Crash course in MATLAB

June 23, 2006

The purpose of this document is to give a medium-length introduction to the essentials of MATLAB and how to use it well. I'm aiming for a document that's somewhere between a two-page introduction and the excellent all-inclusive books available. I use it in a week-long "camp" for graduate students at the University of Delaware in the summer after their first year of study.

No advance knowledge of MATLAB is necessary to read this document, but a working familiarity with basic linear algebra is expected, and general knowledge of programming is a big help. The first four sections cover the basics needed to solve even simple exercises. Section 5 is a bare-bones introduction to the big subject of graphics in MATLAB. The next three sections discuss matters than an intermediate MATLAB programmer ought to know. The remaining sections go into advanced issues in varying degrees of detail.

The version of MATLAB at this writing is 7.1.

Contents

*Department of Mathematical Sciences, Ewing Hall, University of Delaware, Newark, DE 19716; driscoll@math.udel.edu.

1 Introduction

MATLAB is a software package for computation in engineering, science, and applied mathematics. It offers a powerful programming language, excellent graphics, and a wide range of expert knowledge. MATLAB is published by and a trademark of The MathWorks, Inc.

The focus in MATLAB is on computation, not mathematics: Symbolic expressions and manipulations are not possible (except through the optional Symbolic Toolbox, a clever interface to Maple). All results are not only numerical but inexact, thanks to the rounding errors inherent in computer arithmetic. The limitation to numerical computation can be seen as a drawback, but it is a source of strength too: MATLAB is much preferred to Maple, Mathematica, and the like when it comes to numerics.

On the other hand, compared to other numerically oriented languages like C++ and FORTRAN, MATLAB is much easier to use and comes with a huge standard library.[1] The unfavorable comparison here is a gap in execution speed. This gap is not always as dramatic as popular lore has it, and it can often be narrowed or closed with good MATLAB programming (see section 6). Moreover, one can link other codes into MATLAB, or vice versa, and MATLAB now optionally supports parallel computing. Still, MATLAB is usually not the tool of choice for maximum-performance computing.

The MATLAB niche is numerical computation on workstations for non-experts in computation. This is a huge niche—one way to tell is to look at the number of MATLAB-related books on `mathworks.com`. Even for supercomputer users, MATLAB can be a valuable environment in which to explore and fine-tune algorithms before more laborious coding in another language.

Most successful computing languages and environments acquire a distinctive character or culture. In MATLAB, that culture contains several elements: an experimental and graphical bias, resulting from the interactive environment and compression of the write-compile-link-execute-analyze cycle; an emphasis on syntax that is compact and friendly to the interactive mode, rather than tightly constrained and verbose; a kitchen-sink mentality for providing functionality; and a high degree of openness and transparency (though not to the extent of being open source software).

1.1 The fifty-cent tour

When you start MATLAB, you get a multipaneled **desktop**. The layout and behavior of the desktop and its components are highly customizable (and may in fact already be customized for your site). The component that is the heart of MATLAB is called the **Command Window**, located on the

[1] Here and elsewhere I am thinking of the "old FORTRAN," FORTRAN 77. This is not a commentary on the usefulness of FORTRAN 90 but on my ignorance of it.

right by default. Here you can give MATLAB commands typed at the prompt, >>. Unlike FORTRAN and other compiled computer languages, MATLAB is an **interpreted** environment—you give a command, and MATLAB tries to execute it right away before asking for another.

At the top left you can see the **Current Directory**. In general MATLAB is aware only of files in the current directory (folder) and on its **path**, which can be customized. Commands for working with the directory and path include `cd`, `what`, `addpath`, and `editpath` (or you can choose "File/Set path..." from the menus). You can add files to a directory on the path and thereby add commands to MATLAB; we will return to this subject in section 3.

Next to the Current Directory tab is the **Workspace** tab. The workspace shows you what variable names are currently defined and some information about their contents. (At start-up it is, naturally, empty.) This represents another break from compiled environments: variables created in the workspace persist for you to examine and modify, even after code execution stops.

Below the Command Window/Workspace window is the **Command History** window. As you enter commands, they are recorded here. This record persists across different MATLAB sessions, and commands or blocks of commands can be copied from here or saved to files.

As you explore MATLAB, you will soon encounter some **toolboxes**. These are individually packaged sets of capabilities that provide in-depth expertise on particular subject areas. There is no need to load them explicitly—once installed, they are always available transparently. You may also encounter **Simulink**, which is a semi-independent graphical control-engineering package not covered in this document.

1.2 Graphical versus command-line usage

MATLAB was originally entirely a command-line environment, and it retains that orientation. But it is now possible to access a great deal of the functionality from graphical interfaces—menus, buttons, and so on. These interfaces are especially useful to beginners, because they lay out the available choices clearly.[2]

As a rule, graphical interfaces can be more natural for certain types of interactive work, such as annotating a graph or debugging a program, whereas typed commands remain better for complex, precise, repeated, or reproducible tasks. One does not always need to make a choice, though; for instance, it is possible to save a figure's styles as a template that can be used with different data by pointing and clicking. Moreover, you can package code you want to distribute with your own graphical interface, one that itself may be designed with a combination of graphical and command-oriented tools. In the end, an advanced MATLAB user should be able to exploit both modes of work to be productive.

That said, the focus of this document is on typed commands. In many (most?) cases these have graphical interface equivalents, even if I don't explicitly point them out.

[2] In particular, feel free to right-click (on Control-click on a Mac) on various objects to see what you might be able to do to them.

1.3 Help

MATLAB is huge. Nobody can tell you everything that you personally will need to know, nor could you remember it all anyway. It is essential that you become familiar with the online help.

There are two levels of help:

- If you need quick help on the syntax of a command, use `help`. For example, `help plot` shows right in the Commmand Window all the ways in which you can use the `plot` command. Typing `help` by itself gives you a list of categories that themselves yield lists of commands.
- Typing `doc` followed by a command name brings up more extensive help in a separate window.[3] For example, `doc plot` is better formatted and more informative than `help plot`. In the left panel one sees a hierarchical, browsable display of all the online documentation. Typing `doc` alone or selecting Help from the menu brings up the window at a "root" page.

The *Getting Started with MATLAB* manual is a very good place to get a more gentle and thorough introduction than the one to follow here. Depending on your installation, the documentation may be available in PDF form for printing and offline reading.

1.4 Basic commands and syntax

If you type in a valid expression and press Enter, MATLAB will immediately execute it and return the result, just like a calculator.

```
>> 2+2

ans =
     4

>> 4^2

ans =
    16

>> sin(pi/2)

ans =
     1

>> 1/0
Warning: Divide by zero.

ans =
   Inf
```

[3] Indeed, the help browser is a fully functional web browser as well.

```
>> exp(i*pi)

ans =
  -1.0000 + 0.0000i
```

Notice some of the special expressions here: `pi` for π, `Inf` for ∞, and `i` for $\sqrt{-1}$.[4] Another special value is `NaN`, which stands for **not a number**. `NaN` is used to express an undefined value. For example,

```
>> Inf/Inf

ans =
   NaN
```

`NaN`s can be tricky. For example, two `NaN` values are unequal by definition.

You can assign values to variables with alphanumeric names.

```
>> x = sqrt(3)

x =
    1.7321

>> days_since_birth = floor(now) - datenum(1969,05,06)

days_since_birth =
       12810

>> 3*z
??? Undefined function or variable 'z'.
```

Observe that *variables must have values before they can be used*. When an expression returns a single result that is not assigned to a variable, this result is assigned to `ans`, which can then be used like any other variable.

```
>> atan(x)

ans =
    1.0472

>> pi/ans

ans =
     3
```

[4] `j` is also understood for $\sqrt{-1}$. Both names can be reassigned, however, and it's safer to use always `1i` or `1j` to refer to the imaginary unit.

In floating-point arithmetic, you should not expect "equal" values to have a difference of exactly zero. The built-in number `eps` tells you the maximum error in arithmetic on your particular machine.[5] For simple operations, the relative error should be close to this number. For instance,

```
>> eps

ans =
   2.2204e-16

>> exp(log(10)) - 10

ans =
   1.7764e-15

>> ans/10

ans =
   1.7764e-16
```

Here are a few other demonstration statements.

```
>> % Anything after a % sign is a comment.
>> x = rand(100,100);  % ; means "don't print out result"
>> s = 'Hello world';  % single quotes enclose a string
>> t = 1 + 2 + 3 + ...
4 + 5 + 6              % ... continues a line

t =
    21
```

Once variables have been defined, they exist in the **workspace**. You can see what's in the workspace from its desktop window, or by typing

```
>> who

Your variables are:

ans  s    t    x
```

1.5 Saving work

If you enter `save myfile`, all the variables in the workspace will be saved to a file called `myfile.mat` in the current directory. You can also select the variables to be saved by typing them after the filename argument. If you later enter `load myfile`, the saved variables are returned to the workspace (overwriting any values with the same names).

[5] Like other names, `eps` can be reassigned, but doing so has no effect on the roundoff precision.

If you highlight commands in the Command History window, right-click, and select "Create M-File", you can save the typed commands to a text file. This can be very helpful for recreating what you have done. Also see section 3.1.

1.6 Exercises

1. Evaluate the following mathematical expressions in MATLAB.

   (a) $\tanh(e)$ (b) $\log_{10}(2)$
   (c) $\left|\sin^{-1}\left(-\frac{1}{2}\right)\right|$ (d) 123456 mod 789 (remainder after division)

2. What is the name of the built-in function that MATLAB uses to:

   (a) Compute a Bessel function of the second kind?
   (b) Test the primality of an integer?
   (c) Multiply two polynomials together?
   (d) Plot a vector field?
   (e) Report the current date and time?

2 Arrays and matrices

The heart and soul of MATLAB is linear algebra. In fact, MATLAB was originally a contraction of "matrix laboratory." More so than any other language, MATLAB encourages and expects you to make heavy use of arrays, vectors, and matrices.

Some jargon: An **array** is a collection of numbers, called **elements** or **entries**, referenced by one or more indices running over different index sets. In MATLAB, the index sets are always sequential integers starting with 1. The **dimension** of the array is the number of indices needed to specify an element. The **size** of an array is a list of the sizes of the index sets.

A **matrix** is a two-dimensional array with special rules for addition, multiplication, and other operations. It represents a mathematical linear transformation. The two dimensions are called the **rows** and the **columns**. A **vector** is a matrix for which one dimension has only the index 1. A **row vector** has only one row and a **column vector** has only one column.

Although an array is much more general and less mathematical than a matrix, the terms are often used interchangeably. What's more, in MATLAB there is really no formal distinction—sometimes, not even between a scalar and a 1×1 matrix. The commands below are sorted according to the array/matrix distinction, but MATLAB will usually let you mix them freely. The idea (here as elsewhere) is that MATLAB keeps the language simple and natural. It's up to you to stay out of trouble.

2.1 Building arrays

The simplest way to construct a small array is by enclosing its elements in square brackets.

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

A =
     1     2     3
     4     5     6
     7     8     9

>> b = [0;1;0]

b =
     0
     1
     0
```

Separate columns by spaces or commas, and rows by semicolons or new lines. Information about size and dimension is stored with the array.[6]

```
>> size(A)
```

[6] Because of this, array sizes are not usually passed explicitly to functions as they are in FORTRAN.

```
ans =
     3     3

>> ndims(A)

ans =
     2

>> size(b)

ans =
     3     1

>> ndims(b)

ans =
     2
```

Notice that there is really no such thing as a one-dimensional array in MATLAB. Even vectors are technically two-dimensional, with a trivial dimension.[7] Table 1 lists more commands for obtaining information about an array.

Table 1: Matrix information commands.

`size`	size in each dimension
`length`	size of longest dimension (esp. for vectors)
`ndims`	number of dimensions
`find`	indices of nonzero elements

Arrays can be built out of other arrays, as long as the sizes are compatible.

```
>> [A b]

ans =
     1     2     3     0
     4     5     6     1
     7     8     9     0

>> [A;b]
??? Error using ==> vertcat
All rows in the bracketed expression must have the same
number of columns.

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

[7] What's more, the distinction between row and column vectors is often, but not always, important.

```
B =
     1     2     5
     3     4     6
```

One special array is the **empty matrix**, which is entered as `[]`.

Bracket constructions are suitable only for very small matrices. For larger ones, there are many useful functions, some of which are shown in Table 2.

Table 2: Commands for building arrays and matrices.

`eye`	identity matrix
`zeros`	all zeros
`ones`	all ones
`diag`	diagonal matrix (or, extract a diagonal)
`toeplitz`	constant on each diagonal
`triu`	upper triangle
`tril`	lower triangle
`rand`, `randn`	random entries
`linspace`	evenly spaced entries
`cat`	concatenate along a given dimension
`repmat`	duplicate vector across a dimension

An especially important array constructor is the **colon** operator.

```
>> 1:8

ans =
     1     2     3     4     5     6     7     8

>> 0:2:10

ans =
     0     2     4     6     8    10

>> 1:-.5:-1

ans =
    1.0000    0.5000         0   -0.5000   -1.0000
```

The format is `first:step:last`. The result is always a row vector, or the empty matrix if `last` $<$ `first`.

2.2 Referencing elements

It is frequently necessary to access one or more of the elements of a matrix. Each dimension is given a single index or vector of indices. The result is a **block** extracted from the matrix. Some examples using the definitions above:

```
>> A(2,3)

ans =
     6

>> b(2)         % b is a vector

ans =
     1

>> b([1 3])     % multiple elements

ans =
     0
     0

>> A(1:2,2:3)  % a submatrix

ans =
     2     3
     5     6

>> B(1,2:end)  % special keyword

ans =
     2     5

>> B(:,3)      % "include all" syntax

ans =
     5
     6

>> b(:,[1 1 1 1])

ans =
     0     0     0     0
     1     1     1     1
     0     0     0     0
```

The colon is often a useful way to construct these indices. There are some special syntaxes: `end` means the largest index in a dimension, and `:` alone is short for `1:end`—i.e. everything in that

dimension. Note too from the last example that the result need not be a subset of the original array.

Vectors, unsurprisingly, can be given a single subscript. But in fact *any* array can be accessed via a single subscript. Multidimensional arrays are actually stored linearly in memory, varying over the first dimension, then the second, and so on. (Think of the columns of a table being stacked on top of each other.) In this sense the array is equivalent to a vector, and a single subscript will be interpreted in this context. (See `sub2ind` and `ind2sub` for more details.)

```
>> A

A =
     1     2     3
     4     5     6
     7     8     9

>> A(2)

ans =
     4

>> A(7)

ans =
     3

>> A([1 2 3 4])

ans =
     1     4     7     2

>> A([1;2;3;4])

ans =
     1
     4
     7
     2

>> A(:)

ans =
     1
     4
     7
     2
     5
     8
```

```
     3
     6
     9
```

The output of this type of index is in the same shape as the index. The potentially ambiguous `A(:)` is always a column vector.

Subscript referencing can be used on either side of assignments.

```
>> B(1,:) = A(1,:)

B =
     1     2     3
     3     4     6

>> C = rand(2,5)

C =
    0.8125    0.4054    0.4909    0.5909    0.5943
    0.2176    0.5699    0.1294    0.8985    0.3020

>> C(:,4) = []   % delete elements

C =
    0.8125    0.4054    0.4909    0.5943
    0.2176    0.5699    0.1294    0.3020

>> C(2,:) = 0    % expand the scalar into the submatrix

C =
    0.8125    0.4054    0.4909    0.5943
         0         0         0         0

>> C(3,1) = 3    % create a new row to make space

C =
    0.8125    0.4054    0.4909    0.5943
         0         0         0         0
    3.0000         0         0         0
```

An array is resized automatically if you delete elements or make assignments outside the current size. (Any new undefined elements are made zero.) This can be highly convenient, but it can also cause hard-to-find mistakes.

A different kind of indexing is **logical indexing**. Logical indices usually arise from a **relational operator** (see Table 3). The result of applying a relational operator is a **logical array**, whose elements are 0 and 1 with interpretation as "false" and "true."[8] Using a logical array as an index

[8]The commands `false` and `true` can be used to create logical arrays.

returns those values where the index is 1 (in the single-index sense above).

```
>> B>3

ans =
     0     0     0
     0     1     1

>> B(ans)

ans =
     4
     6

>> b(b==0)

ans =
     0
     0

>> b([1 1 1])   % first element, three copies

ans =
     0
     0
     0

>> b(logical([1 1 1]))   % every element

ans =
     0
     1
     0
```

Table 3: Relational operators.

==	equal to	~=	not equal to
<	less than	>	greater than
<=	less than or equal to	>=	greater than or equal to

2.3 Matrix operations

The arithmetic operators +,−,*,^ are interpreted in a matrix sense. When appropriate, scalars are "expanded" to match a matrix.[9]

```
>> A+A

ans =
     2     4     6
     8    10    12
    14    16    18

>> ans-1

ans =
     1     3     5
     7     9    11
    13    15    17

>> 3*B

ans =
     3     6     9
     9    12    18

>> A*b

ans =
     2
     5
     8

>> B*A

ans =
    30    36    42
    61    74    87

>> A*B
??? Error using ==> *
Inner matrix dimensions must agree.

>> A^2

ans =
```

[9] One consequence is that adding a scalar to a square matrix is not the same as adding a scalar times the identity matrix.

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

The apostrophe ' produces the complex-conjugate transpose of a matrix. This corresponds to the mathematical adjoint of the linear operator represented by the matrix.

```
>> A*B'-(B*A')'

ans =
     0     0
     0     0
     0     0

>> b'*b

ans =
     1

>> b*b'

ans =
     0     0     0
     0     1     0
     0     0     0
```

A special operator, \ (backslash), is used to solve linear systems of equations.

```
>> C = [1 3 -1; 2 4 0; 6 0 1];
>> x = C\b

x =
   -0.1364
    0.3182
    0.8182

>> C*x - b

ans =
   1.0e-16 *

    0.5551
         0
         0
```

Several functions from linear algebra are listed in Table 4; there are many, many others.

Table 4: Functions from linear algebra.

`\`	solve linear system (or least squares)
`rank`	rank
`det`	determinant
`norm`	norm (2-norm, by default)
`expm`	matrix exponential
`lu`	LU factorization (Gaussian elimination)
`qr`	QR factorization
`chol`	Cholesky factorization
`eig`	eigenvalue decomposition
`svd`	singular value decomposition

2.4 Array operations

Array operations simply act identically on each element of an array. We have already seen some array operations, namely `+` and `-`. But the operators `*`, `'`, `^`, and `/` have particular matrix interpretations. To get elementwise behavior appropriate for an array, precede the operator with a dot.

```
>> A

A =
     1     2     3
     4     5     6
     7     8     9

>> C

C =
     1     3    -1
     2     4     0
     6     0     1

>> A.*C

ans =
     1     6    -3
     8    20     0
    42     0     9

>> A*C

ans =
    23    11     2
```

```
    50    32     2
    77    53     2

>> A./A

ans =
     1     1     1
     1     1     1
     1     1     1

>> (B+i)'
ans =
  -1.0000 - 1.0000i   3.0000 - 1.0000i
  -2.0000 - 1.0000i   4.0000 - 1.0000i
  -3.0000 - 1.0000i   6.0000 - 1.0000i

>> (B+i).'
ans =
  -1.0000 + 1.0000i   3.0000 + 1.0000i
  -2.0000 + 1.0000i   4.0000 + 1.0000i
  -3.0000 + 1.0000i   6.0000 + 1.0000i
```

There is no difference between `'` and `.'` for real-valued arrays.

Most elementary functions, such as `sin`, `exp`, etc., act elementwise.

```
>> B

B =
     1     2     3
     3     4     6

>> cos(pi*B)
ans =
    -1     1    -1
    -1     1     1

>> exp(A)
ans =
   1.0e+03 *
    0.0027    0.0074    0.0201
    0.0546    0.1484    0.4034
    1.0966    2.9810    8.1031

>> expm(A)
ans =
   1.0e+06 *
    1.1189    1.3748    1.6307
    2.5339    3.1134    3.6929
```

```
    3.9489    4.8520    5.7552
```

It's easy to forget that `exp(A)` is an *array* function. Use `expm(A)` to get the matrix exponential $I + A + A^2/2 + A^3/6 + \cdots$.

Elementwise operators are often useful in functional expressions. Consider evaluating a Taylor approximation to $\sin(t)$ at several values of t:

```
>> t = (0:0.25:1)*pi/2

t =
         0    0.3927    0.7854    1.1781    1.5708

>> t - t.^3/6 + t.^5/120

ans =
         0    0.3827    0.7071    0.9245    1.0045
```

This is easier and better than writing a loop for the calculation. (See section 6.3.) In this particular case, `polyval([1/120,0,-1/6,0,1,0],t)` is arguably even better.

Another kind of array operation works in parallel along one dimension of the array, returning a result that is one dimension smaller.

```
>> C

C =
     1     3    -1
     2     4     0
     6     0     1

>> sum(C,1)

ans =
     9     7     0

>> sum(C,2)

ans =
     3
     6
     7
```

Other functions that behave this way are shown in Table 5.

Table 5: Reducing functions.

max	sum	mean	any
min	diff	median	all
sort	prod	std	cumsum

2.5 Sparse matrices

It's natural to think of a matrix as a complete rectangular table of numbers. However, many real-world matrices are both extremely large and very **sparse**, meaning that most entries are zero.[10] In such cases it's wasteful or downright impossible to store every entry. Instead one can keep a list of nonzero entries and their locations. MATLAB has a `sparse` data type for this purpose. The `sparse` and `full` commands convert back and forth and lay bare the storage difference.

```
>> A = vander(1:3);
>> sparse(A)

ans =
   (1,1)         1
   (2,1)         4
   (3,1)         9
   (1,2)         1
   (2,2)         2
   (3,2)         3
   (1,3)         1
   (2,3)         1
   (3,3)         1

>> full(ans)
ans =

     1     1     1
     4     2     1
     9     3     1
```

Sparsifying a standard full matrix is usually *not* the right way to create a sparse matrix—you should avoid creating very large full matrices, even temporarily. One alternative is to give `sparse` the raw data required by the format.

```
>> sparse(1:4,8:-2:2,[2 3 5 7])
```

[10] This is often the result of a "nearest neighbor interaction" that the matrix is modeling. For instance, the PageRank algorithm used by Google starts with an adjacency matrix in which a_{ij} is nonzero if page j links to page i. Obviously any page links to a tiny fraction of the more than 3 billion and counting public pages!

```
ans =
    (4,2)        7
    (3,4)        5
    (2,6)        3
    (1,8)        2
```

(This is the functional inverse of the `find` command.) Alternatively, you can create an empty sparse matrix with space to hold a specified number of nonzeros, and then fill in using standard subscript assignments. Another useful sparse building command is `spdiags`, which builds along the diagonals of the matrix.

```
>> M = ones(6,1)*[-20 Inf 10]

M =
   -20   Inf    10
   -20   Inf    10
   -20   Inf    10
   -20   Inf    10
   -20   Inf    10
   -20   Inf    10

>> full( spdiags( M,[-2 0 1],6,6 ) )

ans =
   Inf    10     0     0     0     0
     0   Inf    10     0     0     0
   -20     0   Inf    10     0     0
     0   -20     0   Inf    10     0
     0     0   -20     0   Inf    10
     0     0     0   -20     0   Inf
```

The `nnz` command tells how many nonzeros are in a given sparse matrix. Since it's impractical to view directly all the entries (even just the nonzeros) of a realistically sized sparse matrix, the `spy` command helps by producing a plot in which the locations of nonzeros are shown. For instance, `spy(bucky)` shows the pattern of bonds among the 60 carbon atoms in a buckyball.

MATLAB has a lot of ability to work intelligently with sparse matrices. The arithmetic operators `+`, `-`, `*`, and `^` use sparse-aware algorithms and produce sparse results when applied to sparse inputs. The backslash `\` uses sparse-appropriate linear system algorithms automatically as well. There are also functions for the iterative solution of linear equations, eigenvalues, and singular values.

2.6 Exercises

1. (a) Check the help for `diag` and use it (maybe more than once) to build the 16×16 matrix

$$D = \begin{bmatrix} -2 & 1 & 0 & 0 & \cdots & 0 & 1 \\ 1 & -2 & 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & 1 & -2 & 1 & 0 \\ 0 & 0 & \cdots & 0 & 1 & -2 & 1 \\ 1 & 0 & 0 & \cdots & 0 & 1 & -2 \end{bmatrix}$$

   (b) Now read about `toeplitz` and use it to build D.

   (c) Use `toeplitz` and whatever else you need to build

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 1 & 2 & 3 \\ 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \\ \frac{1}{2} & 1 & \frac{1}{2} & \frac{1}{3} \\ \frac{1}{3} & \frac{1}{2} & 1 & \frac{1}{2} \\ \frac{1}{4} & \frac{1}{3} & \frac{1}{2} & 1 \end{bmatrix} \qquad \begin{bmatrix} 4 & 3 & 2 & 1 \\ 3 & 2 & 1 & 2 \\ 2 & 1 & 2 & 3 \\ 1 & 2 & 3 & 4 \end{bmatrix}$$

   Do not just enter the elements directly—your solutions should be just as easy to use if the matrices were 100×100.

2. Find a MATLAB one-line expression to create the $n \times n$ matrix A satisfying

$$a_{ij} = \begin{cases} 1, & \text{if } i - j \text{ is prime} \\ 0, & \text{otherwise} \end{cases}$$

3. Let A be a random 8×8 matrix. Find the maximum values (a) in each column, (b) in each row, and (c) overall. Also (d) find the row and column indices of all elements that are larger than 0.25.

4. Suppose A is a matrix whose entries are all positive numbers. Write one line that will multiply each column of A by a scalar so that in the resulting matrix, every column sums to 1. (To make it more difficult, suppose that zero entries are allowed and leave a column that sums to zero unchanged.)

5. A *magic square* is an $n \times n$ matrix in which each integer 1, 2, ..., n^2 appears once and for which all the row, column, and diagonal sums are identical. MATLAB has a command `magic` that returns magic squares. Check its output at a few sizes and use MATLAB to verify the summation property. (The "antidiagonal" sum will be the trickiest.)

6. Suppose we represent a standard deck of playing cards by a vector `v` containing one copy of each integer from 1 to 52. Show how to "shuffle" `v` by rearranging its contents in a random order. (Note: There is one very easy answer to this problem—if you look hard enough.)

7. Examine the eigenvalues of the family of matrices

$$D_N = -N^2 \begin{bmatrix} -2 & 1 & 0 & 0 & \cdots & 1 \\ 1 & -2 & 1 & 0 & \cdots & 0 \\ 0 & 1 & -2 & 1 & \cdots & 0 \\ & & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 & -2 & 1 \\ 1 & 0 & 0 & \cdots & 1 & -2 \end{bmatrix}$$

where D_N is $N \times N$, for several growing values of N; for example, $N = 4, 8, 16, 32$. (This is one approximate representation of the second-derivative operator for periodic functions. The smallest eigenvalues are near integer multiples of a simple number.)

8. Use several random instances of an $m \times n$ matrix A to convince yourself that

$$\|A\|_F^2 = \sum_{i=1}^{K} \sigma_i^2,$$

where $K = \min\{m, n\}$, $\{\sigma_1, \ldots, \sigma_K\}$ are the singular values of A, and $\|\cdot\|_F$ is the Frobenius norm (root-mean-square of the elements of A).

9. Let `B=bucky` and make a series of `spy` plots of B^2, B^3, etc. to see the phenomenon of *fill-in*: many operations, including multiplication, increase the density of nonzeros. (Can you see why the (i, j) entry of B^n is the number of paths of length n between nodes i and j?) What fill-in do you see with `inv(B)`?

3 Scripts and functions

An **M-file** is a plain text file containing MATLAB commands and saved with the filename extension `.m`. There are two types, **scripts** and **functions**. MATLAB comes with a pretty good editor that is tightly integrated into the environment; start it by typing `edit` with a file name. However, you are free to use any text editor.

An M-file should be saved in the **path** in order to be executed. The path is just a list of directories (folders) in which MATLAB will look for files. Use `editpath` or menus to see and change the path.

There is no need to compile either type of M-file. Simply type in the name of the file (without the extension) in order to run it. Changes that are saved to disk will be included in the next call to the function or script. (You can alter this behavior with `mlock`.)

One important type of statement in an M-file is a **comment**, which is indicated by a percent sign `%`. Any text on the same line after a percent sign is ignored (unless `%` appears as part of a string in quotes). Furthermore, the first contiguous block of comments in an M-file serves as documentation for the file and will be typed out in the command window if `help` is used on the file. For instance, say the following is saved as `myscript.m` on the path:

```
% This script solves the nasty homework problem assigned by
% Professor Driscoll.

x = rand(1);  % He'll never notice.
```

Then at the prompt one would find

```
>> help myscript

  This script solves the nasty homework problem assigned by
  Professor Driscoll.
```

3.1 Using scripts effectively

The contents of a script file are literally interpreted as though they were typed at the prompt. In fact, some people prefer to use MATLAB exclusively by typing all commands into scripts and running them. Good reasons to use scripts are

- Creating or revising a sequence of more than four or five lines.
- Reproducing or rereading your work at a later time.
- Running a CPU-intensive job in the background, allowing you to log off.

The last point here refers specifically to UNIX. For example, suppose you wrote a script called `run.m` that said:

```
result = execute_big_routine(1);
result = another_big_routine(result);
result = an_even_bigger_routine(result);
save rundata result
```

At the UNIX prompt in the directory of `run.m`, you would enter (using `csh` style)

```
nice +19 matlab < run.m >! run.log &
```

which would cause your script to run in the background with low priority. The job will continue to run until finished, even if you log off. The output that would have been typed to the screen is redirected to `run.log`. You will usually need at least one `save` command to save your results. Use it often in the script in case of a crash or other interruption. Also take pains to avoid taking huge chunks of memory or disk space when running in an unattended mode. It's advisable to start the process and monitor it using `top` in the UNIX shell for a little while to make sure it does not gobble up resources unexpectedly.

3.2 Functions

Functions are the main way to extend the capabilities of MATLAB. Compared to scripts, they are much better at compartmentalizing tasks. Each function starts with a line such as

```
function [out1,out2] = myfun(in1,in2,in3)
```

The variables `in1`, etc. are **input arguments**, and `out1` etc. are **output arguments**. You can have as many as you like of each type (including zero) and call them whatever you want. The name `myfun` should match the name of the disk file.

Here is a function that implements (badly, it turns out) the quadratic formula.

```
function [x1,x2] = quadform(a,b,c)

d = sqrt(b^2 - 4*a*c);
x1 = (-b + d) / (2*a);
x2 = (-b - d) / (2*a);
```

From MATLAB you could call

```
>> [r1,r2] = quadform(1,-2,1)

r1 =
     1

r2 =
     1
```

One of the most important features of a function is its **local workspace**. Any arguments or other variables created while the function executes are available only within the function. Conversely, the variables available to the command line (the so-called **base workspace**) are normally not visible to the function. If during the function execution, other functions are called, each of those calls also sets up a private workspace. These restrictions are called **scoping**, and they make it possible to write complex programs without worrying about name clashes. The values of the input arguments are copies of the original data, so any changes you make to them will not change anything outside the function's scope.[11] In general, the only communication between a function and its caller is through the input and output arguments (though see section 6.5 for exceptions). You can always see the variables defined in the current workspace by typing `who` or `whos`.

Another important aspect of function M-files is that most of the functions built into MATLAB (except core math functions) are themselves M-files that you can read and copy. This is an excellent way to learn good programming practice—and dirty tricks.

3.3 Conditionals: if and switch

Often a function needs to branch based on runtime conditions. MATLAB offers structures for this similar to those in most languages.

Here is an example illustrating most of the features of `if`.

```
if isinf(x) | ~isreal(x)
  disp('Bad input!')
  y = NaN;
elseif (x == round(x)) && (x > 0)
  y = prod(1:x-1);
else
  y = gamma(x);
end
```

The conditions for `if` statements may involve the relational operators of Table 3, or functions such as `isinf` that return logical values. Numerical values can also be used, with nonzero meaning true, but "`if x~=0`" is better practice than "`if x`". Note also that if the `if` clause is a non-scalar array, it is taken as true only when all the elements are true/nonzero. To avoid confusion, it's best to use `any` or `all` to reduce array logic to scalar values.

Individual conditions can be combined using

`&` (logical AND) `|` (logical OR) `~` (logical NOT)

Compound logic in `if` statements can be **short-circuited**. As a condition is evaluated from left to right, it may become obvious before the end that truth or falsity is assured. At that point, evaluation of the condition is halted. This makes it convenient to write things like

[11] MATLAB does avoid copying a function argument (i.e., it "passes by reference" as in FORTRAN) if the function never alters the data.

```
if (length(x) > 2) & (x(3)==1) ...
```

that otherwise could create errors or be awkward to write.[12]

The `if`/`elseif` construct is fine when only a few options are present. When a large number of options are possible, it's customary to use `switch` instead. For instance:

```
switch units
  case 'length'
    disp('meters')
  case 'volume'
    disp('liters')
  case 'time'
    disp('seconds')
  otherwise
    disp('I give up')
end
```

The switch expression can be a string or a number. The first matching `case` has its commands executed.[13] If `otherwise` is present, it gives a default option if no case matches.

3.4 Loops: for and while

Many programs require iteration, or repetitive execution of a block of statements. Again, MATLAB is similar to other languages here.

This code illustrates the most common type of `for` loop:

```
>> f = [1 1];
>> for n = 3:10
     f(n) = f(n-1) + f(n-2);
   end
```

You can have as many statements as you like in the body of the loop. The value of the index `n` will change from 3 to 10, with an execution of the body after each assignment. But remember that `3:10` is really just a row vector. In fact, you can use *any* row vector in a `for` loop, not just one created by a colon. For example,

```
>> x = 1:100; s = 0;
>> for j = find(isprime(x))
     s = s + x(j);
   end
```

[12] If you want short-circuit behavior for logical operations outside `if` and `while` statements, you must use the special operators `||` and `&&`.

[13] Execution does not "fall through" as in C.

This finds the sum of all primes less than 100. (For a better version, though, see page 51.)

A warning: If you are using complex numbers, you might want to avoid using `i` as the loop index. Once assigned a value by the loop, `i` will no longer equal $\sqrt{-1}$. However, you can always use `1i` for the imaginary unit.

It is sometimes necessary to repeat statements based on a condition rather than a fixed number of times. This is done with `while`.

```
while x > 1
  x = x/2;
end
```

The condition is evaluated before the body is executed, so it is possible to get zero iterations. It's often a good idea to limit the number of repetitions, to avoid infinite loops (as could happen above if `x` is infinite). This can be done using `break`.

```
n = 0;
while x > 1
  x = x/2;
  n = n+1;
  if n > 50, break, end
end
```

A `break` immediately jumps execution to the first statement after the loop.

3.5 Debugging and profiling

To debug a program that doesn't work,[14] you can set **breakpoints** in one or more functions. (See the Debug menu in the Editor.) When MATLAB reaches a breakpoint, it halts and lets you inspect and modify all the variables currently in scope—in fact, you can do anything at all from the command line. You can then continue execution normally or step by step. It's also possible to set non-specific breakpoints for error and warning conditions. See the help on `debug` (or explore the menus) for all the details.

Sometimes a program spends most of its running time on just a few lines of code. These lines are then obvious candidates for optimization. You can find such lines by **profiling**, which keeps track of time spent on every line of every function. Profiling is also a great way to determine function dependencies (who calls whom). Get started by entering `profile viewer`, or find the Profiler under the Desktop menu.

[14] The term "debugging" is perhaps too limited, since the debugging tools are also very helpful in getting to understand working code that someone else has written.

3.6 Exercises

1. Write a function `quadform2` that implements the quadratic formula differently from `quadform` above (page 26). Once `d` is computed, use it to find

$$x_1 = \frac{-b - \text{sign}(b)d}{2a},$$

which is the root of largest magnitude, and then use the identity $x_1 x_2 = c/a$ to find x_2.

Use both `quadform` and `quadform2` to find the roots of $x^2 - (10^7 + 10^{-7})x + 1$. Do you see why `quadform2` is better?

2. The degree-n Chebyshev polynomial is defined by

$$T_n(x) = \cos\left[n \cos^{-1}(x)\right], \quad -1 \le x \le 1.$$

We have $T_0(x) = 1$, $T_1(x) = x$, and a recursion relation:

$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x), \quad n \ge 1.$$

Write a function `chebeval(x,N)` that evaluates *all* the Chebyshev polynomials of degree less than or equal to `N` at all of the points in column vector `x`. The result should be a matrix of size `length(x)` by `N+1`.

3. One way to compute the exponential function e^x is to use its Taylor series expansion around $x = 0$. Unfortunately, many terms are required if $|x|$ is large. But a special property of the exponential is that $e^{2x} = (e^x)^2$. This leads to a *scaling and squaring* method: Divide x by 2 repeatedly until $|x| < 1/2$, use a Taylor series (16 terms should be more than enough), and square the result repeatedly. Write a function `expss(x)` that does this. (The function `polyval` can help with evaluating the Taylor expansion.) Test your function on x values $-30, -3, 3, 30$.

4. Let `x` and `y` be column vectors describing the vertices of a polygon (given in order). Write functions `polyperim(x,y)` and `polyarea(x,y)` that compute the perimeter and area of the polygon. For the area, use a formula based on Green's theorem:

$$A = \frac{1}{2}\left|\sum_{k=1}^{n} x_k y_{k+1} - x_{k+1} y_k\right|.$$

Here n is the number of vertices and it's understood that $x_{n+1} = x_1$ and $y_{n+1} = y_1$. Test your function on a square and an equilateral triangle.

5. If a data source produces symbol k with probability p_k, the *first-order entropy* of the source is defined as

$$H_1 = -\sum_k p_k \log_2 p_k.$$

Essentially H_1 is the number of bits needed per symbol to encode a long message; i.e., it measures the amount of information content (and therefore the potential success of compression strategies). The value $H_1 = 0$ corresponds to one symbol only—no information—while for M symbols of equal probability, $H_1 = \log_2 M$.

Write a function `[H,M] = entropy(v)` that computes entropy for a vector `v`. The probabilities should be computed empirically, by counting occurrences of each unique symbol and dividing by the length of `v`. (The built-in functions `find` and `unique` may be helpful.) Try your function on some carefully selected examples with different levels of information content. Once source of data is to use `load clown; v = X(:);`. You can also find data sources from `help gallery` and (if you have the Image Processing Toolbox) `help imdemos`. You might also want to stick with integer data by using `round`.

4 More on functions

In the last section we covered the bare minimum on functions. Here we elaborate on more concepts and tools that at some point you will probably find essential.

4.1 Subfunctions and nested functions

A single M-file may hold more than one function definition. The function header line at the top of a file defines the **primary function** of the file. Two other types of functions can be in the same file: **subfunctions** and **nested functions**.

A subfunction is mostly a convenient way to avoid unnecessary files. The subfunction begins with a new header line after the end of the primary function. Every subfunction in the file is available to be called by the primary function and the other subfunctions, but they have private workspaces and otherwise behave like functions in separate files. The difference is that only the primary function, not the subfunctions, can be called from sources outside the file.[15] As a silly example, consider

```
function [x1,x2] = quadform(a,b,c)
d = discrim(a,b,c);
x1 = (-b + d) / (2*a);
x2 = (-b - d) / (2*a);
end  % quadform()

function D = discrim(a,b,c)
D = sqrt(b^2 - 4*a*c);
end  % discrim()
```

(The `end` line is optional for single-function files, but it is a good idea when subfunctions are involved and mandatory when using nested functions.) Since the subfunction `discrim` has its own workspace, changes made to `a` inside `discrim` would not propagate into the rest of `quadform`; nor would any other variables in the primary function be available to `discrim`.

Nested functions behave a little differently. They are defined *within* the scope of another function, and they share access to the containing function's workspace. For example, we can recast our quadratic formula yet again:

```
function [x1,x2] = quadform(a,b,c)

  function discrim
  d = sqrt(b^2 - 4*a*c);
  end  % discrim()
```

[15] However, see section 4.3.

```
discrim;
x1 = (-b + d) / (2*a);
x2 = (-b - d) / (2*a);
end  % quadform()
```

(The indentation, and placement of the nested function before its use, are optional. However, the nested function definition does have to be completed before the `end` of its parent.) As a nested function, `discrim` can read the values `a`, `b`, and `c` without having them passed as input, and its assignment to `d` makes a change visible to `quadform`.

The purpose of nested functions is not made clear in this example—in fact, the nested function version is inferior. But nested functions make certain important programming tasks a lot easier, as explained in section 4.3.

4.2 Anonymous functions

Sometimes you may need a quick, short function definition that doesn't seem to merit a named file on disk, or even a named subfunction. The "old-fashioned" way to do this is by using an **inline function**. Starting in MATLAB 7, a better alternative became available: the **anonymous function**. A simple example of an anonymous function is

```
sincos = @(x) sin(x) + cos(x);
```

One can easily have multiple variables:

```
w = @(x,t,c) cos(x-c*t);
```

More interestingly, anonymous functions can "capture" the values of variables that are defined at the time of the anonymous function's creation. As an illustration, consider the process of interpolation, i. e. the creation of a continuous function that passes through some given data. The built-in function `interp1` provides several methods for this purpose. Each call to `interp1` requires passing in the data to be interpolated. Using anonymous functions we can hide, or least encapsulate, this process:

```
x = 1:10;  y = rand(1,10);
f_lin = @(t) interp1(x,y,t,'linear');
f_spl = @(t) interp1(x,y,t,'spline');
plot(x,y,'o'), hold on
y = rand(1,10);  plot(x,y,'rs')
g = @(t) interp1(x,y,t,'pchip');
fplot(f_lin,[1 10],'b')
fplot(f_spl,[1 10],'k')
fplot(g,[1 10],'r')
```

The two variants of `f` always interpolate the first data set—even after the variable `y` has been changed—while `g` interpolates the second. Each of them is a function of just one independent variable, the one that is usually of interest. Reproducing the effect of the above script by other means is possible but considerably more awkward. A more advanced use of anonymous functions can be found in section 4.3.

Nested functions and anonymous functions have similarities but are not quite identical. Nested functions always share scope with their enclosing parent; their behavior can be changed by and create changes within that scope. An anonymous function can be influenced by variables that were in scope *at its creation time*, but thereafter it acts autonomously.

4.3 Function functions

Many problems in scientific computation involve operating on functions—finding roots, approximating integrals (quadrature), and solving differential equations are three of the most common examples. In many cases one can use or write a "black box" to solve this type of problem in many contexts. Since these black box functions operate on other functions, MATLAB glibly calls them **function functions**. (See the help topic `funfun` for a complete list.)

A function function needs at least one function as an input. One way to do this is by using an anonymous function (section 4.2). For example, the built-in `fzero` finds a root of a scalar function of one variable. We could say

```
>> f = @(x) sin(x);
>> fzero(f,3)

ans =
    3.1416
```

This example is needlessly indirect, however—we can pass in the sin function directly, if we refer to it using a special syntax:

```
>> fzero(@sin,3)

ans =
    3.1416
```

The name `@sin` is called a **function handle** and is a way to refer to a function abstractly, rather than invoking it immediately. By contrast, the syntax `fzero(sin,3)` would cause an error, because the `sin` function would be called first and with no input arguments.

Function handles can be created for any function that is in scope, including subfunctions and nested functions. This can be extremely handy! Suppose for instance that we want to write a function that integrates an interpolant for given data (see section 4.2). The implementation is complicated somewhat by the fact that the built-in interpolator `interp1` accepts four arguments: vectors

of the x and y coordinates of the data to be interpolated, the value(s) t where the interpolant is to be evaluated, and an optional choice of the method of interpolation. For our integration problem, the data points and the method should remain fixed while we integrate with respect to the variable t. Here is an implementation that embodies that idea:

```
function I = interpquad(x,y)

method = 'pchip';
I = quad(@interpolant,x(1),x(end));

  function f = interpolant(t)
    f = interp1(x,y,t,method);
  end  % interpolant()

end  % interpquad()
```

The `quad` function has to repeatedly call the integrand in order to do its job. We supply the integrand as a function handle to the nested function `interpolant`. Each call of `interpolant`gets a new value of `t` but also has access to the values `x`, `y` and `method` of its parent, which remain fixed throughout the course of the integration.[16]

Sooner or later you will probably have to write function functions of your own. This requires that you be able to call a function that was passed as an argument. For example, suppose for the sake of argument that instead of `quad` for integration, we want to use a much cruder type of Riemann sum as an approximation.

```
function I = quadrs(integrand,a,b,n)

h = (b-a)/n;              % subinterval width
x = a + (0:n)*h;          % endpoints of subintervals
f = integrand(x(1:n));
I = h*sum(f);
```

Here `integrand` is an anonymous function or a function handle passed in as an argument.[17]

4.4 Errors and warnings

MATLAB functions may encounter statements that are impossible to execute (for example, multiplication of incompatible matrices). In that case an **error** is thrown: execution of the function halts, a message is displayed, and the output arguments of the function are ignored. You can throw errors in your own functions with the `error` statement, called with a string that is displayed as the message. Similar to an error is a **warning**, which displays a message but allows execution to continue. You can create these using `warning`.

[16] For this simple case, an anonymous function could have been used instead of the nested function.

[17] Past versions of MATLAB required `feval` to call functions in this context, but this syntax is no longer necessary.

Sometimes you would like the ability to recover from an error in a subroutine and continue with a contingency plan. This can be done using the `try–catch` construct. For example, the following will continue asking for a statement until you give it one that executes successfully.

```
done = false;
while ~done
  state = input('Enter a valid statement: ','s');
  try
    eval(state);
    done = true;
  catch
    disp('That was not a valid statement! Look:')
    disp(lasterr)
  end
end
```

Within the `catch` block you can find the most recent error message using `lasterr`.

4.5 Exercises

1. Write a function `newton(fdf,x0,tol)` that implements Newton's iteration for rootfinding on a scalar function:
$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$
The first input is a handle to a function computing f and f', and the second input is an initial root estimate. Continue the iteration until either $|f(x_{n+1})|$ or $|x_{n+1} - x_n|$ is less than `tol`. You might want a "safety valve" as well to avoid an infinite loop.

2. Modify `newton` from the previous exercise so that it works on a system of equations $\mathbf{F}(\mathbf{x})$. The function `fdf` now returns the vector `F` and the Jacobian matrix `J`, and the update is written mathematically as
$$\mathbf{x}_{n+1} = \mathbf{x}_n - J^{-1}\mathbf{F}(\mathbf{x}_n),$$
although in numerical practice one does not compute the inverse of the Jacobian but solves a linear system of equations in which $\mathbf{F}(\mathbf{x}_n)$ is the right-hand side.

3. Write a function `I=trap(f,a,b,n)` that implements the trapezoidal quadrature rule:
$$\int_a^b f(x)\,dx \approx \frac{h}{2}\big[f(x_0) + 2f(x_1) + 2f(x_2) + \cdots + 2f(x_{n-1}) + f(x_n)\big],$$
where $h = (b-a)/n$ and $x_i = a + ih$. Test your function on $\sin(x) + \cos(x)$ for $0 \le x \le \pi/3$. For a greater challenge, write a function `simp` for Simpson's rule,
$$\int_a^b f(x)\,dx \approx \frac{h}{3}\big[f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + 2f(x_4) + \cdots + 4f(x_{n-1}) + f(x_n)\big].$$
(This formula requires n to be even. You may choose to check the input for this.)

4. The implementation of `interpquad` on page 35 is not optimal in terms of execution speed. The reason is that each call to `interp1` has to do some preliminary calculations based on `x`, `y`, and the `method` in order to compute interpolants. These preliminary calculations are independent of `t`, so computing them anew each time is very wasteful.

   Read the help on `interp1` on the piecewise polynomial option for output, and use it to modify `interpquad` to remove this problem.

5. Many simple financial instruments that have regular equal payments, such as car loans or investment annuities, can be modeled by the equation

$$F = P\left(\frac{(1+r)^t - 1}{r}\right),$$

   where P is the regular payment, r is a fixed interest rate (say, $r = 0.05$ for 5% interest), t is the number of payment intervals elapsed, and $F(t)$ is the accumulated value of the instrument at time t. This equation is not easily solved for r.

   Write a script that finds r when $P = 200$, $t = 30$, and F takes the values 10000, 15000, ..., 40000. The simplest approach is to write a loop that creates an anonymous function and calls `fzero`.

6. Lambert's W function is defined as the inverse of the function $f(x) = xe^x$. It has no simple analytic expression. Write a function `W = lambert(x)` that evaluates Lambert's W at any value of $x > 0$. For best style, `lambert` should have a subfunction that is passed in the call to `fzero`. (Hint: You are to solve the expression $x = ye^y$ for y given x. Use `fzero`.)

5 Graphics

Graphical display is one of MATLAB's greatest strengths—and most complicated subjects. The basics are quite simple, but you also get complete control over practically every aspect of each graph, and with that power comes complexity.

Graphics are represented by objects that are rigidly classified by **type**. The available types can be found in the online documentation. The most important object types are the `figure`, which is a window onscreen, the `axes`, into which data-related objects are drawn, and graphics primitives such as `line`, `surface`, `text`, and others. These types obey a hierarchy. For instance, a figure can be parent to one or more axes, each of which has graphical primitives as children, and so on. Each container type provides a local coordinate system for its children to refer to. Thus, objects drawn within an `axes` remain correctly rendered if the `axes` are moved inside its parent `figure`.

The most important aspect of all this is that graphically rendered objects have named properties that control their display. You can access or change these properties by point-and-click or through the commands, as described briefly in section 5.4. Hence, you can control the display of a graphic not only at creation time but at any later time too. The simplest and most common take on this is to add annotations such as title, legend, tick marks, axis labels, and the like to an `axes` containing data objects, but the idea can be carried much further.

5.1 2D plots

The most fundamental plotting command is `plot`. Normally it uses line segments to connect points given by two vectors of x and y coordinates. Here is a simple example.

```
>> t = pi*(0:0.02:2);
>> plot(t,sin(t))
```

A line object is drawn in the current axes of the current figure, both of which are created if not yet present. The line may appear to be a smooth, continuous curve. However, it's really just a game of connect-the-dots, as you can see clearly by entering

```
>> plot(t,sin(t),'o-')
```

Now a circle is drawn at each of the given data points. Just as `t` and `sin(t)` are really vectors, not functions, curves in MATLAB are really joined line segments.[18]

If you now say

```
>> plot(t,cos(t),'r')
```

[18] A significant difference from Maple and other packages is that if the viewpoint is rescaled to zoom in, the "dots" are *not* recomputed to give a smooth curve.

you will get a red curve representing $\cos(t)$. The curve you drew earlier (in fact, its parent `axes` too) is erased. To add curves, rather than replacing them, first enter `hold on`.

```
>> plot(t,sin(t),'b')
>> hold on
>> plot(t,cos(t),'r')
```

You can also do multiple curves in one go, if you use column vectors:

```
>> hold off
>> t = (0:0.01:1)';
>> plot(t,[t t.^2 t.^3])
```

Since it can be a drag to find points yourself that will make a nice graph, there is an alternative plotting command for a mathematical expression or any function handle (section 4.3), e. g.,

```
>> ezplot( @(x) exp(3*sin(x)-cos(2*x)), [0 4] )
```

Other useful 2D plotting commands are given in Table 6. See a bunch more by typing `help graph2d`.

Table 6: 2D plotting commands

`figure`	Open a new figure window.
`subplot`	Multiple axes in one figure.
`semilogx`, `semilogy`, `loglog`	Logarithmic axis scaling.
`axis`, `xlim`, `ylim`	Axes limits.
`legend`	Legend for multiple curves.
`print`	Send to printer.

You may zoom in to particular portions of a plot by clicking on the magnifying glass icon in the figure and drawing a rectangle.

5.2 3D plots

Plots of surfaces and such for functions $f(x, y)$ also operate on the connect-the-dots principle, but the details are more difficult. The first step is to create a grid of points in the xy plane. These are the points where f is evaluated to get the "dots" in 3D.

Here is a typical example:

```
>> x = pi*(0:0.02:1);
>> y = 2*x;
>> [X,Y] = meshgrid(x,y);
>> surf(X,Y,sin(X.^2+Y))
```

Once a 3D plot has been made, you can use the rotation button in the figure window to manipulate the 3D viewpoint. There are additional menus that give you much more control of the view, too.

The `surf` plot begins by using `meshgrid` to make an x-y grid that is stored in the arrays `X` and `Y`. To see the grid graphically, use

```
plot(X(:),Y(:),'k.')
```

With the grid so defined, the expression `sin(X.^2+Y)` is actually an array of values of $f(x, y) = \sin(x^2 + y)$ on the grid. (This array could be assigned to another variable if you wish.) Finally, `surf` makes a solid-looking surface in which color and apparent height describe the given values of f. An alternative command `mesh` is similar to `surf` but makes a "wireframe" surface. The most common 3D plotting commands are shown in Table 7. There are also `ezsurf`, `ezcontour`, and the like when you don't want to find the data points manually.

Table 7: 3D plotting commands

`surf`, `mesh`, `waterfall`	Surfaces in 3D.
`colorbar`	Show color scaling.
`plot3`	Curves in space.
`pcolor`	Top view of a colored surface.
`contour`, `contourf`	Contour plot.

5.3 Annotation

Graphs of data usually need labels and maybe a title. For example,

```
>> t = 2*pi*(0:0.01:1);
>> plot(t,sin(t))
>> xlabel('time')
>> ylabel('amplitude')
>> title('Simple Harmonic Oscillator')
```

You can also add legends, text, arrows, or text/arrow combinations to help label diffrent data. See the Insert menu on the figure window.

5.4 Handles and properties

Every rendered object has an identifier or **handle**. The functions `gcf`, `gca`, and `gco` return the handles to the active figure, axes, and object (usually the most recently drawn).

Properties can be accessed and changed by the functions `get` and `set`, or graphically (see the Plot Edit Toolbar, Plot Browser, and Property Editor in the figure's View menu). Here is just a taste of what you can do:

```
>> h = plot(t,sin(t))
>> set(h,'color','m','linewidth',2,'marker','s')
>> set(gca,'pos',[0 0 1 1],'visible','off')
```

Here is a way to make a "dynamic" graph or simple animation:

```
>> clf, axis([-2 2 -2 2]), axis equal
>> h = line(NaN,NaN,'marker','o','linestyle','-','erasemode','none');
>> t = 6*pi*(0:0.02:1);
>> for n = 1:length(t)
     set(h,'XData',2*cos(t(1:n)),'YData',sin(t(1:n)))
     pause(0.05)
   end
```

From this example you see that even the data displayed by an object are settable properties (XData and YData). Among other things, this means you can extract the precise values used to plot any object from the object itself.

Because of the way handles are used, plots in MATLAB are usually created first in a basic form and then modified to look exactly as you want. However, it can be useful to change the default property values that are first used to render an object. You can do this by resetting the defaults at any level above the target object's type. For instance, to make sure that all future Text objects in the current figure have font size 10, enter

```
>> set(gcf,'defaulttextfontsize',10)
```

All figures are also considered to be children of a `root` object that has handle `0`, so this can be used to create global defaults. Finally, you can make it a global default for all future MATLAB sessions by placing the `set` command in a file called `startup.m` in a certain directory (see the documentation).

5.5 Color

The coloring of lines and text is easy to understand. Each object has a Color property that can be assigned an RGB (red, green, blue) vector whose entries are between zero and one. In addition many one-letter string abbreviations are understood (see `help plot`).

Surfaces are different. You specify color data (CData) at all the points of your surface. In between the points the color is determined by **shading**. In **flat shading**, each face or mesh line has constant color determined by one boundary point. In **interpolated shading**, the color is

determined by interpolation of the boundary values. While interpolated shading makes much smoother and prettier pictures, it can be very slow to render, particularly on printers.[19] Finally, there is **faceted shading** which uses flat shading for the faces and black for the edges. You select the shading of a surface by calling `shading` after the surface is created.

Furthermore, there are two models for setting color data:

Indirect Also called **indexed**. The colors are not assigned directly, but instead by indexes in a lookup table called a **colormap**. This is how things work by default.

Direct Also called **truecolor**. You specify RGB values at each point of the data.

Direct color is more straightforward conceptually, but it produces bigger files and is most suitable for photos and similar images.

Here's how indirect mapping works. Just as a surface has XData, YData, and ZData properties, with axes limits in each dimension, it also has a CData property and "color axis" limits. The color axis is mapped linearly to the colormap, which is a 64×3 list of RGB values stored in the figure. A point's CData value is located relative to the color axis limits in order to look up its color in the colormap. By changing the figure's colormap, you can change all the surface colors instantly. Consider these examples:

```
>> [X,Y,Z] = peaks;    % some built-in data
>> surf(X,Y,Z), colorbar
>> caxis               % current color axis limits

ans =
   -6.5466    8.0752

>> caxis([-8 8]), colorbar  % a symmetric scheme
>> shading interp
>> colormap pink
>> colormap gray
>> colormap(flipud(gray))   % reverse order
```

By default, the CData of a surface is equal to its ZData. But you can make it different and thereby display more information. One use of this is for functions of a complex variable.

```
>> [T,R] = meshgrid(2*pi*(0:0.02:1),0:0.05:1);
>> [X,Y] = pol2cart(T,R);
>> Z = X + 1i*Y;
>> W = Z.^2;
>> surf(X,Y,abs(W),angle(W)/pi)
>> axis equal, colorbar
>> colormap hsv   % ideal for this situation
```

[19] In fact it's often faster on a printer to interpolate the data yourself and print it with flat shading. See `interp2` to get started on this.

5.6 Saving and exporting figures

It often happens that a figure needs to be changed long after its creation. You can save the figure-creation commands in a script (section 3.1), but this has drawbacks. If the plot data take a long time to generate, re-running the script will waste time. Also, any edits made through menus and buttons will be lost.

Instead you can save a figure in a format that allows it to be recreated in future sessions. Just enter

```
>> saveas(gcf,'myfig.fig')
```

to save the current figure in a file `myfig.fig`. (This is equivalent to "Save as..." on the figure's File menu.) Later you can enter `openfig myfig` to recreate it. You can also use "Generate M-file..." on the File menu to create an M-file that will apply the same look as in the current graph to new data.

Ultimately, you will want to save some figures for inclusion in a presentation or publication. Understand that what you see on the screen is not necessarily what you get on paper, because MATLAB tweaks graphs depending on the output device. There are three major issues here: file format, graphics size and position, and color.

The big difference in graphics file formats is between *vector* (representing the lines in an image) and *bitmap* (a pixel-by-pixel snapshot) graphics. Bitmaps, including GIF, JPEG, PNG, and TIFF, are ideal for photographs, but for most other scientific applications they are a bad idea. These formats fix the resolution of your image forever, whereas the resolution of your screen, your printer, and a journal's printer are all very different. Vector formats are usually a much better choice. They include EPS and WMF. The choice here depends somewhat on your platform and word processor.

EPS files (Encapsulated PostScript) are the best choice for documents in LaTeX.[20] [21] For example, to export the current MATLAB figure to file `myfig.eps`, use

```
>> saveas(gcf,'myfig.eps')
```

or

```
>> print -deps myfig
```

For color output use `-depsc` in the second position. EPS also works with MS Word, if you print output on a postscript printer. In this case it's handy to use

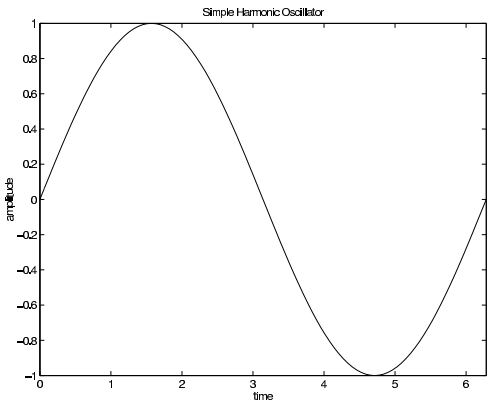
```
>> print -deps -tiff myfig
```

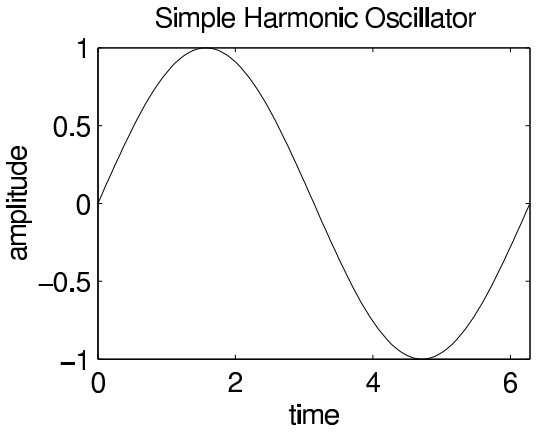
[20] However, for LaTeXdocuments (such as this one) outputting directly to PDF, one must use a bitmap format, preferably PNG.

[21] Check out the LaTeXpackage **psfrag** for putting any text or math formula anywhere in an EPS graphic.

in which case Word will be able to show a crude version of the graph in the document on screen.

A common problem with publishing MATLAB graphs has to do with size. By default, MATLAB figures are rendered at 8 inches by 6 inches on paper. This is great for an isolated output but much too large for most journal papers. It's easy in LaTeXand other word processors to rescale the image to a more reasonable size. However, this reduces *everything*, including the thickness of lines and the size of text, and the result may be unreadable. Instead you should consider rescaling the graph *before* exporting it. Here are two versions of an annotated graph. On the left, the figure was saved at default size and then rescaled in LaTeX. On the right, the figure was rescaled first.





In practice, a compromise between these two would probably be best.

To scale the size a figure will be after export, before saving you need to enter

```
>> set(gcf,'paperpos',[0 0 3 2.25])
```

where the units are in inches. (Or see the File/Page Setup menu of the figure.) Unfortunately, sometimes the axes or other elements need to be repositioned. To make the display match the paper output, you should enter

```
>> set(gcf,'unit','inch','pos',[0 0 3 2.25])
```

Most journals do not normally accept color figures. Colored lines are automatically converted to black when saved in a non-color format, so you should distinguish them by other features, such as symbol or line style. The colors of surfaces are converted to grayscale, which presents a problem. By default, the colors on a surface range from deep blue to deep red, which in grayscale look similar. You should consider using `colormap(gray)` or `colormap(flipud(gray))`, whichever gives less total black, before exporting the figure. Finally, the edges of wireframe surfaces created by `mesh` are also converted to gray, often with poor results. Make them all the lines black by entering

```
>> set(findobj(gcf,'type','surface'),'edgecolor','k')
```

or by pointing and clicking.

The "File/Export setup..." option on any figure window provides a mechanism for imposing entire sets of changes in figure appearance, including size and color conversion, for different contexts (print, presentation, etc.).

Finally, it's worth remembering that figures can be saved in `fig` format and recreated later. It's a good idea to save every figure both as `fig` and EPS in the same directory.

5.7 Exercises

1. Recall the identity
$$e = \lim_{n\to\infty} r_n, \qquad r_n = \left(1 + \frac{1}{n}\right)^n.$$
Make a standard and a log-log plot of $e - r_n$ for $n = 5, 10, 15, \ldots, 500$. What does the log-log plot say about the asymptotic behavior of $e - r_n$?

2. Here are two different ways of plotting a sawtooth wave. Explain concisely why they behave differently.

```
x = [ 0:7; 1:8 ];  y = [ zeros(1,8); ones(1,8) ];
subplot(121), plot(x,y,'b'), axis equal
subplot(122), plot(x(:),y(:),'b'), axis equal
```

(The first version is more mathematically proper, but the second is more likely to appear in a journal article.)

3. Play the "chaos game." Let P_1, P_2, and P_3 be the vertices of an equilateral triangle. Start with a point anywhere inside the triangle. At random, pick one of the three vertices and move halfway toward it. Repeat indefinitely. If you plot all the points obtained, a very clear pattern will emerge. (Hint: This is particularly easy to do if you use complex numbers. If `z` is complex, then `plot(z)` is equivalent to `plot(real(z),imag(z))`.)

4. Make surface plots of the following functions over the given ranges.
   (a) $(x^2 + 3y^2)e^{-x^2-y^2}$, $-3 \le x \le 3$, $-3 \le y \le 3$
   (b) $-3y/(x^2 + y^2 + 1)$, $|x| \le 2$, $|y| \le 4$
   (c) $|x| + |y|$, $|x| \le 1$, $|y| \le 1$

5. Make contour plots of the functions in the previous exercise.

6. Make a contour plot of

$$f(x,y) = e^{-(4x^2+2y^2)}\cos(8x) + e^{-3((2x+1/2)^2+2y^2)}$$

for $-1.5 < x < 1.5$, $-2.5 < y < 2.5$, showing only the contour at the level $f(x,y) = 0.001$. You should see a friendly message.

7. Parametric surfaces are easily done in MATLAB. Plot the surface represented by

$$x = u(3+\cos(v))\cos(2u), \qquad y = u(3+\cos(v))\sin(2u), \qquad z = u\sin(v) - 3u$$

for $0 \le u \le 2\pi$, $0 \le v \le 2\pi$. (Define `U` and `V` as a grid over the specified limits, use them to define `X`, `Y`, and `Z`, and then use `surf(X,Y,Z)`.)

6 Speed, style, and trickery

MATLAB is oriented towards minimizing development and interaction time, not computational time. In some cases even the best MATLAB code may not keep up with good C code, but the gap is not always wide. In fact, on core linear algebra routines such as matrix multiplication and linear system solution, there is very little practical difference in performance. By writing good MATLAB programs—starting with profiling your code (section 3.5) to find out where the bottlenecks are—you can often nearly recover the speed of compiled code.[22]

Since you normally spend quite a bit of time with your code, it pays to write it well. MATLAB's language has features that can make certain operations, most commonly those involving loops in C or FORTRAN, quite compact. There are also some conventions and quirks that are worth knowing about.

6.1 Functions or scripts?

Scripts are always interpreted and executed one line at a time. No matter how many times you execute the same script, MATLAB must spend time parsing your syntax. By contrast, functions are effectively compiled into memory when called for the first time (or modified). Subsequent invocations skip the interpretation step.

Most of your programming should be done in functions, which also require you to carefully state your input and output assumptions. Use scripts for drivers or attention-free execution, as described in section 3.1. As a rule of thumb, call scripts only from the command line, and do not call other scripts from within a script.

6.2 Memory preallocation

MATLAB hides the tedious process of allocating memory for variables. This generosity can cause you to waste a lot of runtime, though. Consider an implementation of Euler's method for the vector differential equation $y' = Ay$ in which we keep the value at every time step:

```
A = rand(100);
y = ones(100,1);
dt = 0.001;
for n = 1:(1/dt)
  y(:,n+1) = y(:,n) + dt*A*y(:,n);
end
```

This takes about 7.3 seconds on a certain PC. Almost all of this time, though, is spent on a non-computational task.

[22] In a pinch, you can write a time-consuming function alone in C and link the compiled code into MATLAB. See the online help under "External interfaces."

When MATLAB encounters the statement `y = ones(100,1)` representing the initial condition, it asks the operating system for a block of memory to hold 100 numbers. On the first execution of the loop, it becomes clear that we actually need space to hold 200 numbers, so a new block of this size is requested. On the next iteration, this also becomes obsolete, and more memory is allocated. The little program above requires 1001 memory allocations of increasing size, and this task occupies most of the execution time. Also, the obsoleted space affects your other software, because (at least in UNIX-like environments) MATLAB can't give memory back to the operating system until it exits.

Changing the second line to `y = ones(100,1001);` changes none of the mathematics but does all the required memory allocation at once. This is called **preallocation**. With preallocation the program takes about 0.4 seconds on the same computer as before.

6.3 Vectorization

Vectorization refers to the removal of loops (`for` and `while`). As an example, suppose x is a column vector and you want to compute a matrix D such that $d_{ij} = x_i - x_j$. The standard implementation would involve two nested loops:

```
n = length(x);
D = zeros(n);   % preallocation
for j = 1:n
  for i = 1:n
    D(i,j) = x(i) - x(j);
  end
end
```

The loops could be written in either order here. But the innermost loop is easily replaced by a vector operation.

```
n = length(x);
D = zeros(n);   % preallocation
for j = 1:n
  D(:,j) = x - x(j);
end
```

We can get rid of the remaining loop, too, by upgrading from vectors to two-dimensional arrays. This is a bit more subtle.

```
n = length(x);
X = x(:,ones(n,1));   % copy columns to make n by n
D = X - X.';
```

The second line here is a trick that was introduced in section 2.2 (and in MATLAB circles is called "Tony's trick"). Note too the use of `.'` in the last line to be compatible with a complex input.

At this point you might ask, why vectorize? There are two answers: speed and style. Neither is a simple issue. Until around 2002, careful vectorization always yielded tremendous improvements in execution speed—often orders of magnitude. But this is changing due a technique called **JIT acceleration**. Acceleration, which is applied automatically, can remove the speed penalty that MATLAB traditionally experiences with loops. At this writing, not every loop can be optimized, but it's clear that code vectorization is no longer speed-critical in every case. For example, with $n = 200$ the the times in milliseconds for each of the three methods above were 1.73, 0.97, and 2.80, respectively. In this case a medium level of vectorization proved fastest. (Observe also that the first two versions, unlike the third, are easily adjusted to account for the obvious antisymmetry in the result.)

Style is a subjective matter, of course. To get a better idea of the trade-offs involved, consider the classic example of Gaussian elimination. Here is a basic implementation without any vectorization.

```
n = length(A);
for k = 1:n-1
  for i = k+1:n
    s = A(i,k)/A(k,k);
    for j = k:n
      A(i,j) = A(i,j) - s*A(k,j);
    end
  end
end
```

Again we start vectorization with the innermost loop, on `j`. Each iteration of this loop is independent of all the others. This parallelism is a big hint that we can use a vector operation:

```
n = length(A);
for k = 1:n-1
  for i = k+1:n
    s = A(i,k)/A(k,k);
    cols = k:n;
    A(i,cols) = A(i,cols) - s*A(k,cols);
  end
end
```

This version is actually makes the linear algebra idea of a row operation much more apparent and, in my view, is clearly preferable. However, the innermost of the remaining loops is also vectorizable.

```
n = length(A);
for k = 1:n-1
  rows = k+1:n;
  cols = k:n;
  s = A(rows,k)/A(k,k);
```

```
  A(rows,cols) = A(rows,cols) - s*A(k,cols);
end
```

You have to flex your linear algebra muscles a bit to see that the vector outer product in the next-to-last line is appropriate. This is an interesting insight, but does not lead to an unquestionable improvement in style.

To compare the speed of these three versions, they were run 20 times each for six different values of the matrix size n on a 2.5GHz Linux machine in MATLAB 7.0.1. The results, per factorization in milliseconds:

	$n = 50$	100	150	200	250	300
Three loops	1.5	14.5	41	108.5	231	419.5
Two loops	17.5	99.5	258.5	521	918	1465
One loop	3	12	76.5	197.5	439.5	672

The version that works naturally at the vector level is now by far the *slowest*. Besides pointing out the pitfalls of vectorization, this experiment illustrates the limitations of comparing algorithms by counting their floating-point operations. In the two-loop version the growth in time is less than the asymptotically expected $O(n^3)$, indicating that non-arithmetic operations are playing a major role.

An uncontroversial application of vectorization is the use of array operators in the evaluation of a mathematical expression such as $t \sin(t^2)$ at many values of t. It's hard to deny that `t.*sin(t.^2)` is plainer and shorter than an equivalent loop. Similarly, once you understand the `cumsum` function for cumulative summation, computing partial sums of a geometric series by

```
s = cumsum( (1/3).^(0:100) );
```

is a little shorter and more transparent than the equivalent loop, which also requires a preallocation (section 6.2):

```
s = ones(1,101);
for j = 1:100,
  s(j+1) = s(j) + (1/3)^j;
end
```

Functions like `sum` and `diff` listed in Table 5 can make code more readable in some cases.

The bottom line: Loops should not be written casually. Writing code that works at a vector level is easy and natural in most cases. But if profiling indicates that a lot of time is being spent in a place where the level of vectorization is selectable, experimentation may be the only way to see what level is best. In the future, MATLAB may become even more forgiving when it comes to loops.

6.4 Masking

An advanced type of vectorization is called **masking**. Let's say that we have a vector `x` of values at which we want to evaluate the piecewise-defined function

$$f(x) = \begin{cases} 1 + \cos(2\pi x), & |x| \le \frac{1}{2} \\ 0, & |x| > \frac{1}{2}. \end{cases}$$

Here is the standard loop method:

```
f = zeros(size(x));
for j = 1:length(x)
  if abs(x(j)) <= 0.5
    f(j) = 1 + cos(2*pi*x(j));
  end
end
```

The shorter way is to use a mask.

```
f = zeros(size(x));
mask = (abs(x) < 0.5);
f(mask) = 1 + cos(2*pi*x(mask));
```

The mask is a logical index into `x` (see page 14). You could refer, if needed, to the unmasked points by using `~mask`.

Consider a new version of the sum-of-primes idea from page 28. Here's how we could count the number of primes less than 100 and add them up:

```
isprm = isprime(1:100);
sum( isprm )
sum( find(isprm) )
```

Here `find` converts a logical index into an absolute one. The only disadvantage of doing so in general is that referring to the unmasked elements becomes more difficult.

Generally, when a graphical function encounters a `NaN` value, it quietly omits drawing that point. This can be useful when combined with masking. For example, the commands

```
L = membrane(1,18,8,8);
surf(L)
```

plots a surface defined over an L-shaped region. The part outside the L is flat at zero, which works OK since the function goes to zero at the boundary. But it looks better if we erase that part of the surface using a mask:

```
[X,Y] = meshgrid(0:36);
outside = (x < 18) & (y > 18);  % logical mask
L(outside) = NaN;
surf(X,Y,L)
```

6.5 Scoping exceptions

Once in a while the scoping rules for functions get in the way. Although you can almost always do what you need within the rules, it's nice to know how to bend them.

The least useful and most potentially troublesome violation of scoping comes from **global variables**. Any function (or the user at the command line) can declares a variable to be `global` before assigning it a value. Then any other workspace may also declare it global and see or change its value. At one time global values were more or less necessary in MATLAB, but that is no longer the case. They should *not* be used, for example, to pass extra parameters into "function functions." As described in section 4.3, there are better and more stable means of doing so. The primary problem with global variables is that it becomes possible to have conflicting names, or to lose track of what functions may modify a value. Input and output parameters make this information much more apparent.

A more interesting type of variable is called **persistent**. One use of persistent variables is to compute some preliminary data that needs to be used on subsequent calls. Although the data could be returned to the caller and passed back in to the function, that is inconvenient when such data are meaningless to the caller. Consider this example for the Fibonacci numbers.

```
function y = fib(n)

persistent f
if length(f) < 2
  f = [1 1];
end
for k = length(f)+1:n
  f(k) = f(k-2) + f(k-1);
end
y = f(1:n);
```

The first time this function is called, `f` will be empty.[23] Immediately `f` will then be given the first two values in the sequence. (The first invocation is the only one in which the conditional of line 5 will be true.) Then, `f` is extended as needed to get the first `n` values. So far this is nothing unusual, if it is a bit indirect. But upon exit of the function, the persistent `f` is not destroyed. Thus future calls will begin with an `f` that has as many entries as the largest `n` encountered in all calls. The same effect could be achieved with a `global` variable, but a `persistent` variable is accessible only to the function that created it. In fact, different functions can use the same name for a `persistent` variable without interference.

[23] `persistent` variables, unlike others, are given an initial value: the empty matrix.

6.6 Exercises

1. Rewrite `trap` or `simp` (page 36) so that it does not use any loops. (Hint: Set up a vector of all x values and evaluate `f` for it. Then set up a vector of quadrature weights and use an inner product.)

2. Explore the issue of whether the order of the loops affects the execution time of either of the first two versions of the vector differencing code given on page 48.

3. Suppose x is a column vector. Compute, without using loops or conditionals, the matrix A given by

$$a_{ij} = \begin{cases} \frac{1}{(x_i - x_j)^2}, & \text{if } i \neq j, \\ 1, & \text{otherwise.} \end{cases}$$

   (One way to do this is by direct assignment of the diagonal elements of A. This is impossible without a loop using row and column indices, but easy using the single-index linear addressing model.)

4. Reconsider the function `chebeval` (page 30) that evaluates Chebyshev polynomials at multiple points. Write the function so that it performs as efficiently as you can make it. Among other things, you have to choose whether to use the definition or the recursion.

5. Consider "shuffling" a vector of integers from 1 to 52 using a physical interpretation of a card shuffle. Divide the cards into two equal stacks, and merge the stacks together such that each time a pair of cards is to fall off the "bottom" of each stack, a random decision is made as to which falls first. A loop-based implementation would be

```
function s = shuffle(x)
n = length(x)/2;
s = [];
for i = 1:n
  if rand(1) > 0.5
    s = [s x([2*i-1 2*i])];
  else
    s = [s x([2*i 2*i-1])];
  end
end
```

   Rewrite this function without loops. (This can be done in as little as four statements using resizing tricks.) It can be interesting to start with a perfectly ordered deck and see how many shuffles it takes to "randomize" it. One crude measure of randomness is the (1,2) element of `corrcoef(1:52,v)`, which is expected to be zero if `v` is random.

6. Rewrite the function `entropy` on page 30 without any loops using `sort`, `diff`, `find`, and (perhaps) `sum`.

7. In the function `newton` (page 36), suppose that input `x0` is actually a vector of initial guesses, and you want to run Newton's method on each. Keep track of an error vector and use masking to rewrite `newton` so that it still uses only one loop.

8. Rewrite `expss` from page 30 so that it works on a vector input and as fast as possible. (You should consider, for example, whether making use of `persistent` variables might help.)

9. Different Fibonacci sequences can be produced by changing the first two members of the sequence. Rewrite `fib` from page 52 so that it accepts these seed values and recomputes the sequence *only* when necessary.

7 Advanced data structures

Not long ago, MATLAB viewed every variable as a matrix or array. While this point of view is ideal for simplicity, it is too limited for some tasks. A few additional data types are provided.

7.1 Strings and formatted output

As we have seen, a string in MATLAB is enclosed in single forward quotes. In fact a string is really just a row vector of character codes. Because of this, strings can be concatenated using matrix concatenation.

```
>> str = 'Hello world';
>> str(1:5)

ans =

Hello

>> double(str)

ans =
    72   101   108   108   111    32   119   111   114   108   100

>> char(ans)

ans =

Hello world

>> ['Hello',' ','world']

ans =

Hello world
```

You can convert a string such as '3.14' into its numerical meaning (not its character codes) by using `eval` or `str2num` on it. Conversely, you can convert a number to string representation using `num2str` or the much more powerful `sprintf` (see below). If you want a quote character within a string, use two quotes, as in `'It''s Cleve''s fault'`.

Multiple strings can be stored as rows in an array using `str2mat`, which pads strings with blanks so that they all have the same length. However, a better way to collect strings is to use cell arrays (section 7.2).

There are lots of string handling functions. See the help on `strfun`. Here are a few:

```
>> upper(str)

ans =

HELLO WORLD

>> strcmp(str,'Hello world')

ans =

     1

>> findstr('world',str)

ans =

     7
```

For the best control over conversion of numbers to strings, use `sprintf` or `fprintf`. These are closely based on the C function *printf*, with the important vectorization enhancement that format specifiers are "recycled" through all the elements of a vector or matrix (in the usual row-first order).

For example, here's a script that prints out successive Taylor approximations for $e^{1/4}$.

```
x=0.25;  n=1:6;  c=1./cumprod([1 n]);
for k=1:7,  T(k)=polyval(c(k:-1:1),x);  end
fprintf('\n     T_n(x)       |T_n(x)-exp(x)|\n');
fprintf('-----------------------------------\n');
fprintf('%15.12f      %8.3e\n', [T;abs(T-exp(x))] )
```

```
     T_n(x)       |T_n(x)-exp(x)|
-----------------------------------
 1.000000000000      2.840e-01
 1.250000000000      3.403e-02
 1.281250000000      2.775e-03
 1.283854166667      1.713e-04
 1.284016927083      8.490e-06
 1.284025065104      3.516e-07
 1.284025404188      1.250e-08
```

Use `sprintf` if you want to save the result as a string rather than have it output immediately.

7.2 Cell arrays

Collecting objects of different sizes is a common chore. For instance, suppose you want to tabulate the Chebyshev polynomials 1, x, $2x^2 - 1$, $4x^3 - 3x$, and so on. In MATLAB one expresses a

polynomial as a vector (highest degree first) of its coefficients. The number of coefficients needed grows with the degree of the polynomial. Although you can put all the Chebyshev coefficients into a triangular array, this is an inconvenient complication.

Cell arrays are used to gather dissimilar objects into one variable. They are indexed like regular numeric arrays, but their elements can be absolutely anything. A cell array is created or referenced using curly braces {} rather than parentheses.

```
>> str = { 'Goodbye', 'cruel', 'world' }

str =

    'Goodbye'    'cruel'    'world'

>> str{2}

ans =

cruel

>> T = cell(1,9);
>> T(1:2) = { [1], [1 0] };
>> for n = 2:8,  T{n+1} = [2*T{n} 0] - [0 0 T{n-1}];  end
>> T

T =
  Columns 1 through 5

    [1]    [1x2 double]    [1x3 double]    [1x4 double]    [1x5 double]

  Columns 6 through 9

    [1x6 double]    [1x7 double]    [1x8 double]    [1x9 double]

>> T{4}

ans =
     4     0    -3     0
```

Cell arrays can have any size and dimension, and their elements do not need to be of the same size or type. Cell arrays may even be nested. Because of their generality, cell arrays are mostly just containers; they do not support any sort of arithmetic.

One special cell syntax is quite useful. The idiom `C{:}` for cell array `C` is interpreted as a comma-separated list of the elements of `C`, just as if they had been typed. For example,

```
>> str2mat(str{:})  % same as str2mat('Goodbye','cruel','world')
```

```
ans =
Goodbye
cruel
world
```

The special cell array `varargin` is used to pass optional arguments into functions. For example, reconsider the function `interpquad` on page 35 for integrating an interpolant. The built-in `interp1` used there to perform the interpolation accepts an optional input parameter to select the type of interpolant used. We can update `interpquad` to produce exactly the same behavior, as follows:

```
function I = interpquad(x,y,varargin)

I = quad(@interpolant,x(1),x(end));

  function f = interpolant(t)
  f = interp1(x,y,t,varargin{:});
  end  % interpolant()

end  % interpquad()
```

The nested function has access to `varargin`. The syntax `varargin{:}` puts all input arguments after the first two into each call to `interp1`. Thus, for example,

```
>> x=pi*sort(rand(100,1));
>> interpquad(x,sin(x))  % default choice
ans =
    1.9987

>> interpquad(x,sin(x),'linear')
ans =
    1.9987

>> interpquad(x,sin(x),'pchip')
ans =
    1.9994
```

Note how an empty `varargin` is valid too and gives the default `interp1` behavior (linear).

7.3 Structures

Structures are much like cell arrays, but they are indexed by names rather than by numbers.

Say you are keeping track of the grades of students in a class. You might start by creating a student `struct` (structure) as follows:

```
>> student.name = 'Moe';
>> student.SSN = 123456789;
>> student.homework = [10 10 7 9 10];
>> student.exam = [98 94];
>> student

student =

        name: 'Moe'
         SSN: 123456789
    homework: [10 10 7 9 10]
        exam: [98 94]
```

The name of the structure is `student`. Data is stored in the structure according to named **fields**, which are accessed using the dot notation above. The field values can be anything.

Probably you have more students.

```
>> student(2).name = 'Curly';
>> student(2).SSN = 987654321;
>> student(2).homework = [4 6 7 3 0];
>> student(2).exam = [53 66];
>> student

student =
1x2 struct array with fields:
    name
    SSN
    homework
    exam
```

Now you have an array of structures. This array can have any size and dimension. However, all elements of the array must have the same fields.

Struct arrays make it easy to extract data into cells:

```
>> [roster{1:2}] = deal(student.name)

roster =

    'Moe'    'Curly'
```

In fact, `deal` is a very handy function for converting between cells and structs. See online help for many examples.

8 Scientific computing

Certain computational problems appear repeatedly in applications motivated by science and engineering. Naturally, MATLAB is thoroughly equipped for such problems. In this section are reviews of some of these methods from a "recipe" point of view. Deeper mathematical understanding, which is vital when things do not work out according to plan, can be found in texts and papers on numerical analysis.

8.1 Linear algebra

The most common tasks in numerical linear algebra are solving linear systems $Ax = b$ or overdetermined least-squares systems $Ax \approx b$, and finding factorizations such as the eigenvalue or singular value decompositions. The greatest dichotomy in methods for these problems is full versus sparse matrices. (For least-squares problems, however, sparse situations seem to be relatively rare and less supported.) See section 2.5 for more on sparse matrices.

For square or rectangular linear systems the method of first resort is the backslash operator. The solution to either $Ax = b$ or $Ax \approx b$ is implemented in MATLAB as `x=A\b`. For square matrices, in fact, the idiom `A\` is mathematically equivalent and computationally preferred to left-multiplication by A^{-1}.[24] The backslash does not apply a single algorithm but rather represents an expert system that applies the best of several methods depending on detectable properties of A (such as sparsity, triangularity, and positive definiteness). These are spelled out in the documentation page for `mldivide`. One consequence is that you must set up A properly to get the fastest solution—for instance, if A is tridiagonal, you should make it formally sparse to get an $O(n)$ solution time.

If you need to solve multiple systems $Ax_i = b_i$, for $i = 1, \ldots, k$, you can use the identity

$$\begin{bmatrix} b_1 & b_2 & \cdots & b_k \end{bmatrix} = \begin{bmatrix} Ax_1 & Ax_2 & \cdots & Ax_k \end{bmatrix} = A \begin{bmatrix} x_1 & x_2 & \cdots & x_k \end{bmatrix}.$$

In MATLAB terms one lets `B` be the matrix whose columns are $b_1, \ldots, b_k$, and then `X=A\B` has the solutions $x_1, \ldots, x_k$ as its columns. But if the right-hand side vectors b_i are not all known at once (e. g., they are produced by an iteration), then backslash is much less useful, because it has to repeat costly work each time. In this case you have to do the key factorization step yourself if you want efficiency. For instance, the so-called inverse iteration for an eigenvector of A could look like

```
[L,U] = lu(A);
v = randn(length(A),1);  v = v/v(1);
for k = 1:50
  v = U \ (L\v);
  v = v/v(1);
end
```

[24] To right-multiply by A^{-1}, use a forward slash, as in `x=b/A`.

Note that `U \ (L\v)` is equivalent to $U^{-1}(L^{-1}v) = (LU)^{-1}v = A^{-1}v$. This version is fine for a generic A but less than ideal if A has one of the special properties detected by the backslash. At this point one really needs to know some numerical linear algebra to make further improvements.

Eigenvalue and singular value decompositions can be found using `eig` and `svd`, respectively. Both functions have two output formats:

```
lambda = eig(A);
[V,D] = eig(A);
```

```
sigma = svd(A);
[U,S,V] = svd(A);
```

The top line returns just a vector of eigenvalues or singular values; the bottom line returns the full decomposition. For algorithmic reasons, the value-only forms can be substantially faster than the full versions, so use the full versions only when you need them.

8.2 Iterative linear algebra

The methods of section 8.1 are appropriate for matrices of size in the several thousands for linear systems, or perhaps about a thousand for the decompositions. For truly large problems these direct methods, which give answers at the end of a finite number of operations[25], are less useful than iterative methods, which produce sequences of improving estimates. Iterative methods also do not factor or modify the given matrix, which is ideal for sparse problems.

For linear systems there are many choices of iterative methods (see the help for `sparfun`). All of these tend to be much better than classical methods such as Gauss–Seidel or SOR. If the system matrix is positive definite, the usual choice is the **conjugate gradient** method, implemented by `pcg`. For more general matrices, `bicgstab` and `gmres` tend to be quite popular, but there is no "best" method. In practice the choice of method is often less critical than the choice of a **preconditioner**, which is a matrix M that is somehow "close to" A yet allows very fast solutions of $Mx = b$.

For the decompositions the choices are simpler: `eigs` for eigenvalues and `sigs` for singular values. Both are much preferred to the power iteration and related classical methods. Usually one does not seek the full decomposition (which might be impractical even to store), but a few of the largest or smallest values and perhaps their associated vectors.

All of these iterative methods have a surprising additional flexibility: they don't require matrices! They can instead be given a function that returns the vector Au for any given vector u. This is more useful than it may sound, for example in applications where one can use multipole or other "fast" approximate methods for the matrix-times-vector operation.

[25] Technically, all eigenvalue and singular value methods must be iterative. However, the classical methods used by `eig` and `svd` converge very quickly and in practice spend most of their time in a finite preprocessing step.

8.3 Rootfinding

One of the oldest and most common numerical problems is trying to find a root or zero of a scalar function $f(x)$ of one variable. (This is equivalent to solving $f(x) = c$ for any constant c, since in that case we can find a root of the function $f(x) - c$.) Usually one learns Newton's method for this problem, but it turns out not to be the best general-purpose method because it needs an explicit derivative calculation and has erratic convergence. The MATLAB function `fzero` is much better than a handwritten algorithm for most problems. It can use a starting point or (preferably) a starting interval containing a root, and requires a function only to compute $f(x)$, not $f'(x)$.

Roots of polynomials are handled differently using `roots`. You should be aware that polynomial roots can be very sensitive to perturbations in the coefficients, so interpret the results with some caution.

For solving multidimensional systems of nonlinear equations $F(x) = 0$, MATLAB offers `fsolve` in the Optimization Toolbox. You only need to supply a function for computing F and a starting point, but you might greatly improve the convergence if you also supply a function computing the Jacobian matrix of F, or at least giving the sparsity pattern of the Jacobian. There are several algorithms to choose from, so read the documentation if you have trouble on a problem. All of the methods expect F to be reasonably smooth, and they benefit from good starting guesses.

8.4 Optimization

To find the minimum of a scalar function $f(x)$ of one variable, use `fminbnd`. It requires a function computing f and an interval on which to optimize. (To maximize $f(x)$, you minimize $-f(x)$.)

For minimization with multiple variables there are two main choices: `fminunc` (in the Optimization Toolbox) and `fminsearch`. For smooth functions `fminunc` is usually faster, but `fminsearch` does not use any derivative information and may be better for nonsmooth problems. In the special case of **nonlinear least squares**,

$$f(x) = f_1(x)^2 + f_2(x)^2 + \cdots f_m(x)^2,$$

one should use `lsqnonlin` instead of either of these. (In fact, this is one of the methods `fsolve` uses for solving systems.)

Optimization in engineering often comes with side constraints on the variables. None of the multidimensional methods named here respects such constraints. See the Optimization Toolbox for methods for several variants of these problems.

8.5 Fitting data

A common task is to make sense of data from noisy observations of a presumably simple relationship. Here "noise" can refer to experimental error or uncertainty, or neglected higher-order

effects. In data fitting one posits a form for the underlying relationship involving some unknown parameters, then uses the data to find the best parameters. This leads to an optimization.

When the form of the relationship is a linear combination of basis functions and the parameters are the coefficients, the resulting optimization is a linear least-squares problem and can be solved using the backslash. For example,

```
t = (1:100)';  y = 4 + 0.1*cos(t) - 0.2*sin(t);
y = y + 0.1*randn(100,1);
A = [ ones(100,1) cos(t) sin(t) ];
A\y

ans =
    4.0048
    0.0938
   -0.1951
```

When the functional form is a polynomial, the entire process is automated by `polyfit`. Also, a graphical data fitting tool for many linear fits is available from the Tools menu of every figure window.

If the coefficients appear nonlinearly, a nonlinear optimization results. For instance, MATLAB ships with U. S. census data for the national population every ten years since 1790. To fit these data with an exponential curve of the form $c_1 + c_2 e^{c_3 t}$, we could proceed as follows:

```
load census
t = (cdate-1790)/10;
residual = @(param) sum( (param(1) + param(2)*exp(param(3)*t) - pop).^2 );
fminsearch(residual,[1 1 1])

ans =
  -39.1124   35.8209    0.1053
```

You can check graphically that this is a pretty good fit. (Note that if we set $c_1 = 0$ in this example, then by taking logs we get a fit to a linear function.)

8.6 Quadrature

Numerical approximation of definite integrals is called **quadrature** in one dimension and **cubature** if the integral is over multiple dimensions. The basic use of `quad` for quadrature was shown in section 4.3. An alternative `quadl` may be faster for high accuracy with smooth integrands. MATLAB offers `dblquad` for cubature over rectangular domains. More general domains can be implemented literally using iterated integration.

In truth, MATLAB's integration offerings fall a bit short in accuracy, reliability, generality, and speed. If you need to do anything beyond some basic one-dimensional integrands, you might do well to look up CUBPACK or QUADPACK on the web.

8.7 Initial-value problems

MATLAB has an excellent library of solvers for the initial value problem (IVP) $x'(t) = f(t, x(t))$, $x(a) = x_0$, where x and f are understood to be vectors. Any higher-order differential equation must first be written in first-order form. The primary division between IVP solvers is **stiff** versus **nonstiff**. Loosely speaking, stiff problems have simultaneous phenomena over very different time scales, and they require methods that are much more expensive for each time step taken, especially in the face of high-dimensional nonlinearity. The logical extreme of stiffness is the **differential–algebraic equation**, which has side constraints on the variables in addition to the dynamics.

Basic ODE solution takes the form

$$\textit{output} = \texttt{ode}\text{xxx}(\ \textit{odefunc}, \textit{time}, \textit{init}, \textit{options}\ )$$

Here the *odefunc* is a function accepting scalar t and vector x, returning the vector $f(t, x)$; *time* is a vector of two or more times spanning the time interval of solution; *init* is the initial condition x_0; *options* controls aspects of the solution behavior, including error tolerances, and is created using `odeset`; and the solver's name is one of `ode45`, `ode113`, `ode15s`, or the other solvers listed in the documentation.

There are two forms for the output. If two names `[t,y]` are given, then `t` is a vector of times and each row of `y` is the solution vector at one of those times. If the *time* input is a vector of length greater than two, then `t` is identical to it; otherwise, `t` is the vector of time steps that were chosen automatically.[26] The other form of output is to give a single variable `soln`, which can be used in `deval` to evaluate the solution at any desired time values.

For example, consider the well-known predator-prey model

$$\frac{dx_1}{dt} = x_1 - \alpha x_1 x_2$$
$$\frac{dx_2}{dt} = \beta x_1 x_2 - x_2.$$

This can be solved by

```
f = @(t,x) [ x(1)-0.2*x(1)*x(2);  0.5*x(1)*x(2) - x(2) ];
[t,x] = ode45(f, 0:0.05:20, [8;1]);
subplot(2,1,1), plot(t,x)
subplot(2,1,2), plot(x(:,1),x(:,2))
```

The IVP solvers offer lots of customization and additional features. Investigate online help if you are doing any more than the basics.

[26] The only difference between the two cases is in the output—the time steps chosen by the algorithm in the solver are unaffected.

8.8 Boundary-value problems

A boundary-value problem also starts with a first-order ordinary differential equation, $y'(x) = f(x, y)$, for $a \le x \le b$, but unlike an initial-value problem, it has side conditions involving two points: $g(y(a), y(b)) = 0$. Existence and uniqueness theory is much less general than for IVPs, and the numerical solution process is less straightforward. Here we just give a simple case study, the Allen–Cahn equation

$$\epsilon y'' + y(1 - y^2) = 0, \qquad 0 \le x \le 1, \qquad y(0) = -1, \quad y(1) = 1.$$

Both $y \equiv -1$ and $y \equiv 1$ are constant solutions, and this equation models a "phase change" between them. If ϵ is small, this change is abrupt.

Three elements are needed to attempt a numerical solution: a function for the ODE, a function for the boundary conditions, and an initial guess to the solution. Although in general this is probably best done in an M-file, here we use anonymous functions. The ODE must be cast in first order form by introducing $y_1 = y$, $y_2 = y'$:

```
f = @(x,y) [ y(2); (y(1)^3-y(1))/0.01 ];
```

Here we set $\epsilon = 0.01$. Next, we create a function that should return all zeros when the boundary conditions are satisfied:

```
g = @(ya,yb) [ ya(1)+1; yb(1)-1 ];
```

Finally, we must create an initial guess to the solution using `bvpinit`. We choose a linear interpolation on equally spaced nodes between the boundary values.

```
x = linspace(0,1,10)';
y0 = @(x) [ -1+2*x; 2 ];
ACinit = bvpinit(x,y0);
```

The initial guess may be crucial to obtaining a solution in difficult problems. Finally, we call `bvp4c` to create the solution and use `deval` to evaluate it for a plot:

```
soln = bvp4c(f,g,ACinit);
xp=0:0.05:1;  plot(xp,deval(soln,xp,1))
```

(The third argument to `deval` asks for just the first component of the solution, the original variable y.) If you inspect the output `soln`, you will see that it is a structure, and that the original 10 nodes in x have been replaced by 35 unequally spaced ones.

If we attempt the same process with $\epsilon = 10^{-3}$, the solver appears to hang. The problem is that the linear initial guess does not give a sufficient clue about the shape of the solution. In this case one gets better results by using the solution with $\epsilon = 10^{-2}$ as the initial guess, a crude form of a process known as **continuation**. The syntax of `bvp4c` makes this easy.

```
soln = bvp4c(f,g,soln)

ans =
         x: [1x63 double]
         y: [2x63 double]
        yp: [2x63 double]
    solver: 'bvp4c'
```

Now we see that 63 nodes are being used, and a plot confirms that the transition region near $x = 1/2$ is much steeper.

8.9 Initial-boundary value problems

An initial-boundary value problem is a time-dependent partial differential equation with side conditions at an initial time and on geometric boundaries. While MATLAB does have an automatic facility for certain PDEs with one space and one time dimension (`pdepe`), here we study a semi-automated process known as semidiscretization or **the method of lines**. In the method of lines ones discretizes space first and then time independently. At a practical level this means the original PDE is replaced by a (typically large) system of ordinary differential equations. In principle, and often in practice too, one can solve the ODE system using the IVP solvers from section 8.7.

For example, consider the problem

$$u_t = u_{xx} + u^2, \quad 0 \le x \le 1, \quad u(t,0) = u(t,1) = 0.$$

This problem mixes blowup in finite time (via $u_t = u^2$) with diffusion. If the initial condition is large enough, blowup wins. If we use a simple second-order finite difference for the u_{xx} term, this problem can be attacked using the remarkably simple script

```
N = 100; h = 1/N;
x = h*(1:N-1)';
D = sparse(toeplitz([-2 1 zeros(1,N-3)]/h^2));
f = @(t,u) D*u + u.^2;
u0 = 30*sin(pi*x);
[t,u] = ode113(f,[0 .1],u0);
```

Upon execution one receives the message

```
Warning: Failure at t=4.561109e-02.  Unable to meet integration
tolerances without reducing the step size below the smallest value
allowed (2.220446e-16) at time t.
```

This is an indication of the finite-time blowup. By increasing N or using more accurate finite differences, one could quickly determine that at least two digits of the blowup time have apparently been found accurately.