributions
- The PLoT Scheme plot collection has been extended to plot histograms
- Both 1D and 2D histograms are provided

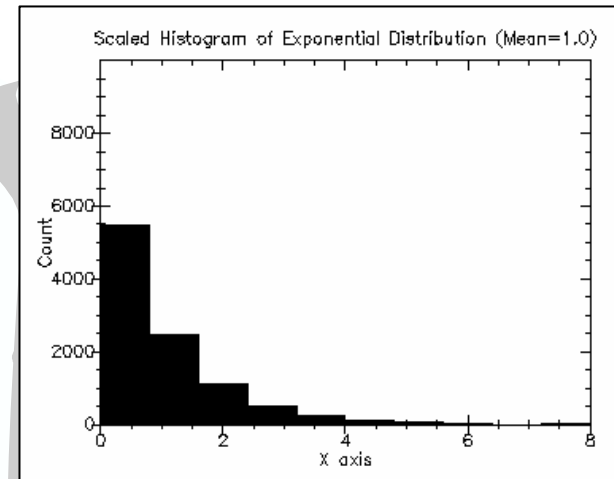
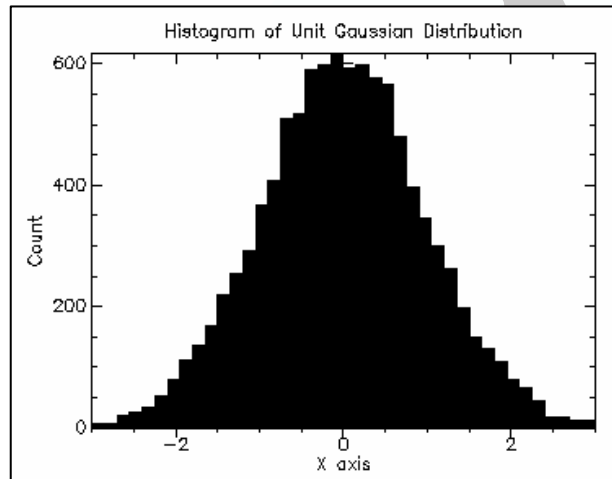
Basic Histogram Functions

- `(histogram? X)`
- `(make-histogram n)`
- `(make-histogram-with-ranges-uniform n min max)`
- `(histogram-n h)`
- `(histogram-ranges h)`
- `(set-histogram-ranges! h ranges)`
- `(set-histogram-ranges-uniform! h min mix)`
- `(histogram-bins h)`
- `(histogram-increment! h x)`
- `(histogram-accumulate! h x weight)`
- `(histogram-get h i)`
- `(histogram-get-ranges h i)`

Histogram Statistics

- `(histogram-max h)`
- `(histogram-min h)`
- `(histogram-mean h)`
- `(histogram-sigma h)`
- `(histogram-sum h)`

Histogram Graphics



- Histogram Plot
(`histogram-plot h [title]`)
 - Scaled to max bin value
- Scaled Histogram Plot
(`histogram-plot-scaled h [title]`)
 - Scaled to sum of bin values
 - Most useful for small number of bins (say 10 or less)

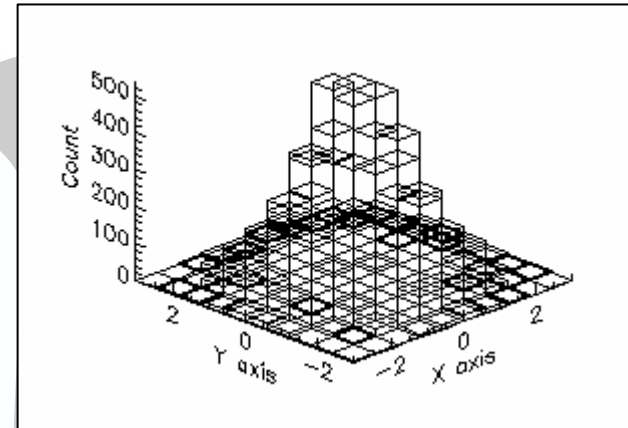
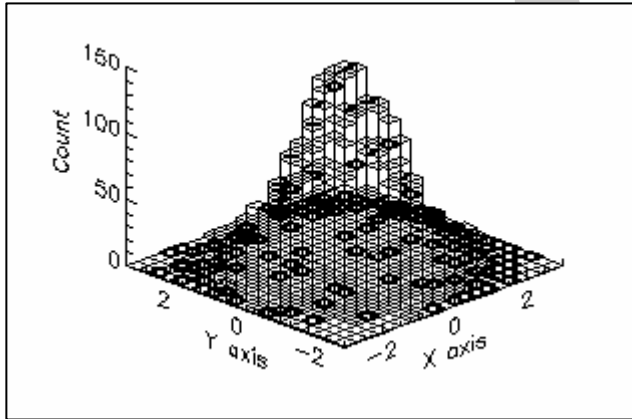
Basic 2D Histogram Functions

- `(histogram-2d? X)`
- `(make-histogram-2d nx ny)`
- `(make-histogram-2d-with-ranges-uniform
 nx ny x-min x-max y-min y-max)`
- `(histogram-2d-nx h)`
- `(histogram-2d-ny h)`
- `(histogram-2d-x-ranges h)`
- `(histogram-2d-y-ranges h)`
- `(set-histogram-2d-ranges! h x-ranges y-ranges)`
- `(set-histogram-2d-ranges-uniform!
 h x-min x-max y-min y-max)`
- `(histogram-2d-bins h)`
- `(histogram-2d-increment! h x y)`
- `(histogram-2d-accumulate! h x y weight)`
- `(histogram-2d-get h i j)`
- `(histogram-get-x-range h i j)`
- `(histogram-get-y-range h i j)`

2D Histogram Statistics

- `(histogram-2d-max h)`
- `(histogram-2d-min h)`

2D Histogram Graphics



- 2D Histogram Plot
(`histogram-2d-plot h [title]`)
 - Scaled to the max bin value

Demo – Extending PLoT Scheme

- The histogram graphics are implemented as extensions to the PLoT Scheme plot collection
- Alexander Friedman and Jamie Raymond. "PLoT Scheme". *Scheme Workshop 2003*. November 2003
- `plot-histogram.ss`
- `plot-histogram-2d.ss`

Chebyshev Approximations

- Compute Chebyshev approximations to univariate functions
- `chebyshev-series` structure
(`define-struct chebyshev-series`
 (`coefficients`
 `order`
 `lower`
 `upper`))
- **Functions**
(`make-chebyshev-series-order order`)
(`chebyshev-series-init cs func a b`)
(`chebyshev-eval cs x`)
(`chebyshev-eval-n cs n x`)

Chebyshev Example

```
(require (lib "chebyshev.ss" "science"))

(define (f x)
  (if (< x 0.5) .25 .75))

(define (chebyshev-example n)
  (let ((cs (make-chebyshev-series-order 40)))
    (chebyshev-series-init cs f 0.0 1.0)
    (do ((i 0 (+ i 1)))
        ((= i n) (void))
      (let ((x (exact->inexact (/ i n))))
        (printf "~a ~a ~a ~a\n"
                 x
                 (f x)
                 (chebyshev-eval-n cs 10 x)
                 (chebyshev-eval cs x)))))))
```

Chebyshev Series Usage

```
;; Chebyshev expansion for log (gamma(x) / gamma (8))
;; 5 < x < 10, -1 < t < 1
(define gamma-5-10-data
  #(-1.5285594096661578881275075214
    4.8259152300595906319768555035
    0.2277712320977614992970601978
    -0.0138867665685617873604917300
    0.0012704876495201082588139723
    -0.0001393841240254993658962470
    0.0000169709242992322702260663
    -2.2108528820210580075775889168e-06
    3.0196602854202309805163918716e-07
    -4.2705675000079118380587357358e-08
    6.2026423818051402794663551945e-09
    -9.1993973208880910416311405656e-10
    1.3875551258028145778301211638e-10
    -2.1218861491906788718519532978e-11
    3.2821736040381439555133562600e-12
    -5.1260001009953791220611135264e-13
    8.0713532554874636696982146610e-14
    -1.2798522376569209083811628061e-14
    2.0417711600852502310258808643e-15
    -3.2745239502992355776882614137e-16
    5.2759418422036579482120897453e-17
    -8.5354147151695233960425725513e-18
    1.3858639703888078291599886143e-18
    -2.2574398807738626571560124396e-19))

(define gamma-5-10-cs
  (make-chebyshev-series
   gamma-5-10-data
   23 -1.0 1.0))

;; gamma-xgthalf: real -> real
;; Gamma(x) for x >= 1/2
(define (gamma-xgthalf x)
  (cond ((= x 0.5)
         1.772453850905516002729817)
        ...
        ((< x 10.0)
         ;; This is a sticky area. The logarithm is
         ;; too large and the gammastar series is
         ;; not good.
         (let ((gamma-8 5040.0)
               (t (/ (- (* 2.0 x) 15.0) 5.0)))
           (* gamma-8
              (exp (chebyshev-eval gamma-5-10-cs t)))))
        ...
        (else
         +inf.0)))
```

PLT Scheme Simulation Collection

- Simulation Environment
- Simulation Control
- Queues
- Events
- Processes
- Resources
- Variables
- Histories
- Statistics

PLT Scheme Inference Collection

- Inference Environment
- Inference Control
- Rule Sets
- Rules
- Assertions

Status

- **PLT Scheme Science Collection**
 - Largely complete
 - Pre-release code cleanup and documentation underway
 - Adding contracts to all module provided functions
- **PLT Scheme Simulation Collection**
 - All code prototyped in PLT Scheme
 - Design for process interaction model being considered
 - Based on Ada rendezvous scheme
 - May be postponed until a later release
- **PLT Scheme Inference Collection**
 - Code for naïve forward-chaining inference engine previously ported to PC Scheme

Schedule

- **PLT Scheme Science Collection**
 - Release 1.0 October 2004
 - Release 2.0 July 2005 (tentative)
 - Remaining cdf's
 - Additional random distributions
 - Ordinary Differential Equations (ODEs)
- **PLT Scheme Simulation Collection**
 - Release 1.0 January 2005
 - Release 2.0 July 2005 (tentative)
 - Add continuous simulation (using ODEs)
- **PLT Scheme Inference Collection**
 - Release 1.0 April 2005