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 \[ y = \text{bernoullipmf}(p, x) \]

```matlab
function pv = bernoullipmf(p, x)
% For Bernoulli (p) rv X
% input = vector x
% output = vector pv
% such that pv(i) = \text{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 \[ y = \text{bernoullicdf}(p, x) \]

```matlab
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) = \text{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 \[ x = \text{bernoullirv}(p, m) \]

```matlab
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  \[ y = \text{bignomialpmf}(n, p, x) \]

```matlab
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 `bignomialpmf(n,p,x)`, however, the function calculates the logarithm of the probability and this may lead to small numerical inaccuracy.

binomialcdf  \[ y = \text{binomialcdf}(n, p, x) \]

```matlab
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 \( y(i) = F_X(x(i)) \).
\begin{verbatim}
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));

pmf=okx.*pb(x+1);
end
\end{verbatim}

**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 \( y(i) = P_X(x(i)) \).

\begin{verbatim}
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);
end
\end{verbatim}

**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 \( X \).

\begin{verbatim}
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));
end
\end{verbatim}

**Input:** Scalar parameters \( \mu_X, \mu_Y, \sigma_X, \sigma_Y, \rho \) of the bivariate Gaussian PDF, scalars \( x \) and \( y \).

**Output:** \( f \) the value of the bivariate Gaussian PDF at \( x, y \).
**duniformcdf**

\[ y = \text{duniformcdf}(k, l, x) \]

```matlab
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) = \Pr[X <= x(i)]
    x = floor(x(:)); % for noninteger x_i
    allx = k: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);
end
```

**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**

\[ y = \text{duniformpmf}(k, l, x) \]

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

**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**

\[ x = \text{duniformrv}(k, l, m) \]

```matlab
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);
end
```

**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 \(X\)
\textbf{erlangb} \hspace{1cm} \texttt{pb=erlangb(rho,c)}

\begin{verbatim}
function pb=erlangb(rho,c);
%Usage: pb=erlangb(rho,c)
%returns the Erlang-B blocking
%probability for an M/M/c/c
%queue with load rho
pn=exp(-rho)*poissonpmf(rho,0:c);
pb=pn(c+1)/sum(pn);
\end{verbatim}

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

\textbf{Output:} $pb$, the blocking probability of the queue.

\textbf{erlangcdf} \hspace{1cm} \texttt{y=erlangcdf(n,lambda,x)}

\begin{verbatim}
function F=erlangcdf(n,lambda,x)
F=1.0-poissoncdf(lambda*x,n-1);
\end{verbatim}

\textbf{Input:} $n$ and $\lambda$ are the parameters of an Erlang random variable $X$, vector $x$

\textbf{Output:} Vector $y$ such that $y_i = F_X(x_i)$.

\textbf{erlangpdf} \hspace{1cm} \texttt{y=erlangpdf(n,lambda,x)}

\begin{verbatim}
function f=erlangpdf(n,lambda,x)
f=((lambda^n)/factorial(n))...*(x.^(n-1)).*exp(-lambda*x);
\end{verbatim}

\textbf{Input:} $n$ and $\lambda$ are the parameters of an Erlang random variable $X$, vector $x$

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

\textbf{erlangrv} \hspace{1cm} \texttt{x=erlangrv(n,lambda,m)}

\begin{verbatim}
function x=erlangrv(n,lambda,m)
y=exponentialrv(lambda,m*n);
x=sum(reshape(y,m,n),2);
\end{verbatim}

\textbf{Input:} $n$ and $\lambda$ are the parameters of an Erlang random variable $X$, integer $m$

\textbf{Output:} Length $m$ vector $x$ such that each $x_i$ is a sample of $X$.

\textbf{exponentialcdf} \hspace{1cm} \texttt{y=exponentialcdf(lambda,x)}

\begin{verbatim}
function F=exponentialcdf(lambda,x)
F=1.0-exp(-lambda*x);
\end{verbatim}

\textbf{Input:} $\lambda$ is the parameter of an exponential random variable $X$, vector $x$

\textbf{Output:} Vector $y$ such that $y_i = F_X(x_i) = 1 - e^{-\lambda x_i}$.
exponentialpdf   \( y = \text{exponentialpdf}(\lambda, x) \)

\[
\begin{align*}
\text{function } & f = \text{exponentialpdf}(\lambda, x) \\
& f = \lambda \cdot \exp(-\lambda \cdot x); \\
& f = f \cdot (x \geq 0);
\end{align*}
\]

**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   \( x = \text{exponentialrv}(\lambda, m) \)

\[
\begin{align*}
\text{function } & x = \text{exponentialrv}(\lambda, m) \\
x = -(1/\lambda) \cdot \log(1 - \text{rand}(m, 1));
\end{align*}
\]

**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   \( y = \text{finitecdf}(sx, p, x) \)

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

**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   \( \rho = \text{finitecoeff}(SX, SY, PXY) \)

\[
\begin{align*}
\text{function } & \rho = \text{finitecoeff}(SX, SY, PXY); \\
& \% \text{Usage: } \rho = \text{finitecoeff}(SX, SY, PXY) \\
& \% \text{Calculate the correlation coefficient } \rho \text{ of finite random variables } X \text{ and } Y \\
& ex = \text{finiteexp}(SX, PXY); \ vx = \text{finitevar}(SX, PXY); \\
& ey = \text{finiteexp}(SY, PXY); \ vy = \text{finitevar}(SY, PXY); \\
& R = \text{finiteexp}(SX \cdot SY, PXY); \\
& \rho = (R - ex * ey) / \sqrt{vx * vy};
\end{align*}
\]

**Input:** Grids \( SX, SY \) and probability grid \( PXY \) describing the finite random variables \( X \) and \( Y \).

**Output:** \( \rho \), the correlation coefficient of \( X \) and \( Y \).
function covxy=f infinite cov (SX, SY, PXY);
%Usage: cxy=f infinite cov (SX, SY, PXY)
%returns the covariance of
%finite random variables X and Y
given by grids SX, SY, and PXY
ex=f infinite exp (SX, PXY);
ey=f infinite exp (SY, PXY);
R=f infinite exp (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.

function ex=f infinite exp (sx, px);
%Usage: ex=f infinite exp (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].

function pmf=f infinite pmf (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)].

function x=f inite r v (sx, 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-(exˆ2);

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 a Gaussian random variable X, vector x
Output: Vector y such that yi = FX(x_i) = Φ((x_i − µ)/σ).

gausspdf  
y=gausspdf(mu,sigma,x)

function f=gausspdf(mu,sigma,x)
f=exp(-(x-mu).^2/(2*sigmaˆ2))/...
sqrt(2*pi*sigmaˆ2);

Input: mu and sigma are the parameters of a Gaussian random variable X, vector x
Output: Vector y such that yi = 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 a Gaussian random variable X, integer m
Output: Length m vector x such that each x_i is a sample of X.
gaussvector \( x=\text{gaussvector}(\mu, C, m) \)

\begin{verbatim}
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*(Dˆ(0.5))*randn(n,m)...+
        (mu(:)*ones(1,m));
\end{verbatim}

**Input:** For a Gaussian \((\mu_X, C_X)\) random vector \(X\), \texttt{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 \(C_X\), \(\mu\) 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 \(X\) is assumed to have mean \(\mu\).

**Output:** \(n \times m\) matrix \(x\) such that each column \(x(:,i)\) is a sample vector of \(X\).

gaussvectorpdf \( f=\text{gaussvectorpdf}(\mu, C, x) \)

\begin{verbatim}
function f=gaussvectorpdf(mu,C,x)
    n=length(x);
    z=x(:)-mu(:);
    f=exp(-z'\times inv(C)\times z)/...
        sqrt((2*pi)ˆn*det(C));
\end{verbatim}

**Input:** For a Gaussian \((\mu_X, C_X)\) random vector \(X\), \(\mu\) 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 \(f_X(x)\) evaluated at \(x\).

goodfitness \( y=\text{geometriccdf}(p,x) \)

\begin{verbatim}
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);
\end{verbatim}

**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  \[ y = \text{geometricpmf}(p, x) \]

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

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  \[ x = \text{geometricrv}(p, m) \]

\[
\text{function x=geometricrv(p,m)} \\
\% \text{Usage: x=geometricrv(p,m)} \\
\% \text{returns m samples of a geometric (p) rv} \\
\text{r=rand(m,1);} \\
\text{x=ceil(log(1-r)/log(1-p));}
\]

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  \[ x = \text{icdfrv}(@icdf, m) \]

\[
\text{function x=icdfrv(icdfhandle,m)} \\
\% \text{Usage: x=icdfrv(@icdf,m)} \\
\% \text{returns m samples of rv } X \\
\text{u=rand(m,1);} \\
\text{x=feval(icdfhandle,u);}
\]

Input: \( @icdf \) is a “handle” (a kind of pointer) to a MATLAB function \( \text{icdf.m} \) 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 \( X \).
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)]
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);

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(:);
% p is all n-k+1 pascal probs
p=(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) = F_X(x(i)).
**pascalrv**

```matlab
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**

```matlab
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**

```matlab
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 (\( \alpha \)) random variable \( X \), \( x \) is a vector of possible sample values  

**Output:** \( y \) is a vector with \( y(i) = F_X(x(i)) \).
function pmf=poissonpmf(alpha,x)
%Poisson (alpha) rv X,
%out=vector pmf: pmf(i)=P[X=x(i)]
\[ \text{x} = \text{x}(:); \]
\[ \text{k} = (1:max(x))'; \]
\[ \text{logfacts} = \text{cumsum} (\log (\text{k})); \]
\[ \text{pb} = \exp (-\text{alpha}; \]
\[ \quad -\text{alpha} + (\text{k} \times \text{log(alpha)}) - \text{logfacts}; \]
\[ \quad \text{okx} = (\text{x} >= 0) .*(\text{x} == \text{floor} (\text{x})); \]
\[ \text{x} = \text{okx} .* \text{x}; \]
\[ \text{pmf} = \text{okx} .* \exp (\text{pb}(\text{x}+1)); \]
%pmf(i)=0 for zero-prob x(i)

Input: alpha is the parameter of a Poisson (\( \alpha \)) random variable \( X \), \( x \) is a vector of possible sample values
Output: \( y \) is a vector with \( y(i) = P_X(x(i)) \).

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

Input: alpha is the parameter of a Poisson (\( \alpha \)) random variable \( X \), \( m \) is a positive integer
Output: \( x \) is a vector of \( m \) independent samples of random variable \( X \)

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

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 \quad y=\text{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 \quad x=\text{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 variable
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  
\[ w = \text{brownian}(\alpha, t) \]

```matlab
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);
end
```

**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 \( i \)th 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  
\[ pv = \text{cmcprob}(Q,p0,t) \]

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

**Input:** \( n \times n \) state transition matrix \( Q \) for a continuous-time finite Markov chain, length \( n \) vector \( p0 \) denoting the initial state probabilities, nonnegative 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  
\[ pv = \text{cmcstatprob}(Q) \]

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

**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  
\[ pv = \text{dmcstatprob}(P) \]

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

**Input:** \( n \times n \) stochastic matrix \( P \) representing a discrete-time aperiodic irreducible finite Markov chain

**Output:** \( pv \) is the stationary probability vector.
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);
end

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);

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(s,:)/v(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: 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 i'th arrival time. Note that length n is a Poisson random variable with expected value \lambda T.
Comment: This code is pretty stupid. There are decidedly better ways to create a set of arrival times; see Problem 10.13.5.

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).

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 × 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 \quad 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 \quad 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

\[ n = \text{count}(x, y) \]

\begin{verbatim}
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))';
end
\end{verbatim}

Input: Vectors \( x \) and \( y \)
Output: Vector \( n \) such that \( n(i) \) is the number of elements of \( x \) less than or equal to \( y(i) \).

\[ n = \text{countequal}(x, y) \]

\begin{verbatim}
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))';
end
\end{verbatim}

Input: Vectors \( x \) and \( y \)
Output: Vector \( n \) such that \( n(i) \) is the number of elements of \( x \) equal to \( y(i) \).

\[ n = \text{countless}(x, y) \]

\begin{verbatim}
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))';
end
\end{verbatim}

Input: Vectors \( x \) and \( y \)
Output: Vector \( n \) such that \( n(i) \) is the number of elements of \( x \) strictly less than \( y(i) \).

\[ F = \text{dftmat}(N) \]

\begin{verbatim}
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);
end
\end{verbatim}

Input: Integer \( N \).
Output: \( F \) is the \( N \) by \( N \) discrete Fourier transform matrix.
freqxy

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]);

fftc

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=fftc(r,N);
n=reshape(0:(N-1),size(R));
phase=2*pi*(n/N)*(L-1);
S=R.*exp((1.0j)*phase);
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
\[ x_i = 0. \]
function h=simplot(S,xlabel,ylabel);
% 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 $S = \text{simdmc}(P,p0,n)$ or the $n \times 2$ state/time matrix ST generated by either

$$ST = \text{simcmc}(Q,p0,T)$$

or

$$ST = \text{simcmcstep}(Q,p0,n).$$

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.