Dirk P. Kroese and Joshua C.C. Chan

Statistical Modeling and Computation

An Inclusive Approach to Statistics

June 3, 2013

Springer

Appendix A
Matlab Primer

MATLAB, a portmanteau of MATrix LABoratory, is an interactive matrix-based program for numerical computation. It is a very easy to use high-level language that requires minimal programming skills. The purpose of this appendix is to introduce the reader to some basic MATLAB functions that are used in the main text. For more detailed information and full documentation, please visit the official documentation site

`http://www.mathworks.com/help/techdoc/`.

In addition, the command `help` *function_name* gives information about the function *function_name*. Alternatively, select `Help -> Product Help` in the toolbar in the MATLAB command window.

A.1 Matrices and Matrix Operations

The most fundamental objects in MATLAB are, not surprisingly, matrices. For instance, to create a 1×3 matrix (that is, row vector) $\mathbf{a}$, enter into the MATLAB command window:

```
a = [1 2 3]
```

MATLAB returns

```
a =
      1 2 3
```

To create a matrix with more than one row, use semi-colons to separate the rows. For example, the line

```
A = [1 2 3; 4 5 6; 7 8 9];
```

creates a 3×3 matrix A. It is worth noting that MATLAB is case sensitive for variable names and built-in functions. That means MATLAB treats a and A as different objects. To display the i-th element in a vector **x**, just type `x(i)`. For example,

```
a(2)
```

refers to the second element of **a**. Similarly, one can access a particular element of A by specifying its row and column number (row first followed by column). For instance,

```
A(2,3)
```

displays the $(2, 3)$-entry of the matrix A. To display multiple elements in the matrix, one can use expressions involving colons. For example,

```
A(1,1:2)
```

displays the first and second element in the first row, whereas

```
A(:,2)
```

displays all the elements in the second column.

To perform numerical computation, one needs some basic matrix operations. In MATLAB, the following matrix operations, among many others, are available:

+ addition	/ right division
− subtraction	^ power
∗ multiplication	′ transpose
\ left division	

For example,

```
a'
```

returns the transpose of **a**:

```
ans =
```

```
     1
     2
     3
```

whereas

```
a*A
```

gives the product of $\mathbf{a}$ and A:

```
ans =

    30    36    42
```

Other operations are obvious, except for the matrix divisions \ and /. If A is an invertible square matrix and $\mathbf{a}$ is a compatible vector, then $\mathbf{x} = A\backslash\mathbf{a}$ is the solution of $A\,\mathbf{x} = \mathbf{a}$ and $\mathbf{x} = \mathbf{a}/A$ is the solution of $\mathbf{x}\,A = \mathbf{a}$. In other words, $A\backslash\mathbf{a}$ gives the same result (in principle) as $A^{-1}\,\mathbf{a}$, though they compute their results in different ways. Specifically, the former solves the linear system $A\,\mathbf{x} = \mathbf{a}$ for $\mathbf{x}$ by Gaussian elimination, whereas the latter first computes the inverse A^{-1}, then multiplies it by $\mathbf{a}$. As such, the second method is in general slower as computing the inverse of a matrix is time-consuming (and inaccurate).

It is important to note that although addition and subtraction are element-wise operations, the other operations listed above are not — they are matrix operations. For example, $A\hat{}2$ gives the square of the matrix A, not a matrix whose entries are the squares of those in A. One can make the operations $*$, \, /, and $\hat{}$ to operate element-wise by preceding them by a full stop. For example,

```
A^2
```

returns the square of the matrix A:

```
ans =

    30    36    42
    66    81    96
   102   126   150
```

On the other hand,

```
A.^2
```

computes the squares element-wise:

```
ans =

     1     4     9
    16    25    36
    49    64    81
```

A.2 Some Useful Built-in Functions

In this section we list some common built-in functions which are used throughout the main text. One can learn more about a specific function, say, `eye`, by typing `help eye` in the command window. Here are some useful matrix-building functions:

`eye`	create an identity matrix
`zeros`	create a matrix of zeros
`ones`	create a matrix of ones
`diag`	create a diagonal matrix or extract the diagonal from a matrix
`rand`	generate $\mathsf{U}(0,1)$ random variables
`randn`	generate $\mathsf{N}(0,1)$ random variables

For example, `eye(n)` creates an $n \times n$ identity matrix, and `ones(m,n)` produces an $m \times n$ matrix of ones. Given the 3×3 matrix A,

```
diag(A)
```

extracts the diagonal of the matrix A:

```
ans =

     1
     5
     9
```

But for the 3×1 vector $\mathbf{a}$, the same command

```
diag(a)
```

builds a diagonal matrix whose main diagonal is **a**:

```
ans =

     1     0     0
     0     2     0
     0     0     3
```

Some other useful vector and matrix functions:

`exp`	exponential	`log`	natural log
`sqrt`	square root	`abs`	absolute value
`sin`	sine	`cos`	cosine
`sum`	sum	`prod`	product
`max`	maximum	`min`	minimum
`chol`	Cholesky factorization	`inv`	inverse
`det`	determinant	`size`	size

If **x** is a vector, `sum(x)` returns the sum of the elements in **x**. For a matrix X, `sum(X)` returns a row vector consisting of sums of each column, while `sum(X,2)` returns a column vector of sums of each row. For example,

```
sum(A)
```

returns

```
ans =

    12    15    18
```

whereas

```
sum(A,2)
```

gives

```
ans =

     6
    15
    24
```

For a positive definite matrix C, `chol(C,'lower')` returns the lower Cholesky factorization B such that $BB^\top = C$. For example,

```
B = [ 1 0 0; 2 3 0; 4 5 6];
```

```
C = B*B';
chol(C,'lower')
```

returns the lower Cholesky factor of $BB^\top$, which is, of course, B.

A.3 Flow Control

MATLAB has the usual control flow statements such as `if-then-else`, `while`, and `for`. For instance, the general form of a simple `if` statement is

```
if condition
   statements
end
```

The statements will be executed if the condition is true. Multiple branching is done by using `elseif` and `else`. For example, the following code simulate rolling a four-sided die:

```
u = rand;
if u <= .25
    disp('1');
elseif u <= .5
    disp('2');
elseif u <= .75
    disp('3');
else
    disp('4');
end
```

The general form of a `while` loop is

```
while condition
   statements
end
```

The statements will be repeatedly executed while the condition remains true. To illustrate the `while` loop syntax, suppose we wish to generate a positive
☞ 55 normal random variable (with pdf given in (2.25)). We can do that using the following simple `while` loop:

```
u = randn;
while u <= 0
    u = randn;
end
```

Another useful control flow statement is the `for` loop, whose general form is

```
for count
   statements
end
```

Unlike a `while` loop, the `for` loop executes the statements for a fixed number of times. As an example, the following code generates five draws from the positive normal distribution.

```
x = zeros(1,5);  %% create a storage vector
for i=1:5
    u = randn;
    while u <= 0
        u = randn;
    end
    x(i)=u;
end
```

A.4 Function Handles and Function Files

In previous sections we have introduced some built-in functions in MATLAB. For instance, `sqrt` is a function that takes an argument and returns an output (its square root). Later on we will need to create our own functions that take one or more input arguments, operate on them, and return the results. One way to create new functions is through function handles. For example,

```
f = @(x) x.^2 + 5*x - 10 ;  % Note the use of the dot
```

creates the function $f(x) = x^2 + 5x - 10$ with the function handle `f`. Technically, the above command creates an anonymous function with a function handle called `f`. The function handle gives you a means of invoking the function. To evaluate, say, $f(10)$, we can type `feval(f,10)`, or simply, `f(10)`.

Function handles can be passed to other functions as inputs. For instance, if we want to find the minimum point of $f(x)$ in the interval $(-10, 10)$, we can use the build-in function `fminbnd` (see Section A.6 for a more detailed discussion on optimization routines):

```
[xmin fmin] = fminbnd(f,-10, 10)
```

Note that `fminbnd` takes three inputs (a function handle, and the two end-points of the interval) and returns two outputs (the minimizer and the minimal value). In our example, the minimizer of $f(x)$ in $(-10, 10)$ is -2.5, and the corresponding functional value is -16.25. To create a function that takes more than one input is just as easy. For example,

```
g = @(x,y) x.^2 + y.^3 + x.*y
```

defines the two-variable function $g(x, y) = x^2 + y^3 + xy$.

For more complex functions that involve multiple lines and intermediate variables, we need the command `function`. For example, the following code takes a column vector of data and computes its mean and standard deviation:

```
function [meanx, stdevx] = stat(x)
n = length(x);
meanx = sum(x)/n;
stdevx = sqrt(sum(x.^2)/n - meanx.^2);
```

It is important to note that all the code must be written and saved in a separate m-file. Also, the name of the file should coincide with the name of the function; in this case the file must be called `stat.m`. After saving the file, it can be used the same way as other built-in functions, for example:

```
[meanx stdx] = stat(randn(100,1))
```

returns

```
meanx =

    0.0530


stdx =

    0.9902
```

A.5 Graphics

MATLAB has several high-level graphical routines and very extensive plotting capabilities. It allows users to create various graphical objects including two- and three-dimensional graphs. One can also have a title on a graph, add a legend, change the font and font size, label the axis, etc. For more information, in the command window click on `Help` and next select `Demos`. Then choose `Graphics` followed by `2D Plots`.

In MATLAB the most basic function used to create 2D graphs is `plot`. For example, to make a graph of $y = \sin(x)$ on the interval from $x = 0$ to $x = 2\pi$, we use the following code:

```
x = 0:.01:2*pi;
y = sin(x);
plot(x,y);
```

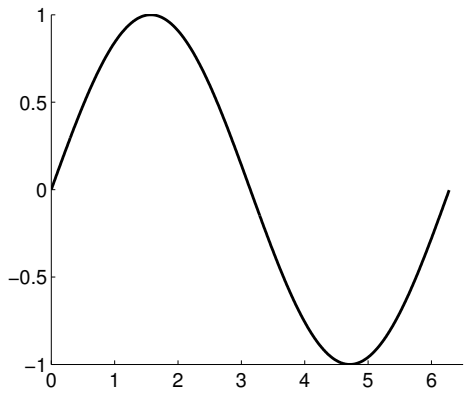

Fig. A.1 A plot of the graph $y = \sin(x)$ from 0 to 2π.

The graph produced is given in Figure A.1. Note that the command `x = 0:.01:2*pi;` creates a vector whose components range from 0 to 2π in steps of 0.01. Another useful command to create a grid is `linspace` (use `help linspace` to learn more about this function).

Another useful function is `hist`, which allows us to plot histograms. For example,

```
hist(randn(1000,1),50);
```

creates a histogram where the 1000 standard normal draws are put into 50 equally spaced bins (see Figure A.2).

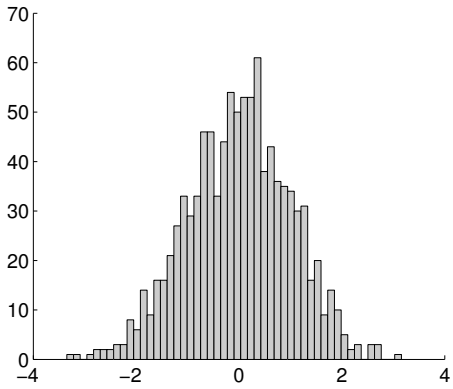

Fig. A.2 A histogram of 1000 standard normal draws.

Instead of a histogram, it is often more useful to have a density estimate. One fast and reliable Gaussian kernel density estimator is the **theta KDE** of [Botev et al., 2010]. The MATLAB function `kde.m` can be downloaded from `http://www.mathworks.com/matlabcentral/fileexchange/1b4034-kernel-density-estimator`.
☞ 195 See also Example 7.4 for an illustration.

It is often desirable to plot several graphs in the same figure window. For this purpose we need the function `subplot`. The function `subplot(i,j,k)` takes three arguments: the first two tell MATLAB that an $i \times j$ array of plots will be created, and the third is the running index that indicates the k-th subplot is currently generated. Suppose we wish to plot the functions $y = \sin(x^2/2)$ and $y = \sin(2x)$ in the same figure window. A little modification of the above code accomplishes this goal:

```
x = 0:.01:2*pi;
y1 = sin(x.^2/2);    y2 = sin(2*x);
subplot(1,2,1); plot(x,y1);
subplot(1,2,2); plot(x,y2);
```

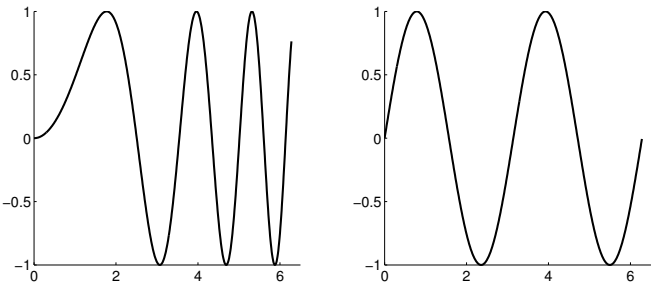

Fig. A.3 Plots of the graphs $y = \sin(x^2/2)$ and $y = \sin(2x)$ from 0 to 2π.

In addition, one can also easily produce 3D graphical objects in MATLAB. To illustrate various useful routines, suppose we want to plot the density function of the bivariate normal distribution (see Section 3.6) given by ☞ 79

$$f(x, y; \varrho) = \frac{1}{2\pi\sqrt{1-\varrho^2}}\, \mathrm{e}^{-\frac{1}{2(1-\varrho^2)}(x^2 - 2\varrho xy + y^2)} .$$

As in plotting a 2D graph, we first need to build a grid, and this can be done with the function `meshgrid`. After computing the values of the function at each point on the grid, we can plot the 3D graph using `mesh`. For example, we use the following code

```
rho = .6;
[x y] = meshgrid(-2:.1:2, -2:.1:2); %% build a 2D grid
z = 1/(2*pi*sqrt(1-rho^2)) ...
    * exp(-(x.^2 -2*rho*x.*y + y.^2)/(2*(1-rho^2)));
mesh(x,y,z);
```

to plot the the bivariate normal density function with $\varrho = 0.6$ in Figure A.4. The ellipsis (...) is used to break up a long line into multiple lines.

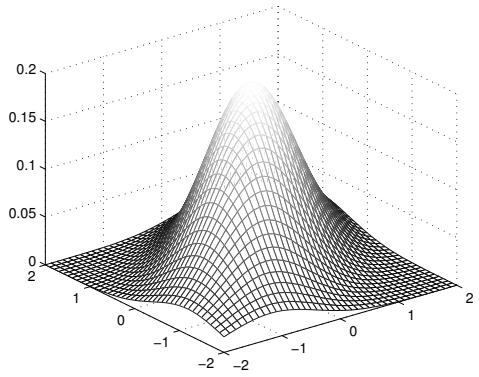

Fig. A.4 The density function of the bivariate normal distribution with $\varrho = 0.6$.

We can produce a contour plot by using the function `contour`:

```
contour(x,y,z);
```

The result is shown in Figure A.5.

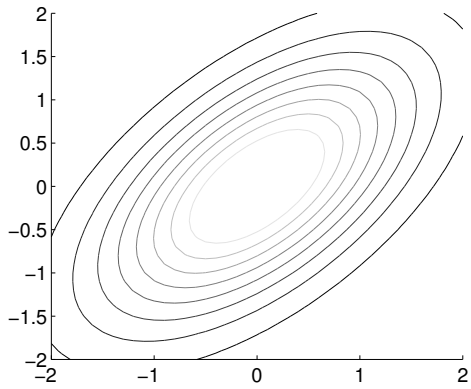

Fig. A.5 A contour plot of the bivariate normal density function with $\varrho = 0.6$.

A.6 Optimization Routines

MATLAB provides various built-in optimization routines. In this section we discuss some of them that are used in the main text. Note that all the optimization routines in MATLAB are framed in terms of minimization. In order to perform maximization, some minor changes to the objective function are required. More precisely, suppose we want to maximize the function $f(\mathbf{x})$ and find a maximizer $\mathbf{x}_{\max} = \operatorname{argmax}_{\mathbf{x}} f(\mathbf{x})$. Instead of the original maximization problem, consider minimizing $-f(\mathbf{x})$ and noting that

$$\mathbf{x}_{\max} = \underset{\mathbf{x}}{\operatorname{argmax}}\, f(\mathbf{x}) = \underset{\mathbf{x}}{\operatorname{argmin}} -f(\mathbf{x}).$$

Hence, without loss of generality, we will focus on minimization routines. One basic minimization function is `fminbnd`, which finds the minimum of a single-variable function on a fixed interval. To illustrate its usage, suppose we wish to minimize the function $f(x) = \sin(x^2)$ over the interval $[0, 3]$ (see Figure A.6).

After defining the function $f(x) = \sin(x^2)$ using the command

```
f = @(x) sin(x.^2);
```

we pass `f` to `fminbnd`, which takes three inputs (the function handle, lower and upper bounds of the interval) and gives two outputs (the minimizer and value of the function evaluated at the minimizer):

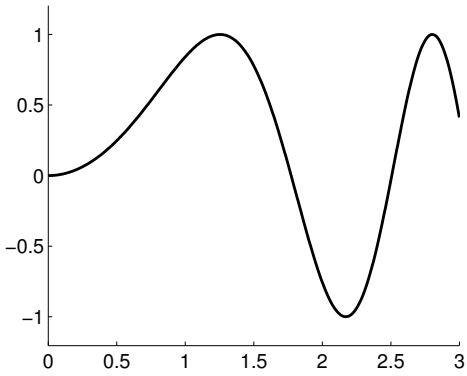

Fig. A.6 A plot of $f(x) = \sin(x^2)$ from 0 to 3.

```
[xmin fmin] = fminbnd(f,0,3);
```

For this example, we have

```
[xmin fmin]

ans =

    2.1708   -1.0000
```

The function `fminbnd` can only be used to minimize univariate functions on a closed interval. For multivariate minimization, one very useful function is `fminsearch` that finds the unconstrained minimum of a function of several variables. `fminsearch` takes two inputs, namely, the function handle and a starting value. Like `fminbnd`, `fminsearch` gives two outputs: the minimizer and the minimum (the value of the function evaluated at the minimizer). As an example, suppose we wish to maximize the bivariate normal pdf

$$f(x_1, x_2; \varrho) = \frac{1}{2\pi\sqrt{1-\varrho^2}}\, \mathrm{e}^{-\frac{1}{2(1-\varrho^2)}(x_1^2 - 2\varrho x_1 x_2 + y_2^2)}$$

with respect to $\mathbf{x} = (x_1, x_2)$ with $\varrho = 0.6$. To this end, first define $g(x_1, x_2) = -f(x_1, x_2; \varrho)$:

```
rho = .6;
g = @(x) -1/(2*pi*sqrt(1-rho^2)) ...
    *exp(-(x(1).^2 -2*rho*x(1).*x(2) +x(2).^2)/(2*(1-rho^2)));
```

Note that the variable `x` is a 1×2 vector. Then, we pass `g` to `fminsearch` with starting values, say, $[1, -1]$:

```
[xmin gmin] = fminsearch(g, [1 -1]);
```

For this example, we have

```
[xmin gmin]

ans =

    0.0000    0.0000   -0.1989
```

That is, the mode of $f(x_1, x_2; \varrho = 0.6)$ is $\mathbf{x} = (0, 0)$, and $f(0, 0) = 0.1989$.

A.7 Handling Sparse Matrices

A **sparse matrix** is simply a matrix that contains a large proportion of zeros. Computation for sparse matrices can typically be done much faster than for full matrices. In addition, as most of the elements in a sparse matrix are zeros, the storage cost of a sparse matrix is also small. In statistics we often need to deal with large sparse matrices. Thus it is useful to learn how to handle them in MATLAB.

A basic function for creating sparse matrices is `sparse`. For example, suppose the matrix

$$W = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 3 & 1 \end{pmatrix}$$

is stored as a full matrix in MATLAB. The command `sparse(W)` converts W to sparse form by squeezing out any zero elements, and returns:

```
ans =

   (1,1)        1
   (2,2)        1
   (3,3)        2
   (4,4)        3
   (4,5)        1
```

Notice that only the non-zero elements in W are stored. In general, we can create a matrix S by the command `S = sparse(i,j,s,m,n)`, which uses vectors $\mathbf{i}$, $\mathbf{j}$, and $\mathbf{s}$ to generate an $m \times n$ sparse matrix such that $S(\mathbf{i}(k), \mathbf{j}(k)) = \mathbf{s}(k)$. For example, to create the matrix W above, we first need to build a vector $\mathbf{s}$ that stores all the non-zero elements:

```
s = [1 1 2 3 1]';
```

Next, we create a vector $\mathbf{i}$ that stores the row position for each element in $\mathbf{s}$. For example, the first element in $\mathbf{s}$ should be in the first row, the second element in second row, and so on. We then do the same thing for the column positions and store them in the vector $\mathbf{j}$:

```
i = [1 2 3 4 4]';
j = [1 2 3 4 5]';
```

Finally,

```
W = sparse(i,j,s,4,5);
```

creates the 4×5 matrix W above.

There are several useful built-in functions for creating special sparse matrices. For example,

```
I = speye(100);
```

creates the 100×100 sparse identity matrix. Of course we can accomplish the same goal by using

```
I = sparse(1:100,1:100,ones(1,100));
```

though the latter is more clumsy. Another useful function is `spdiags`, the sparse version of `diag`, which can be used to extract and create sparse diagonal matrices. Use `help spdiags` to learn more about this function.

As mentioned earlier, one main advantage of working with sparse rather than full matrices is that computations involving sparse matrices are usually much quicker. For instance, it takes about 2.7 seconds to obtain the Cholesky decomposition of the full 5000×5000 identity matrix:

```
tic; chol(eye(5000)); toc;
Elapsed time is 2.728245 seconds.
```

whereas the same operation takes only 0.15 second for a sparse 5000×5000 identity matrix:

```
tic; chol(speye(5000)); toc;
Elapsed time is 0.014867 seconds.
```

A.8 Gamma and Dirichlet Generator

The following MATLAB program `gamrand` implements the method developed in [Marsaglia & Tsang, 2000] to generate samples from a $\mathsf{Gamma}(\alpha, \lambda)$ distribution. If the *Statistics Toolbox* is available, the function `gamrnd` can be used instead; but note that `gamrnd(a,b)` generates random variables from a $\mathsf{Gamma}(a, 1/b)$ distribution.

```
function x=gamrand(alpha,lambda)
if alpha>1
    d=alpha-1/3; c=1/sqrt(9*d); flag=1;
    while flag
        Z=randn;
        if Z>-1/c
            V=(1+c*Z)^3; U=rand;
            flag=log(U)>(0.5*Z^2+d-d*V+d*log(V));
        end
    end
    x=d*V/lambda;
else
    x=gamrand(alpha+1,lambda);
    x=x*rand^(1/alpha);
end
```

☞ 233 As a direct consequence of Theorem 8.2, the following MATLAB program `dirichrnd` generates samples from a $\mathsf{Dirichlet}(\boldsymbol{\alpha})$ distribution. Draws from a $\mathsf{Beta}(\alpha, \beta)$ are obtained by taking $\boldsymbol{\alpha} = (\alpha, \beta)$.

```
function x=dirichrnd(alpha)
n=length(alpha)-1;
Y=nan(1,n+1);
for k=1:n+1
    Y(k)=gamrand(alpha(k),1);
```

```
end
x=Y(1:n)/sum(Y);
```

A.9 Cdfs and Inverse Cdfs

The following MATLAB program `cumdf` evaluates the cdfs of normal, Student's t, gamma, chi-squared, and F distributions. If the *Statistics Toolbox* is available, the function `cdf` can be used instead.

```
function y = cumdf(dist,x,varargin)
switch dist
    case 'norm'
        mu = varargin{1}; sigma = varargin{2};
        y = (erf(( (x - mu)/sigma )/sqrt(2)) + 1)/2;
    case 't'
        nu = varargin{1};
        y = 1-0.5*betainc(nu/(nu+x.^2),nu/2,1/2);
    case 'gamma'
        alpha = varargin{1}; lambda = varargin{2};
            % different from Stats toolbox
        y = gammainc(lambda*x,alpha);
    case 'chi2'
        n = varargin{1};
        y = gammainc(x/2,n/2);
    case 'F'
        m = varargin{1}; n = varargin{2};
        y = 1 - betainc(n/(n+m*x),n/2,m/2);
end
```

The following MATLAB program `icumdf` evaluates the inverse cdfs of normal, Student's t, gamma, chi-squared, and F distributions. The corresponding built-in function in the *Statistics Toolbox* is `icdf`.

```
function x = icumdf(dist,y,varargin)
switch dist
    case 'norm'
        mu = varargin{1}; sigma = varargin{2};
        x = mu  + sigma*sqrt(2)*erfinv(2*y -1);
    case 't'
        nu = varargin{1};
        x = sqrt(nu/betaincinv(2*(1-y),nu/2,1/2) - nu);
```

```
    case 'gamma'
        alpha = varargin{1}; lambda = varargin{2};
            % different from Stats toolbox
        x = gammaincinv(y,alpha)/lambda;
    case 'chi2'
        n = varargin{1};
        x = gammaincinv(y,n/2)*2;
    case 'F'
        m = varargin{1}; n = varargin{2};
        x = n/m/betaincinv(1-y,n/2,m/2) - n/m;
end
```

A.10 Further Reading and References

The official MATLAB documentation site is

`http://www.mathworks.com/help/techdoc/`

A good place to learn more about the major functionality in MATLAB is the MATLAB *Getting Started Guide* available at

`http://www.mathworks.com/help/pdf_doc/matlab/getstart.pdf`

MATLAB programs for generating random variables from a wide range of distributions may be found on the homepage of the *Handbook of Monte Carlo Methods* [Kroese et al., 2011]:

`http://www.montecarlohandbook.org`

Finally, all programs and (large) data files in this book may be downloaded from the homepage

`http://www.statmodcomp.org`

To accommodate the users of the statistical programming language R, we have mirrored each MATLAB program with its equivalent in R.