Probability and Stochastic Processes

A Friendly Introduction for Electrical and Computer Engineers

SECOND EDITION

MATLAB Function Reference

Roy D. Yates and David J. Goodman

May 22, 2004

This document is a supplemental reference for MATLAB functions described in the text *Probability and Stochastic Processes: A Friendly Introduction for Electrical and Computer Engineers*. This document should be accompanied by `matcode.zip`, an archive of the corresponding MATLAB `.m` files. Here are some points to keep in mind in using these functions.

- The actual programs can be found in the archive `matcode.zip` or in a directory `matcode`. To use the functions, you will need to use the MATLAB command `addpath` to add this directory to the path that MATLAB searches for executable `.m` files.

- The `matcode` archive has both general purpose programs for solving probability problems as well as specific `.m` files associated with examples or quizzes in the text. This manual describes only the general purpose `.m` files in `matcode.zip`. Other programs in the archive are described in main text or in the *Quiz Solution Manual*.

- The MATLAB functions described here are intended as a supplement the text. The code is not fully commented. Many comments and explanations relating to the code appear in the text, the *Quiz Solution Manual* (available on the web) or in the *Problem Solution Manual* (available on the web for instructors).

- The code is instructional. The focus is on MATLAB programming techniques to solve probability problems and to simulate experiments. The code is definitely not bulletproof; for example, input range checking is generally neglected.

- *This is a work in progress.* At the moment (May, 2004), the homework solution manual has a number of unsolved homework problems. As these solutions require the development of additional MATLAB functions, these functions will be added to this reference manual.

- There is a nonzero probability (in fact, a probability close to unity) that errors will be found. If you find errors or have suggestions or comments, please send email to ***ryates@winlab.rutgers.edu***. When errors are found, revisions both to this document and the collection of MATLAB functions will be posted.

Functions for Random Variables

`bernoullipmf` `y=bernoullipmf(p,x)`

```
function pv=bernoullipmf(p,x)
%For Bernoulli (p) rv X
%input = vector x
%output = vector pv
%such that pv(i)=Prob(X=x(i))
pv=(1-p)*(x==0) + p*(x==1);
pv=pv(:);
```

Input: `p` is the success probability of a Bernoulli random variable X, `x` is a vector of possible sample values

Output: `y` is a vector with `y(i)` $= P_X($`x(i)`$)$.

`bernoullicdf` `y=bernoullicdf(p,x)`

```
function cdf=bernoullicdf(p,x)
%Usage: cdf=bernoullicdf(p,x)
% For Bernoulli (p) rv X,
%given input vector x, output is
%vector pv such that pv(i)=Prob[X<=x(i)]
x=floor(x(:));
allx=0:1;
allcdf=cumsum(bernoullipmf(p,allx));
okx=(x>=0); %x_i < 1 are bad values
x=(okx.*x); %set bad x_i=0
cdf= okx.*allcdf(x); %zeroes out bad x_i
```

Input: `p` is the success probability of a Bernoulli random variable X, `x` is a vector of possible sample values

Output: `y` is a vector with `y(i)` $= F_X($`x(i)`$)$.

`bernoullirv` `x=bernoullirv(p,m)`

```
function x=bernoullirv(p,m)
%return m samples of bernoulli (p) rv
r=rand(m,1);
x=(r>=(1-p));
```

Input: `p` is the success probability of a Bernoulli random variable X, `m` is a positive integer vector of possible sample values

Output: `x` is a vector of `m` independent sample values of X

`bignomialpmf` `y=bignomialpmf(n,p,x)`

```
function pmf=bignomialpmf(n,p,x)
%binomial(n,p) rv X,
%input = vector x
%output= vector pmf: pmf(i)=Prob[X=x(i)]
k=(0:n-1)';
a=log((p/(1-p))*((n-k)./(k+1)));
L0=n*log(1-p);
L=[L0; L0+cumsum(a)];
pb=exp(L);
% pb=[P[X=0] ... P[X=n]]^t
x=x(:);
okx =(x>=0).*(x<=n).*(x==floor(x));
x=okx.*x;
pmf=okx.*pb(x+1);
```

Input: `n` and `p` are the parameters of a binomial (n, p) random variable X, `x` is a vector of possible sample values

Output: `y` is a vector with `y(i)` $=$ P_X(`x(i)`).

Comment: This function should always produce the same output as `binomialpmf(n,p,x)`; however, the function calculates the logarithm of the probability and thismay lead to small numerical innaccuracy.

`binomialcdf` `y=binomialcdf(n,p,x)`

```
function cdf=binomialcdf(n,p,x)
%Usage: cdf=binomialcdf(n,p,x)
%For binomial(n,p) rv X,
%and input vector x, output is
%vector cdf: cdf(i)=P[X<=x(i)]
x=floor(x(:)); %for noninteger x(i)
allx=0:max(x);
%calculate cdf from 0 to max(x)
allcdf=cumsum(binomialpmf(n,p,allx));
okx=(x>=0); %x(i) < 0 are zero-prob values
x=(okx.*x); %set zero-prob x(i)=0
cdf= okx.*allcdf(x+1); %zero for zero-prob x(i)
```

Input: `n` and `p` are the parameters of a binomial (n, p) random variable X, `x` is a vector of possible sample values

Output: `y` is a vector with `y(i)` $= F_X$(`x(i)`).

binomialpmf   y=binomialpmf(n,p,x)

```
function pmf=binomialpmf(n,p,x)
%binomial(n,p) rv X,
%input = vector x
%output= vector pmf: pmf(i)=Prob[X=x(i)]
if p<0.5
    pp=p;
else
    pp=1-p;
end
    i=0:n-1;
    ip= ((n-i)./(i+1))*(pp/(1-pp));
    pb=((1-pp)^n)*cumprod([1 ip]);
if pp < p
    pb=fliplr(pb);
end
pb=pb(:); % pb=[P[X=0] ... P[X=n]]^t
x=x(:);
okx =(x>=0).*(x<=n).*(x==floor(x));
x=okx.*x;
pmf=okx.*pb(x+1);
```

Input: `n` and `p` are the parameters of a binomial (n, p) random variable X, `x` is a vector of possible sample values

Output: `y` is a vector with `y(i)` $=$ P_X(`x(i)`).

binomialrv   x=binomialrv(n,p,m)

```
function x=binomialrv(n,p,m)
% m binomial(n,p) samples
r=rand(m,1);
cdf=binomialcdf(n,p,0:n);
x=count(cdf,r);
```

Input: `n` and `p` are the parameters of a binomial random variable X, m is a positive integer

Output: `x` is a vector of `m` independent samples of random variable X

bivariategausspdf

```
function f=bivariategausspdf(muX,muY,sigmaX,sigmaY,rho,x,y)
%Usage: f=bivariategausspdf(muX,muY,sigmaX,sigmaY,rho,x,y)
%Evaluate the bivariate Gaussian (muX,muY,sigmaX,sigmaY,rho) PDF
nx=(x-muX)/sigmaX;
ny=(y-muY)/sigmaY;
f=exp(-((nx.^2) +(ny.^2) - (2*rho*nx.*ny))/(2*(1-rho^2)));
f=f/(2*pi*sigmax*sigmay*sqrt(1-rho^2));
```

Input: Scalar parameters `muX,muY,sigmaX,sigmaY,rho` of the bivariate Gaussian PDF, scalars `x` and `y`.

Output: `f` the value of the bivariate Gaussian PDF at `x,y`.

`duniformcdf` `y=duniformcdf(k,l,x)`

```
function cdf=duniformcdf(k,l,x)
%Usage: cdf=duniformcdf(k,l,x)
% For discrete uniform (k,l) rv X
% and input vector x, output is
% vector cdf: cdf(i)=Prob[X<=x(i)]
x=floor(x(:)); %for noninteger x_i
allx=k:max(x);
%allcdf = cdf values  from 0 to max(x)
allcdf=cumsum(duniformpmf(k,l,allx));
%x_i < k are zero prob values
okx=(x>=k);
%set zero prob x(i)=k
x=((1-okx)*k)+(okx.*x);
%x(i)=0 for zero prob x(i)
cdf= okx.*allcdf(x-k+1);
```

Input: `k` and `l` are the parameters of a discrete uniform (k,l) random variable X, `x` is a vector of possible sample values

Output: `y` is a vector with `y(i)` $= F_X($`x(i)`$)$.

`duniformpmf` `y=duniformpmf(k,l,x)`

```
function pmf=duniformpmf(k,l,x)
%discrete uniform(k,l) rv X,
%input = vector x
%output= vector pmf: pmf(i)=Prob[X=x(i)]
pmf= (x>=k).*(x<=l).*(x==floor(x));
pmf=pmf(:)/(l-k+1);
```

Input: `k` and `l` are the parameters of a discrete uniform (k,l) random variable X, `x` is a vector of possible sample values

Output: `y` is a vector with `y(i)` $= P_X($`x(i)`$)$.

`duniformrv` `x=duniformrv(k,l,m)`

```
function x=duniformrv(k,l,m)
%returns m samples of a discrete
%uniform (k,l) random variable
r=rand(m,1);
cdf=duniformcdf(k,l,k:l);
x=k+count(cdf,r);
```

Input: `k` and `l` are the parameters of a discrete uniform (k,l) random variable X, `m` is a positive integer

Output: `x` is a vector of `m` independent samples of random variable X

`erlangb` `pb=erlangb(rho,c)`

```
function pb=erlangb(rho,c);
%Usage: pb=erlangb(rho,c)
%returns the Erlang-B blocking
%probability for sn M/M/c/c
%queue with load rho
pn=exp(-rho)*poissonpmf(rho,0:c);
pb=pn(c+1)/sum(pn);
```

Input: Offered load `rho` ($\rho = \lambda/\mu$), and the number of servers `c` of an M/M/c/c queue.

Output: `pb`, the blocking probability of the queue

`erlangcdf` `y=erlangcdf(n,lambda,x)`

```
function F=erlangcdf(n,lambda,x)
F=1.0-poissoncdf(lambda*x,n-1);
```

Input: `n` and `lambda` are the parameters of an Erlang random variable X, vector `x`

Output: Vector `y` such that $y_i = F_X(x_i)$.

`erlangpdf` `y=erlangpdf(n,lambda,x)`

```
function f=erlangpdf(n,lambda,x)
f=((lambda^n)/factorial(n))...
    *(x.^(n-1)).*exp(-lambda*x);
```

Input: `n` and `lambda` are the parameters of an Erlang random variable X, vector `x`

Output: Vector `y` such that $y_i = f_X(x_i) = \lambda^n x_i^{n-1} e^{-\lambda x_i}/(n-1)!$.

`erlangrv` `x=erlangrv(n,lambda,m)`

```
function x=erlangrv(n,lambda,m)
y=exponentialrv(lambda,m*n);
x=sum(reshape(y,m,n),2);
```

Input: `n` and `lambda` are the parameters of an Erlang random variable X, integer `m`

Output: Length `m` vector `x` such that each x_i is a sample of X

`exponentialcdf` `y=exponentialcdf(lambda,x)`

```
function F=exponentialcdf(lambda,x)
F=1.0-exp(-lambda*x);
```

Input: `lambda` is the parameter of an exponential random variable X, vector `x`

Output: Vector `y` such that $y_i = F_X(x_i) = 1 - e^{-\lambda x_i}$.

`exponentialpdf` `y=exponentialpdf(lambda,x)`

```
function f=exponentialpdf(lambda,x)
f=lambda*exp(-lambda*x);
f=f.*(x>=0);
```

Input: `lambda` is the parameter of an exponential random variable X, vector `x`

Output: Vector `y` such that $y_i = f_X(x_i) = \lambda e^{-\lambda x_i}$.

`exponentialrv` `x=exponentialrv(lambda,m)`

```
function x=exponentialrv(lambda,m)
x=-(1/lambda)*log(1-rand(m,1));
```

Input: `lambda` is the parameter of an exponential random variable X, integer `m`

Output: Length `m` vector `x` such that each x_i is a sample of X

`finitecdf` `y=finitecdf(sx,p,x)`

```
function cdf=finitecdf(s,p,x)
% finite random variable X:
% vector sx of sample space
% elements {sx(1),sx(2), ...}
% vector px of probabilities
% px(i)=P[X=sx(i)]
% Output is the vector
% cdf: cdf(i)=P[X=x(i)]
cdf=[];
for i=1:length(x)
    pxi= sum(p(find(s<=x(i))));
    cdf=[cdf; pxi];
end
```

Input: `sx` is the range of a finite random variable X, `px` is the corresponding probability assignment, `x` is a vector of possible sample values

Output: `y` is a vector with `y(i)` $= F_X($`x(i)`$)$.

`finitecoeff` `rho=finitecoeff(SX,SY,PXY)`

```
function rho=finitecoeff(SX,SY,PXY);
%Usage:  rho=finitecoeff(SX,SY,PXY)
%Calculate the correlation coefficient rho of
%finite random variables X and Y
ex=finiteexp(SX,PXY); vx=finitevar(SX,PXY);
ey=finiteexp(SY,PXY); vy=finitevar(SY,PXY);
R=finiteexp(SX.*SY,PXY);
rho=(R-ex*ey)/sqrt(vx*vy);
```

Input: Grids `SX`, `SY` and probability grid `PXY` describing the finite random variables X and Y.

Output: `rho`, the correlation coefficient of X and Y

finitecov `covxy=finitecov(SX,SY,PXY)`

```
function covxy=finitecov(SX,SY,PXY);
%Usage: cxy=finitecov(SX,SY,PXY)
%returns the covariance of
%finite random variables X and Y
%given by grids SX, SY, and PXY
ex=finiteexp(SX,PXY);
ey=finiteexp(SY,PXY);
R=finiteexp(SX.*SY,PXY);
covxy=R-ex*ey;
```

Input: Grids `SX`, `SY` and probability grid `PXY` describing the finite random variables X and Y.

Output: `covxy`, the covariance of X and Y.

finiteexp `ex=finiteexp(sx,px)`

```
function ex=finiteexp(sx,px);
%Usage: ex=finiteexp(sx,px)
%returns the expected value E[X]
%of finite random variable X described
%by samples sx and probabilities px
ex=sum((sx(:)).*(px(:)));
```

Input: Probability vector `px`, vector of samples `sx` describing random variable X.

Output: `ex`, the expected value $E[X]$.

finitepmf `y=finitepmf(sx,p,x)`

```
function pmf=finitepmf(sx,px,x)
% finite random variable X:
% vector sx of sample space
% elements {sx(1),sx(2), ...}
% vector px of probabilities
% px(i)=P[X=sx(i)]
% Output is the vector
% pmf: pmf(i)=P[X=x(i)]
pmf=zeros(size(x(:)));
for i=1:length(x)
    pmf(i)= sum(px(find(sx==x(i))));
end
```

Input: `sx` is the range of a finite random variable X, `px` is the corresponding probability assignment, `x` is a vector of possible sample values

Output: `y` is a vector with `y(i)` $=$ $P[X = $ `x(i)`$]$.

finiterv `x=finiterv(sx,p,m)`

```
function x=finiterv(s,p,m)
% returns m samples
% of finite (s,p) rv
%s=s(:);p=p(:);
r=rand(m,1);
cdf=cumsum(p);
x=s(1+count(cdf,r));
```

Input: `sx` is the range of a finite random variable X, `p` is the corresponding probability assignment, `m` is positive integer

Output: `x` is a vector of `m` sample values `y(i)` $=$ $F_X($`x(i)`$)$.

`finitevar`   `v=finitevar(sx,px)`

```
function v=finitevar(sx,px);
%Usage: ex=finitevar(sx,px)
%   returns the variance Var[X]
%   of finite random variables X described by
%   samples sx and probabilities px
ex2=finiteexp(sx.^2,px);
ex=finiteexp(sx,px);
v=ex2-(ex^2);
```

Input: Probability vector `px` and vector of samples `sx` describing random variable X.

Output: `v`, the variance Var[X].

`gausscdf`   `y=gausscdf(mu,sigma,x)`

```
function f=gausscdf(mu,sigma,x)
f=phi((x-mu)/sigma);
```

Input: `mu` and `sigma` are the parameters of an Guassian random variable X, vector `x`

Output: Vector `y` such that $y_i = F_X(x_i) = \Phi((x_i - \mu)/\sigma)$.

`gausspdf`   `y=gausspdf(mu,sigma,x)`

```
function f=gausspdf(mu,sigma,x)
f=exp(-(x-mu).^2/(2*sigma^2))/...
    sqrt(2*pi*sigma^2);
```

Input: `mu` and `sigma` are the parameters of an Guassian random variable X, vector `x`

Output: Vector `y` such that $y_i = f_X(x_i)$.

`gaussrv`   `x=gaussrv(mu,sigma,m)`

```
function x=gaussrv(mu,sigma,m)
x=mu +(sigma*randn(m,1));
```

Input: `mu` and `sigma` are the parameters of an Gaussian random variable X, integer `m`

Output: Length `m` vector `x` such that each x_i is a sample of X

`gaussvector`   `x=gaussvector(mu,C,m)`

```
function x=gaussvector(mu,C,m)
%output: m Gaussian vectors,
%each with mean mu
%and covariance matrix C
if (min(size(C))==1)
    C=toeplitz(C);
end
n=size(C,2);
if (length(mu)==1)
    mu=mu*ones(n,1);
end
[U,D,V]=svd(C);
x=V*(D^(0.5))*randn(n,m)...
    +(mu(:)*ones(1,m));
```

Input: For a Gaussian ($\boldsymbol{\mu}_{\mathbf{X}}$, $\mathbf{C}_{\mathbf{X}}$) random vector $\mathbf{X}$, `gaussvector` can be called in two ways:

- `C` is the $n \times n$ covariance matrix, `mu` is either a length n vector, or a length 1 scalar, `m` is an integer.
- `C` is the length n vector equal to the first row of a symmetric Toeplitz covariance matrix $\mathbf{C}_{\mathbf{X}}$, `mu` is either a length n vector, or a length 1 scalar, `m` is an integer.

If `mu` is a length n vector, then `mu` is the expected value vector; otherwise, each element of $\mathbf{X}$ is assumed to have mean `mu`.

Output: $n \times m$ matrix `x` such that each column `x(:,i)` is a sample vector of $\mathbf{X}$

`gaussvectorpdf`   `f=gaussvector(mu,C,x)`

```
function f=gaussvectorpdf(mu,C,x)
n=length(x);
z=x(:)-mu(:);
f=exp(-z'*inv(C)*z)/...
    sqrt((2*pi)^n*det(C));
```

Input: For a Gaussian ($\boldsymbol{\mu}_{\mathbf{X}}$, $\mathbf{C}_{\mathbf{X}}$) random vector $\mathbf{X}$, `mu` is a length n vector, `C` is the $n \times n$ covariance matrix, `x` is a length n vector.

Output: `f` is the Gaussian vector PDF $f_{\mathbf{X}}(\mathbf{x})$ evaluated at `x`.

`geometriccdf`   `y=geometriccdf(p,x)`

```
function cdf=geometriccdf(p,x)
% for geometric(p) rv X,
%For input vector x, output is vector
%cdf such that cdf_i=Prob(X<=x_i)
x=(x(:)>=1).*floor(x(:));
cdf=1-((1-p).^x);
```

Input: `p` is the parameter of a geometric random variable X, `x` is a vector of possible sample values

Output: `y` is a vector with `y(i)` $=$ F_X(`x(i)`).

`geometricpmf` `y=geometricpmf(p,x)`

```
function pmf=geometricpmf(p,x)
%geometric(p) rv X
%out: pmf(i)=Prob[X=x(i)]
x=x(:);
pmf= p*((1-p).^(x-1));
pmf= (x>0).*(x==floor(x)).*pmf;
```

Input: `p` is the parameter of a geometric random variable X, `x` is a vector of possible sample values

Output: `y` is a vector with `y(i)` $= P_X($`x(i)`$)$.

`geometricrv` `x=geometricrv(p,m)`

```
function x=geometricrv(p,m)
%Usage: x=geometricrv(p,m)
%   returns m samples of a geometric (p) rv
r=rand(m,1);
x=ceil(log(1-r)/log(1-p));
```

Input: `p` is the parameters of a geometric random variable X, m is a positive integer

Output: `x` is a vector of `m` independent samples of random variable X

`icdfrv` `x=icdfrv(@icdf,m)`

```
function x=icdfrv(icdfhandle,m)
%Usage: x=icdfrv(@icdf,m)
%returns m samples of rv X
%with inverse CDF icdf.m
u=rand(m,1);
x=feval(icdfhandle,u);
```

Input: `@icdfrv` is a “handle” (a kind of pointer) to a MATLAB function `icdf.m` that is MATLAB’s representation of an inverse CDF $F_X^{-1}(x)$ of a random variable X, integer `m`

Output: Length `m` vector `x` such that each x_i is a sample of X

`pascalcdf` `y=pascalcdf(k,p,x)`

```
function cdf=pascalcdf(k,p,x)
%Usage: cdf=pascalcdf(k,p,x)
%For a pascal (k,p) rv X
%and input vector x, the output
%is a vector cdf such that
% cdf(i)=Prob[X<=x(i)]
x=floor(x(:)); % for noninteger x(i)
allx=k:max(x);
%allcdf holds all needed cdf values
allcdf=cumsum(pascalpmf(k,p,allx));
%x_i < k have zero-prob,
% other values  are OK
okx=(x>=k);
%set zero-prob x(i)=k,
%just so indexing is not fouled up
x=(okx.*x) +((1-okx)*k);
cdf= okx.*allcdf(x-k+1);
```

Input: `k` and `p` are the parameters of a Pascal (k, p) random variable X, `x` is a vector of possible sample values

Output: `y` is a vector with `y(i)` $= F_X($`x(i)`$)$.

`pascalpmf` `y=pascalpmf(k,p,x)`

```
function pmf=pascalpmf(k,p,x)
%For Pascal (k,p) rv X, and
%input vector x, output is a
%vector pmf: pmf(i)=Prob[X=x(i)]
x=x(:);
n=max(x);
i=(k:n-1)';
ip= [1 ;(1-p)*(i./(i+1-k))];
%pb=all n-k+1 pascal probs
pb=(p^k)*cumprod(ip);
okx=(x==floor(x)).*(x>=k);
%set bad x(i)=k to stop bad indexing
x=(okx.*x) + k*(1-okx);
% pmf(i)=0 unless x(i)  >= k
pmf=okx.*pb(x-k+1);
```

Input: `k` and `p` are the parameters of a Pascal (k, p) random variable X, `x` is a vector of possible sample values

Output: `y` is a vector with `y(i)` $= P_X($`x(i)`$)$.

pascalrv x=pascalrv(k,p,m)

```
function x=pascalrv(k,p,m)
% return m samples of pascal(k,p) rv
r=rand(m,1);
rmax=max(r);
xmin=k;
xmax=ceil(2*(k/p)); %set max range
sx=xmin:xmax;
cdf=pascalcdf(k,p,sx);
while cdf(length(cdf)) <=rmax
    xmax=2*xmax;
    sx=xmin:xmax;
    cdf=pascalcdf(k,p,sx);
end
x=xmin+countless(cdf,r);
```

Input: `k` and `p` are the parameters of a Pascal random variable X, m is a positive integer

Output: `x` is a vector of `m` independent samples of random variable X

phi y=phi(x)

```
function y=phi(x)
sq2=sqrt(2);
y= 0.5 + 0.5*erf(x/sq2);
```

Input: Vector `x`

Output: Vector `y` such that `y(i)` $= \Phi($`x(i)`$)$.

poissoncdf y=poissoncdf(alpha,x)

```
function cdf=poissoncdf(alpha,x)
%output cdf(i)=Prob[X<=x(i)]
x=floor(x(:));
sx=0:max(x);
cdf=cumsum(poissonpmf(alpha,sx));
  %cdf from 0 to max(x)
okx=(x>=0);%x(i)<0 -> cdf=0
x=(okx.*x);%set negative x(i)=0
cdf= okx.*cdf(x+1);
  %cdf=0 for x(i)<0
```

Input: `alpha` is the parameter of a Poisson (α) random variable X, `x` is a vector of possible sample values

Output: `y` is a vector with `y(i)` $= F_X($`x(i)`$)$.

`poissonpmf`   `y=poissonpmf(alpha,x)`

```
function pmf=poissonpmf(alpha,x)
%Poisson (alpha) rv X,
%out=vector pmf: pmf(i)=P[X=x(i)]
x=x(:);
k=(1:max(x))';
logfacts =cumsum(log(k));
pb=exp([-alpha; ...
    -alpha+ (k*log(alpha))-logfacts]);
okx=(x>=0).*(x==floor(x));
x=okx.*x;
pmf=okx.*pb(x+1);
  %pmf(i)=0 for zero-prob x(i)
```

Input: `alpha` is the parameter of a Poisson (α) random variable X, `x` is a vector of possible sample values

Output: `y` is a vector with `y(i)` $=$ $P_X($`x(i)`$)$.

`poissonrv`   `x=poissonrv(alpha,m)`

```
function x=poissonrv(alpha,m)
%return m samples of poisson(alpha) rv X
r=rand(m,1);
rmax=max(r);
xmin=0;
xmax=ceil(2*alpha); %set max range
sx=xmin:xmax;
cdf=poissoncdf(alpha,sx);
%while ( sum(cdf <=rmax) ==(xmax-xmin+1) )
while cdf(length(cdf)) <=rmax
    xmax=2*xmax;
    sx=xmin:xmax;
    cdf=poissoncdf(alpha,sx);
end
x=xmin+countless(cdf,r);
```

Input: `alpha` is the parameter of a Poisson (α) random variable X, `m` is a positive integer

Output: `x` is a vector of `m` independent samples of random variable X

`uniformcdf`   `y=uniformcdf(a,b,x)`

```
function F=uniformcdf(a,b,x)
%Usage: F=uniformcdf(a,b,x)
%returns the CDF of a continuous
%uniform rv evaluated at x
F=x.*((x>=a) & (x<b))/(b-a);
F=f+1.0*(x>=b);
```

Input: `a` and `(` b) are parameters for continuous uniform random variable X, vector `x`

Output: Vector `y` such that $y_i = F_X(x_i)$

`uniformpdf` `y=uniformpdf(a,b,x)`

```
function f=uniformpdf(a,b,x)
%Usage: f=uniformpdf(a,b,x)
%returns the PDF of a continuous
%uniform rv evaluated at x
f=((x>=a) & (x<b))/(b-a);
```

Input: `a` and `(` b) are parameters for continuous uniform random variable X, vector `x`

Output: Vector `y` such that $y_i = f_X(x_i)$

`uniformrv` `x=uniformrv(a,b,m)`

```
function x=uniformrv(a,b,m)
%Usage: x=uniformrv(a,b,m)
%Returns m samples of a
%uniform (a,b) random varible
x=a+(b-a)*rand(m,1);
```

Input: `a` and `(` b) are parameters for continuous uniform random variable X, positive integer `m`

Output: `m` element vector `x` such that each `x(i)` is a sample of X.

Functions for Stochastic Processes

`brownian` `w=brownian(alpha,t)`

```
function w=brownian(alpha,t)
%Brownian motion process
%sampled at t(1)<t(2)< ...
t=t(:);
n=length(t);
delta=t-[0;t(1:n-1)];
x=sqrt(alpha*delta).*gaussrv(0,1,n);
w=cumsum(x);
```

Input: `t` is a vector holding an ordered sequence of inspection times, `alpha` is the scaling constant of a Brownian motion process such that the ith increment has variance $\alpha(t_i - t_{i-1})$.

Output: `w` is a vector such that `w(i)` is the position at time `t(i)` of the particle in Brownian motion.

`cmcprob` `pv=cmcprob(Q,p0,t)`

```
function pv = cmcprob(Q,p0,t)
%Q has zero diagonal rates
%initial state probabilities p0
K=size(Q,1)-1; %max no. state
%check for integer p0
if (length(p0)==1)
    p0=((0:K)==p0);
end
R=Q-diag(sum(Q,2));
pv= (p0(:)'*expm(R*t))';
```

Input: $n \times n$ state transition matrix `Q` for a continuous-time finite Markov chain, length n vector `p0` denoting the initial state probabilities, nonengative scalar `t`

Output: Length `n` vector `pv` such that `pv(t)` is the state probability vector at time `t` of the Markov chain

Comment: If `p0` is a scalar integer, then the simulation starts in state `p0`

`cmcstatprob` `pv=cmcstatprob(Q)`

```
function pv = cmcstatprob(Q)
%Q has zero diagonal rates
R=Q-diag(sum(Q,2));
n=size(Q,1);
R(:,1)=ones(n,1);
pv=([1 zeros(1,n-1)]*R^(-1))';
```

Input: State transition matrix `Q` for a continuous-time finite Markov chain

Output: `pv` is the stationary probability vector for the continuous-time Markov chain

`dmcstatprob` `pv=dmcstatprob(P)`

```
function pv = dmcstatprob(P)
n=size(P,1);
A=(eye(n)-P);
A(:,1)=ones(n,1);
pv=([1 zeros(1,n-1)]*A^(-1))';
```

Input: $n \times n$ stochastic matrix `P` representing a discrete-time aperiodic irreducible finite Markov chain

Output: `pv` is the stationary probability vector.

`poissonarrivals` `s=poissonarrivals(lambda,T)`

```
function s=poissonarrivals(lambda,T)
%arrival times s=[s(1) ... s(n)]
%  s(n)<= T < s(n+1)
n=ceil(1.1*lambda*T);
s=cumsum(exponentialrv(lambda,n));
while (s(length(s))< T),
  s_new=s(length(s))+ ...
    cumsum(exponentialrv(lambda,n));
  s=[s; s_new];
end
s=s(s<=T);
```

Input: `lambda` is the arrival rate of a Poisson process, `T` marks the end of an observation interval $[0, T]$.

Output: `s=[s(1), ..., s(n)]'` is a vector such that `s(i)` is ith arrival time. Note that length `n` is a Poisson random variable with expected value λT.

Comment: This code is pretty stupid. There are decidedly better ways to create a set of arrival times; see Problem 10.13.5.

`poissonprocess` `N=poissonprocess(lambda,t)`

```
function N=poissonprocess(lambda,t)
%input: rate lambda>0, vector t
%For a sample function of a
%Poisson process of rate lambda,
%N(i) = no. of arrivals by t(i)
s=poissonarrivals(lambda,max(t));
N=count(s,t);
```

Input: `lambda` is the arrival rate of a Poisson process, `t` is a vector of "inspection times'.'

Output: `N` is a vector such that `N(i)` is the number of arrival by inspection time `t(i)`.

`simcmc` `ST=simcmc(Q,p0,T)`

```
function ST=simcmc(Q,p0,T);
K=size(Q,1)-1; max no. state
%calc  average trans. rate
ps=cmcstatprob(Q);
v=sum(Q,2); R=ps'*v;
n=ceil(0.6*T/R);
ST=simcmcstep(Q,p0,2*n);
while (sum(ST(:,2))<T),
    s=ST(size(ST,1),1);
    p00=Q(1+s,:)/v(1+s);
    S=simcmcstep(Q,p00,n);
    ST=[ST;S];
end
n=1+sum(cumsum(ST(:,2))<T);
ST=ST(1:n,:);
%truncate last holding time
ST(n,2)=T-sum(ST(1:n-1,2));
```

Input: state transition matrix `Q` for a continuous-time finite Markov chain, vector `p0` denoting the initial state probabilities, integer `n`

Output: A simulation of the Markov chain system over the time interval $[0, T]$: The output is an $n \times 2$ matrix `ST` such that the first column `ST(:,1)` is the sequence of system states and the second column `ST(:,2)` is the amount of time spent in each state. That is, `ST(i,2)` is the amount of time the system spends in state `ST(i,1)`.

Comment: If `p0` is a scalar integer, then the simulation starts in state `p0`. Note that n, the number of state occupancy periods, is random.

`simcmcstep`   `S=simcmcstep(Q,p0,n)`

```
function S=simcmcstep(Q,p0,n);
%S=simcmcstep(Q,p0,n)
%  Simulate n steps of a cts
%  Markov Chain, rate matrix Q,
%  init. state probabilities p0
K=size(Q,1)-1; %max no. state
S=zeros(n+1,2);%init allocation
%check for integer p0
if (length(p0)==1)
    p0=((0:K)==p0);
end
v=sum(Q,2); %state dep. rates
t=1./v;
P=diag(t)*Q;
S(:,1)=simdmc(P,p0,n);
S(:,2)=t(1+S(:,1)) ...
    .*exponentialrv(1,n+1);
```

Input: State transition matrix `Q` for a continuous-time finite Markov chain, vector `p0` denoting the initial state probabilities, integer `n`

Output: A simulation of *n* steps of the continuous-time Markov chain system: The output is an $n \times 2$ matrix `ST` such that the first column `ST(:,1)` is the length *n* sequence of system states and the second column `ST(:,2)` is the amount of time spent in each state. That is, `ST(i,2)` is the amount of time the system spends in state `ST(i,1)`.

Comment: If `p0` is a scalar integer, then the simulation starts in state `p0`. This program is the basis for `simcmc`.

`simdmc`   `x=simdmc(P,p0,n)`

```
function x=simdmc(P,p0,n)
K=size(P,1)-1;             %highest no. state
sx=0:K;                    %state space
x=zeros(n+1,1);            %initialization
if (length(p0)==1)         %convert integer p0 to prob vector
    p0=((0:K)==p0);
end
x(1)=finiterv(sx,p0,1);    %x(m)= state at time m-1
for m=1:n,
  x(m+1)=finiterv(sx,P(x(m)+1,:),1);
end
```

Input: $n \times n$ stochastic matrix `P` which is the state transition matrix of a discrete-time finite Markov chain, length *n* vector `p0` denoting the initial state probabilities, integer `n`.

Output: A simulation of the Markov chain system such that for the length `n` vector `x`, `x(m)` is the state at time `m-1` of the Markov chain.

Comment: If `p0` is a scalar integer, then the simulation starts in state `p0`

Random Utilities

`count` `n=count(x,y)`

```
function n=count(x,y)
     %Usage n=count(x,y)
%n(i)= # elements of x <= y(i)
[MX,MY]=ndgrid(x,y);
%each column of MX = x
%each row of MY = y
n=(sum((MX<=MY),1))';
```

Input: Vectors `x` and `y`

Output: Vector `n` such that $n(i)$ is the number of elements of `x` less than or equal to `y(i)`.

`countequal` `n=countequal(x,y)`

```
function n=countequal(x,y)
%Usage:  n=countequal(x,y)
%n(j)= # elements of x = y(j)
[MX,MY]=ndgrid(x,y);
%each column of MX = x
%each row of MY = y
n=(sum((MX==MY),1))';
```

Input: Vectors `x` and `y`

Output: Vector `n` such that $n(i)$ is the number of elements of `x` equal to `y(i)`.

`countless` `n=countless(x,y)`

```
function n=countless(x,y)
%Usage:  n=countless(x,y)
%n(i)= # elements of x < y(i)
[MX,MY]=ndgrid(x,y);
%each column of MX = x
%each row of MY = y
n=(sum((MX<MY),1))';
```

Input:

Input: Vectors `x` and `y`

Output: Vector `n` such that $n(i)$ is the number of elements of `x` strictly less than `y(i)`.

`dftmat` `F=dftmat(N)`

```
function F = dftmat(N);
Usage: F=dftmat(N)
%F is the N by N DFT matrix
n=(0:N-1)';
F=exp((-1.0j)*2*pi*(n*(n'))/N);
```

Input: Integer N.

Output: `F` is the `N` by `N` discrete Fourier transform matrix `freqxy` `fxy=freqxy(xy,SX,SY)`

```
function fxy = freqxy(xy,SX,SY)
%Usage: fxy = freqxy(xy,SX,SY)
%xy is an m x 2 matrix:
%xy(i,:)= ith sample pair X,Y
%Output fxy is a K x 3 matrix:
% [fxy(k,1) fxy(k,2)]
%   = kth unique pair [x y] and
%   fxy(k,3)= corresp. rel. freq.

%extend xy to include a sample
%for all possible (X,Y) pairs:
xy=[xy; SX(:) SY(:)];
[U,I,J]=unique(xy,'rows');
N=hist(J,1:max(J))-1;
N=N/sum(N);
fxy=[U N(:)];
%reorder fxy rows to match
%rows of [SX(:) SY(:) PXY(:)]:
fxy=sortrows(fxy,[2 1 3]);
```

Input: For random variables X and Y, `xy` is an $m \times 2$ matrix holding a list of sample values pairs; `yy(i,:)` is the ith sample pair (X, Y). Grids `SX` and `SY` representing the sample space.

Output: `fxy` is a $K \times 3$ matrix. In each row

`[fxy(k,1) fxy(k,2) fxy(k,3)]`

`[fxy(k,1) fxy(k,2)]` is a unique (X, Y) pair with relative frequency `fxy(k,3)`.

Comment: Given the grids `SX`, `SY` and the probability grid `PXY`, a list of random sample value pairs `xy` can be simulated by the commands

```
S=[SX(:) SY(:)];
xy=finiterv(S,PXY(:),m);
```

The output `fxy` is ordered so that the rows match the ordering of rows in the matrix

`[SX(:) SY(:) PXY(:)].`

`fftc` `S=fftc(r,N); S=fftc(r)`

```
function S=fftc(varargin);
%DFT for a signal r
%centered at the origin
%Usage:
%  fftc(r,N): N point DFT of r
%  fftc(r): length(r) DFT of r
r=varargin{1};
L=1+floor(length(r)/2);
if (nargin>1)
    N=varargin{2}(1);
else
    N=(2*L)-1;
end
R=fft(r,N);
n=reshape(0:(N-1),size(R));
phase=2*pi*(n/N)*(L-1);
S=R.*exp((1.0j)*phase);
```

Input: Vector `r=[r(1) ... r(2k+1)]` holding the time sequence $r_{-k}, \ldots, r_0, \ldots, r_k$ centered around the origin.

Output: `S` is the DFT of `r`

Comment: Supports the same calling conventions as `fft`.

`pmfplot` `pmfplot(sx,px,'x','y axis text')`

```
function h=pmfplot(sx,px,xls,yls)
%Usage: pmfplot(sx,px,xls,yls)
%sx and px are vectors, px is the PMF
%xls and yls are x and y label strings
nonzero=find(px);
sx=sx(nonzero); px=px(nonzero);
sx=(sx(:))'; px=(px(:))';
XM = [sx; sx];
PM=[zeros(size(px)); px];
h=plot(XM,PM,'-k');
set(h,'LineWidth',3);
if (nargin==4)
  xlabel(xls);
  ylabel(yls,'VerticalAlignment','Bottom');
end
xmin=min(sx); xmax=max(sx);
xborder=0.05*(xmax-xmin);
xmax=xmax+xborder;
xmin=xmin-xborder;
ymax=1.1*max(px);
axis([xmin xmax 0 ymax]);
```

Input: Sample space vector `sx` and PMF vector `px` for finite random variable `PXY`, optional text strings `xls` and `yls`

Output: A plot of the PMF $P_X(x)$ in the bar style used in the text.

`rect` `y=rect(x)`

```
function y=rect(x);
%Usage:y=rect(x);
y=1.0*(abs(x)<0.5);
```

Input: Vector `x`

Output: Vector `y` such that

$$y_i = \text{rect}(x_i) = \begin{cases} 1 & |x_i| < 0.5 \\ 0 & \text{otherwise} \end{cases}$$

`sinc` `y=sinc(x)`

```
function y=sinc(x);
xx=x+(x==0);
y=sin(pi*xx)./(pi*xx);
y=((1.0-(x==0)).*y)+ (1.0*(x==0));
```

Input: Vector `x`

Output: Vector `y` such that

$$y_i = \text{sinc}(x_i) = \frac{\sin(\pi x_i)}{\pi x_i}$$

Comment: The code is ugly because it makes sure to produce the right limit value at $x_i = 0$.

`simplot` `simplot(S,xlabel,ylabel)`

```
function h=simplot(S,xls,yls);
%h=simplot(S,xlabel,ylabel)
%   Plots the output of a simulated state sequence
%   If S is N by 1, a discrete time chain is assumed
%   with visit times of one unit.
%   If S is an N by 2 matrix, a cts time Markov chain
%   is assumed where
%   S(:,1) = state sequence.
%   S(:,2) = state visit times.
%   The cumulative sum
%   of visit times are transition instances.
%   h is a handle to a stairs plot of the state sequence
%       vs state transition times

%in case of discrete time simulation
if (size(S,2)==1)
    S=[S ones(size(S))];
end
Y=[S(:,1) ; S(size(S,1),1)];
X=cumsum([0 ; S(:,2)]);
h=stairs(X,Y);
if (nargin==3)
xlabel(xls);
ylabel(yls,'VerticalAlignment','Bottom');
end
```

Input: The simulated state sequence vector `S` generated by `S=simdmc(P,p0,n)` or the $n \times 2$ state/time matrix `ST` generated by either

`ST=simcmc(Q,p0,T)`

or

`ST=simcmcstep(Q,p0,n).`

Output: A "stairs" plot showing the sequence of simulation states over time.

Comment: If `S` is just a state sequence vector, then each stair has equal width. If `S` is $n \times 2$ state/time matrix `ST`, then the width of the stair is proportional to the time spent in that state.