MATLAB Tutorial

to accompany

Partial Differential Equations: Analytical and Numerical Methods, 2nd edition

by

Mark S. Gockenbach

(SIAM, 2010)

MATLAB Tutorial **1**
- Introduction 3
  - About this tutorial 3
  - About MATLAB 3
  - MATLAB M-Book 3
  - Getting help with MATLAB commands 4
  - Getting started with MATLAB 4
  - Vectors and matrices in MATLAB 8
  - More about M-Books 9
  - Simple graphics in MATLAB 9
  - Symbolic computation in MATLAB 15
  - Manipulating functions in MATLAB 18
  - Saving your MATLAB session 20
  - About the rest of this tutorial 20

Chapter 1: Classification of differential equations **21**

Chapter 2: Models in one dimension **24**
- Section 2.1: Heat flow in a bar; Fourier's Law 24
  - Solving simple boundary value problems by integration 25
  - The MATLAB solve command 26

Chapter 3: Essential linear algebra **31**
- Section 3.1 Linear systems as linear operator equations 31
- Section 3.2: Existence and uniqueness of solutions to Ax=b 32
- Section 3.3: Basis and dimension 35
  - Symbolic linear algebra 37
  - Programming in MATLAB, part I 38
  - Defining a MATLAB function in an M-file. 38
  - Optional inputs with default values 43
  - Comments in M-files 44
  - M-files as scripts 44
- Section 3.4: Orthogonal bases and projection 46
  - Working with the L^2 inner product 46
- Section 3.5: Eigenvalues and eigenvectors of a symmetric matrix 53
  - Numerical eigenvalues and eigenvectors 53
  - Symbolic eigenvalues and eigenvectors 53
  - Review: Functions in MATLAB 56

Chapter 4: Essential ordinary differential equations **59**
- Section 4.2: Solutions to some simple ODEs 59
  - Second-order linear homogeneous ODEs with constant coefficients 60
  - A special inhomogeneous second-order linear ODE 61
  - First-order linear ODEs 62
- Section 4.3: Linear systems with constant coefficients 64
  - Inhomogeneous systems and variation of parameters 66
  - Programming in MATLAB, Part II 68
  - Conditional execution 69
  - Passing one function into another 70

Section 4.4: Numerical methods for initial value problems 72
Programming in MATLAB, part III 78
Efficient MATLAB programming 81
More about graphics in MATLAB 81
Chapter 5: Boundary value problems in statics 83
Section 5.2: Introduction to the spectral method; eigenfunctions 83
Section 5.5: The Galerkin method 88
Computing the stiffness matrix and load vector in loops 92
Section 5.6: Piecewise polynomials and the finite element method 93
Computing the load vector 95
Computing the stiffness matrix 96
The nonconstant coefficient case 99
Creating a piecewise linear function from the nodal values 100
More about sparse matrices 101
Chapter 6: Heat flow and diffusion 104
Section 6.1: Fourier series methods for the heat equation 104
Section 6.4: Finite element methods for the heat equation 107
Chapter 8: First-Order PDEs and the Method of Characteristics 110
Section 8.1: The simplest PDE and the method of characteristics 110
Two-dimensional graphics in MATLAB 110
Section 8.2: First-order quasi-linear PDEs 113
Chapter 11: Problems in multiple spatial dimensions 114
Section 11.2: Fourier series on a rectangular domain 114
Section 8.3: Fourier series on a disk 117
Graphics on the disk 118
Chapter 12: More about Fourier series 121
Section 12.1: The complex Fourier series 121
Section 9.2: Fourier series and the FFT 123
Chapter 13: More about finite element methods 125
Section 13.1 Implementation of finite element methods 125
Creating a mesh 125
Computing the stiffness matrix and the load vector 128
Testing the code 132
Using the code 136

Introduction

In this introduction, I will explain the organization of this tutorial and give some basic information about MATLAB and MATLAB notebooks. I will also give a preliminary introduction to the capabilities of MATLAB.

About this tutorial

The purpose of this document is to explain the features of MATLAB that are useful for applying the techniques presented in my textbook. This really is a tutorial (not a reference), meant to be read and used in parallel with the textbook. For this reason, I have structured the tutorial to have the same chapter and sections titles as the book. However, the purpose of the sections of this document is not to re-explain the material in the text; rather, it is to present the capabilities of MATLAB as they are needed by someone studying the text.

Therefore, for example, in Section 2.1, "Heat flow in a bar; Fourier's Law", I do not explain any physics or modeling. (The physics and modeling are found in the text.) Instead, I explain the MATLAB command for integration, because Section 2.1 is the first place in the text where the student is asked to integrate a function. Because of this style of organization, some parts of the text have no counterpart in this tutorial. For example, there is no Chapter 7, because, by the time you have worked through the first six chapters of the tutorial, you have learned all of the capabilities of MATLAB that you need to address the material in Chapter 7 of the text. For the same reason, you will see that some individual sections are missing; Chapter 5, for example, begins with Section 5.2.

I should point out that my purpose is writing this tutorial is not to show you how to solve the problems in the text; rather, it is to give you the tools to solve them. Therefore, I do not give you a worked-out example of every problem type---if I did, your "studying" could degenerate to simply looking for an example, copying it, and making a few changes. At crucial points, I do provide some complete examples, since I see no other way to illustrate the power of MATLAB than in context. However, there is still plenty for you to figure out for yourself!

About MATLAB

MATLAB, which is short for Matrix Laboratory, incorporates numerical computation, symbolic computation, graphics, and programming. As the name suggests, it is particularly oriented towards matrix computations, and it provides both state-of-the-art algorithms and a simple, easy to learn interface for manipulating matrices. In this tutorial, I will touch on all of the capabilities mentioned above: numerical and symbolic computation, graphics, and programming.

MATLAB M-Book

This document you are reading is called an M-Book. It integrates text and MATLAB commands (with their output, including graphics). If you are running MATLAB under Microsoft Windows, then an M-Book becomes an interactive document: by running the M-Book under MATLAB, you can enter new MATLAB commands and see their output inside the M-Book itself. The MATLAB command that allows you to do this is called **notebook**. To run this tutorial under MATLAB, just type "notebook tutorial.docx" at the MATLAB prompt. The file tutorial.docx must be in the working directory or in some directory in the MATLAB path (both of these concepts are explained below.)

Since the M-Book facility is available only under Microsoft Windows, I will not emphasize it in this tutorial. However, Windows users should take advantage of it. The most important thing to understand about a M-Book is that it is interactive---at any time you can execute a MATLAB command and see what it does. This makes a MATLAB M-Book a powerful learning environment: when you read an explanation of a MATLAB feature, you can immediately try it out.

Getting help with MATLAB commands

Documentation about MATLAB and MATLAB commands is available from within the program itself. If you know the name of the command and need more information about how it works, you can just type "**help** <command name>" at the MATLAB prompt. In the same way, you can get information about a group of commands with common uses by typing "**help** <topic name>". I will show examples of using the command-line help feature below.

The MATLAB desktop contains a help browser covering both reference and tutorial material. To access the browser, click on the **Help** menu and choose **MATLAB Help**. You can then choose "**Getting Started**" from the table of contents for a tutorial introduction to MATLAB, or use the index to find specific information.

Getting started with MATLAB

As mentioned above, MATLAB has many capabilities, such as the fact that one can write programs made up of MATLAB commands. The simplest way to use MATLAB, though, is as an interactive computing environment (essentially, a very fancy graphing calculator). You enter a command and MATLAB executes it and returns the result. Here is an example:

```
clear
2+2
ans =
     4
```

You can assign values to variables for later use:

```
x=2
x =
     2
```

The variable **x** can now be used in future calculations:

```
x^2
ans =
     4
```

At any time, you can list the variables that are defined with the **who** command:

```
who

Your variables are:

ans   x
```

At the current time, there are 2 variables defined. One is **x**, which I explicitly defined above. The other is **ans** (short for "answer"), which automatically holds the most recent result that was *not* assigned to a variable (you may have noticed how **ans** appeared after the first command above). You can always check the value of a variable simply by typing it:

```
x
x =
     2

ans
ans =
     4
```

If you enter a variable that has not been defined, MATLAB prints an error message:

```
y
??? Undefined function or variable 'y'.
```

To clear a variable from the workspace, use the **clear** command:

```
who

Your variables are:

ans  x

clear x

who

Your variables are:

ans
```

To clear of the variables from the workspace, just use **clear** by itself:

```
clear
who
```

MATLAB knows the elementary mathematical functions: trigonometric functions, exponentials, logarithms, square root, and so forth. Here are some examples:

```
sqrt(2)
ans =
    1.4142

sin(pi/3)
ans =
    0.8660


exp(1)
ans =
    2.7183

log(ans)
```

```
ans =
     1
```

A couple of remarks about the above examples:

- MATLAB knows the number π, which is called **pi**.
- Computations in MATLAB are done in floating point arithmetic by default. For example, MATLAB computes the sine of π/3 to be (approximately) 0.8660 instead of exactly √3/2.

A complete list of the elementary functions can be obtained by entering "**help elfun**":

```
help elfun
  Elementary math functions.

  Trigonometric.
    sin         - Sine.
    sind        - Sine of argument in degrees.
    sinh        - Hyperbolic sine.
    asin        - Inverse sine.
    asind       - Inverse sine, result in degrees.
    asinh       - Inverse hyperbolic sine.
    cos         - Cosine.
    cosd        - Cosine of argument in degrees.
    cosh        - Hyperbolic cosine.
    acos        - Inverse cosine.
    acosd       - Inverse cosine, result in degrees.
    acosh       - Inverse hyperbolic cosine.
    tan         - Tangent.
    tand        - Tangent of argument in degrees.
    tanh        - Hyperbolic tangent.
    atan        - Inverse tangent.
    atand       - Inverse tangent, result in degrees.
    atan2       - Four quadrant inverse tangent.
    atanh       - Inverse hyperbolic tangent.
    sec         - Secant.
    secd        - Secant of argument in degrees.
    sech        - Hyperbolic secant.
    asec        - Inverse secant.
    asecd       - Inverse secant, result in degrees.
    asech       - Inverse hyperbolic secant.
    csc         - Cosecant.
    cscd        - Cosecant of argument in degrees.
    csch        - Hyperbolic cosecant.
    acsc        - Inverse cosecant.
    acscd       - Inverse cosecant, result in degrees.
    acsch       - Inverse hyperbolic cosecant.
    cot         - Cotangent.
    cotd        - Cotangent of argument in degrees.
    coth        - Hyperbolic cotangent.
    acot        - Inverse cotangent.
    acotd       - Inverse cotangent, result in degrees.
    acoth       - Inverse hyperbolic cotangent.
    hypot       - Square root of sum of squares.

  Exponential.
    exp         - Exponential.
    expm1       - Compute exp(x)-1 accurately.
    log         - Natural logarithm.
    log1p       - Compute log(1+x) accurately.
    log10       - Common (base 10) logarithm.
    log2        - Base 2 logarithm and dissect floating point number.
    pow2        - Base 2 power and scale floating point number.
    realpow     - Power that will error out on complex result.
    reallog     - Natural logarithm of real number.
    realsqrt    - Square root of number greater than or equal to zero.
    sqrt        - Square root.
    nthroot     - Real n-th root of real numbers.
    nextpow2    - Next higher power of 2.

  Complex.
    abs         - Absolute value.
    angle       - Phase angle.
    complex     - Construct complex data from real and imaginary parts.
    conj        - Complex conjugate.
    imag        - Complex imaginary part.
    real        - Complex real part.
    unwrap      - Unwrap phase angle.
    isreal      - True for real array.
    cplxpair    - Sort numbers into complex conjugate pairs.

  Rounding and remainder.
    fix         - Round towards zero.
    floor       - Round towards minus infinity.
    ceil        - Round towards plus infinity.
    round       - Round towards nearest integer.
    mod         - Modulus (signed remainder after division).
    rem         - Remainder after division.
    sign        - Signum.
```

For more information about any of these elementary functions, type "**help** <function_name>". For a list of help topics like "elfun", just type "**help**". There are other commands that form part of the help system; to see them, type "**help help**".

MATLAB does floating point arithmetic using the IEEE standard, which means that numbers have about 16 decimal digits of precision (the actual representation is in binary, so the precision is not exactly 16 digits). However, MATLAB only displays 5 digits by default. To change the display, use the **format** command. For example, "**format long**" changes the display to 15 digits:

```
format long

pi
ans =
   3.141592653589793
```

Other options for the **format** command are "**format short e**" (scientific notation with 5 digits) and "**format long e**" (scientific notation with 15 digits).

In addition to **pi**, other predefined variables in MATLAB include **i** and **j**, both of which represent the imaginary unit: **i=j=sqrt(-1)**.

```
clear
i^2
ans =
    -1
```

```
j^2
ans =
    -1
```

Although it is usual, in mathematical notation, to use **i** and **j** as arbitrary indices, this can sometimes lead to errors in MATLAB because these symbols are predefined. For this reason, I will use **ii** and **jj** as my standard indices when needed.

Vectors and matrices in MATLAB

The default type for any variable or quantity in MATLAB is a matrix---a two-dimensional array. Scalars and vectors are regarded as special cases of matrices. A scalar is a 1 by 1matrix, while a vector is an *n* by 1 or 1 by *n* matrix. A matrix is entered by rows, with entries in a row separated by spaces or commas, and the rows separated by semicolons. The entire matrix is enclosed in square brackets. For example, I can enter a 3 by 2 matrix as follows:

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

Here is how I would enter a 2 by 1 (column) vector:

```
x=[1;-1]
x =
     1
    -1
```

A scalar, as we have seen above, requires no brackets:

```
a=4
a =
     4
```

A variation of the **who** command, called **whos**, gives more information about the defined variables:

```
whos
  Name      Size            Bytes  Class     Attributes

  A         3x2                48  double
  a         1x1                 8  double
  ans       1x1                 8  double
  x         2x1                16  double
```

The column labeled "size" gives the size of each array; you should notice that, as I mentioned above, a scalar is regarded as a 1 by 1 matrix (see the entry for **a**, for example).

MATLAB can perform the usual matrix arithmetic. Here is a matrix-vector product:

```
A*x
ans =
    -1
    -1
    -1
```

Here is a matrix-matrix product:

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

A*B
ans =
     3     3     6     2
     5     9    16    10
     7    15    26    18
```

MATLAB signals an error if you attempt an operation that is undefined:

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


A+B
??? Error using ==> plus
Matrix dimensions must agree.
```

More about M-Books

If you are reading this document using the MATLAB notebook facility, then you may wish to execute the commands as you read them. Otherwise, the variables shown in the examples are not actually created in the MATLAB workspace. To execute a command, click on it (or select it) and press control-enter (that is, press the enter key while holding down the control key). While reading the tutorial, you should execute each of my commands as you come to it. Otherwise, the state of MATLAB is not what it appears to be, and if you try to experiment by entering your own commands, you might get unexpected results if your calculations depend on the ones you see in this document.

Notice that the command lines in this document appear in green, and are enclosed in gray square brackets. Output appears in blue text, also enclosed in gray square brackets. These comments may not apply if you are reading a version of this document that has been printed or converted to another format (such as or PDF).

If you are reading this using MATLAB's notebook command, then, as I mentioned above, you can try your own MATLAB commands at any time. Just move the cursor to a new line, type the command, and then type control-enter. You should take advantage of this facility, as it will make learning MATLAB much easier.

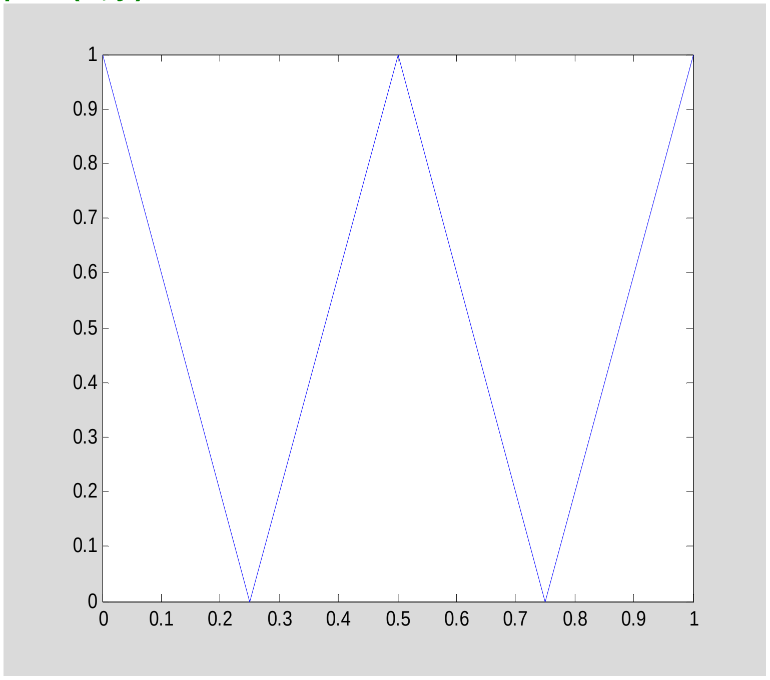
Simple graphics in MATLAB

Two-dimensional graphics are particularly easy to understand: If you define vectors **x** and **y** of equal length (each with n components, say), then MATLAB's **plot** command will graph the points (x_1,y_1), (x_2,y_2), …, (x_n,y_n) in the plane and connect them with line segments. Here is an example:

```
format short
x=[0,0.25,0.5,0.75,1]
x =
         0    0.2500    0.5000    0.7500    1.0000
```

```
y=[1,0,1,0,1]
y =
     1     0     1     0     1
```

```
plot(x,y)
```



Two features of MATLAB make it easy to generate graphs. First of all, the command **linspace** creates a vector with linearly spaced components---essentially, a regular grid on an interval. (Mathematically, **linspace** creates a finite arithmetic sequence.) To be specific, **linspace(a,b,n)** creates the (row) vector whose components are $a,a+h,a+2h,\ldots,a+(n-1)h$, where $h=1/(n-1)$.

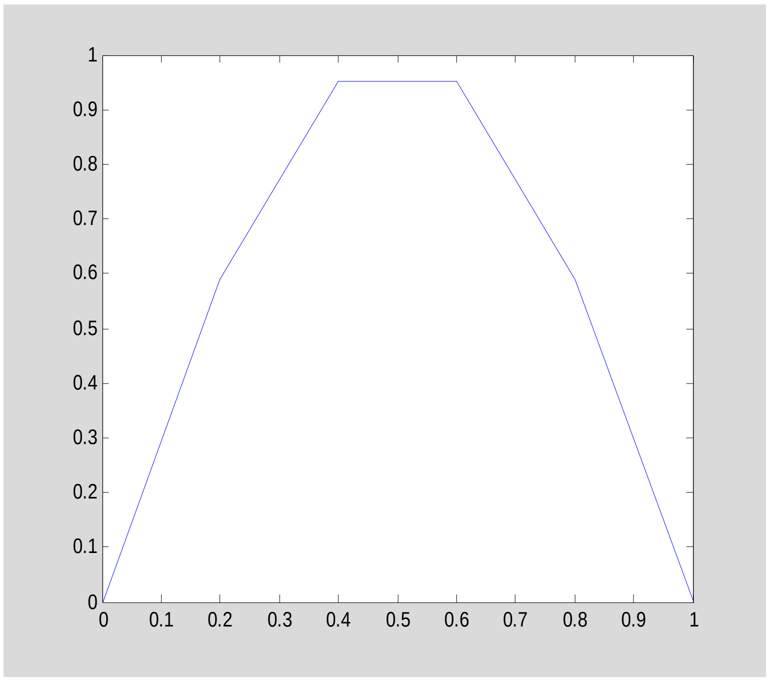
```
x=linspace(0,1,6)
x =
         0    0.2000    0.4000    0.6000    0.8000    1.0000
```

The second feature that makes it easy to create graphs is the fact that all standard functions in MATLAB, such as sine, cosine, exp, and so forth, are *vectorized*. A vectorized function **f**, when applied to a vector **x**, returns a vector **y** (of the same size as **x**) with ith component equal to $f(x_i)$. Here is an example:

```
y=sin(pi*x)
y =
         0    0.5878    0.9511    0.9511    0.5878    0.0000
```

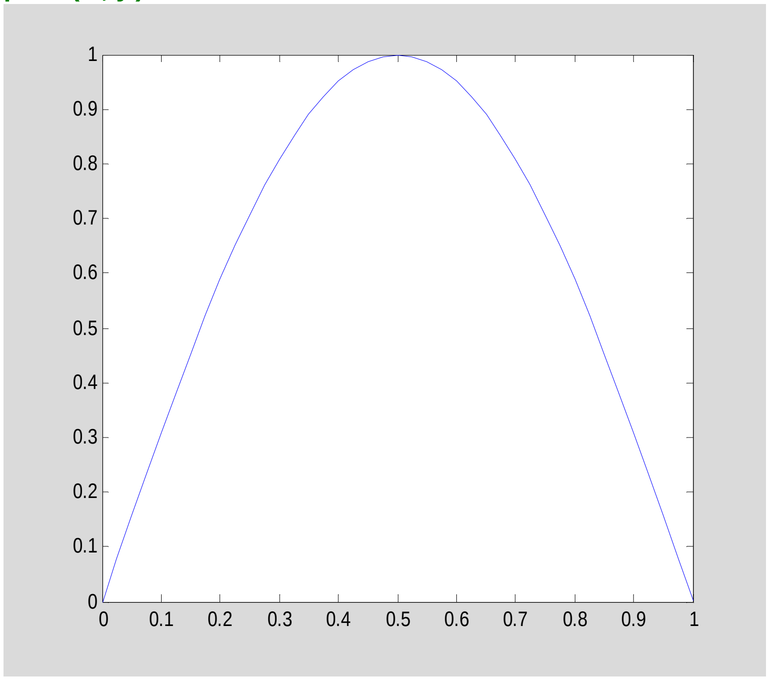
I can now plot the function:

```
plot(x,y)
```



Of course, this is not a very good plot of sin(π*x*), since the grid is too coarse. Below I create a finer grid and thereby obtain a better graph of the function. Often when I create a new vector or matrix, I do not want MATLAB to display it on the screen. (The following example is a case in point: I do not need to see the 41 components of vector **x** or vector **y**.) When a MATLAB command is followed by a semicolon, MATLAB will not display the output.

```
x=linspace(0,1,41);
y=sin(pi*x);
plot(x,y)
```



The basic arithmetic operations have vectorized versions in MATLAB. For example, I can multiply two vectors component-by-component using the ".*" operator. That is, **z**=**x**.***y** sets z_i equal to x_iy_i. Here is an example:

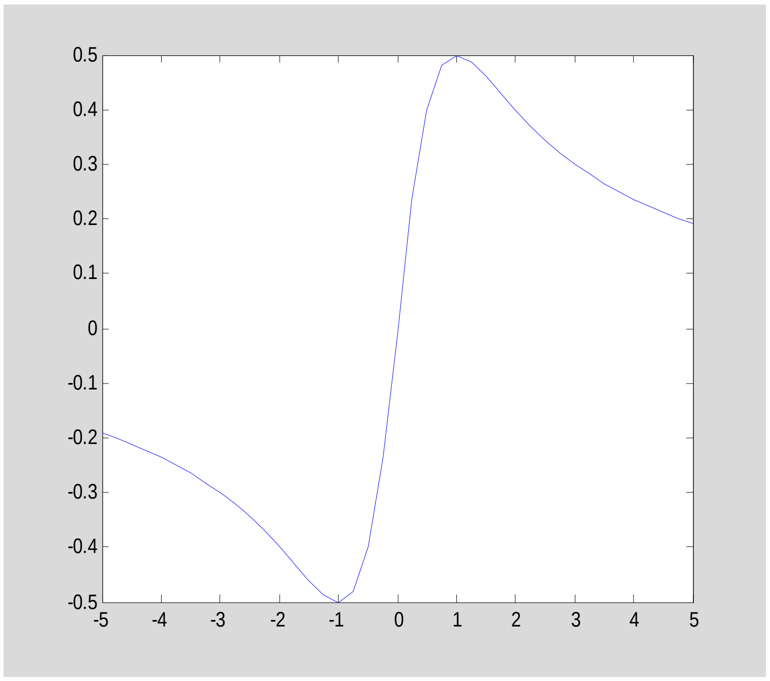
```
x=[1;2;3]
x =
     1
     2
     3

y=[2;4;6]
y =
     2
     4
     6

z=x.*y
z =
     2
     8
    18
```

The "./" operator works in an analogous fashion. There are no ".+" or ".-" operators, since addition and subtraction of vectors are defined componentwise already. However, there is a ".^" operator that applies an exponent to each component of a vector.

```
x
x =
     1
     2
     3

x.^2
ans =
     1
     4
     9
```

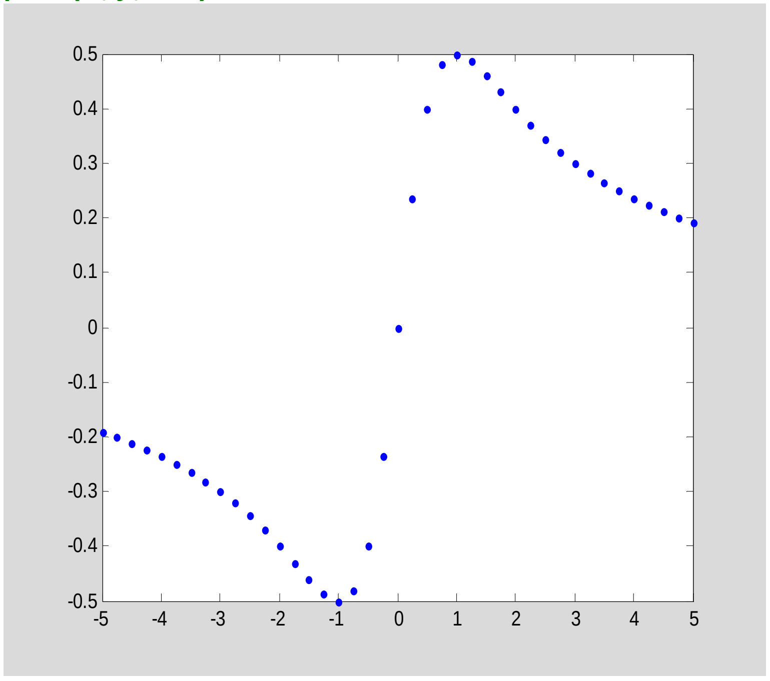
Finally, scalar addition is automatically vectorized in the sense that **a**+**x**, where **a** is a scalar and **x** is a vector, adds **a** to every component of **x**.
The vectorized operators make it easy to graph a function such as $f(x)=x/(1+x^2)$. Here is how it is done:

```
x=linspace(-5,5,41);
y=x./(1+x.^2);
plot(x,y)
```

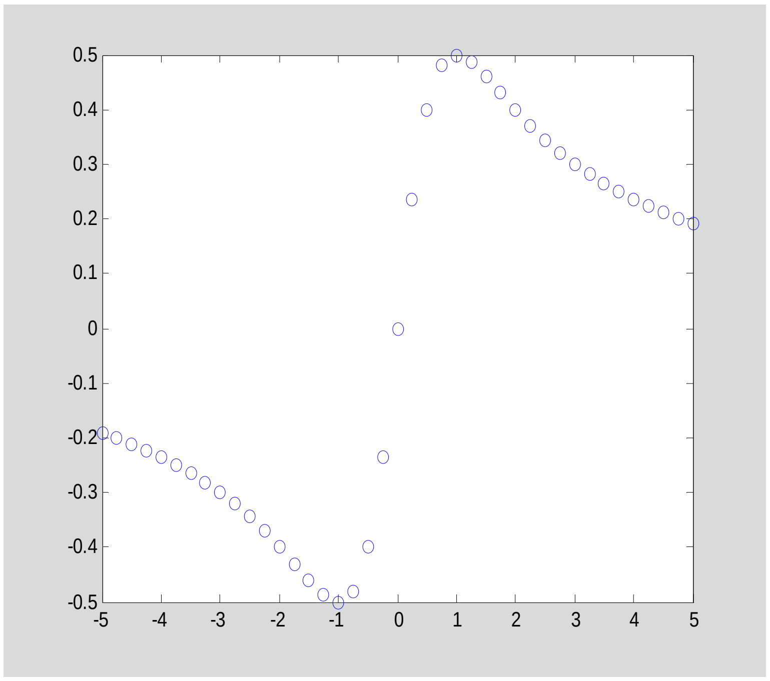


If I prefer, I can just graph the points themselves, or connect them with dashed line segments. Here are examples:

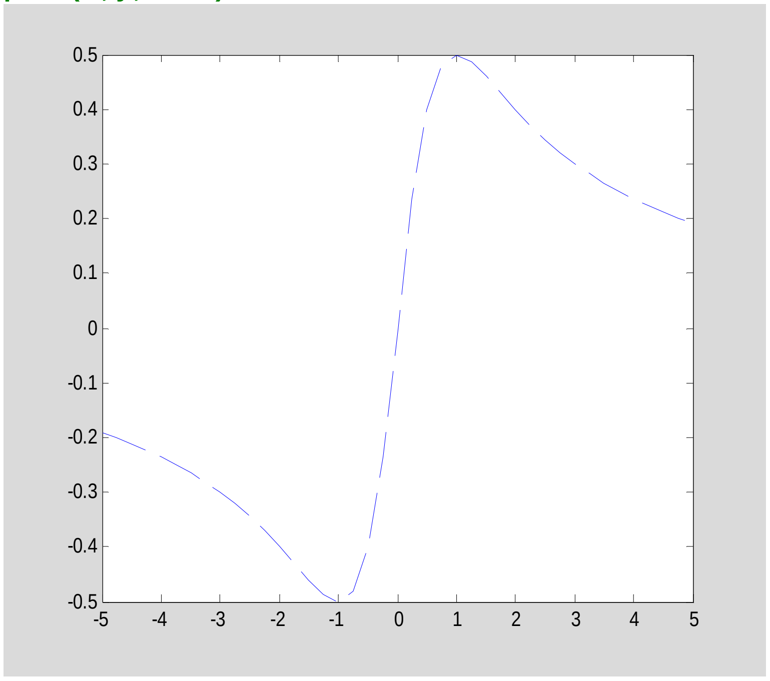
```
plot(x,y,'.')
```



```
plot(x,y,'o')
```



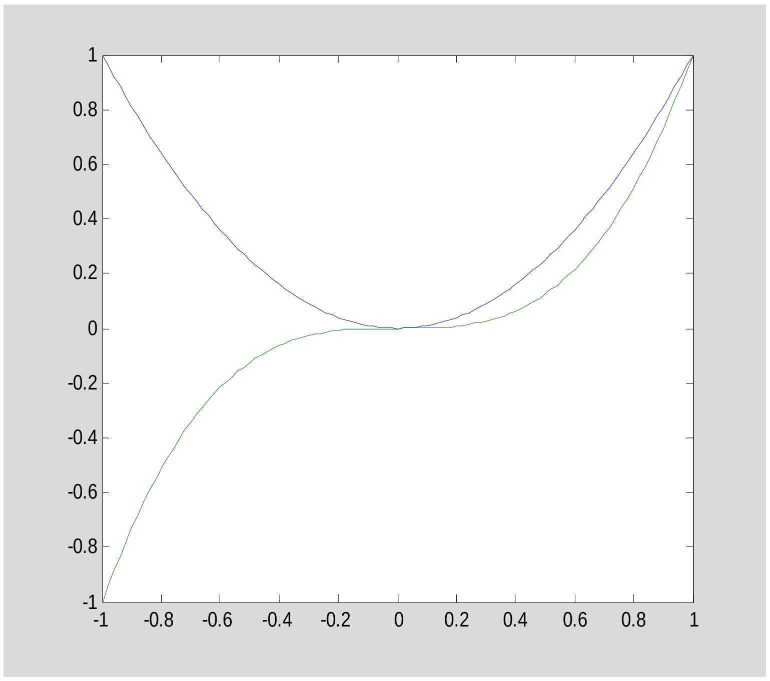
```
plot(x,y,'--')
```



The string following the vectors **x** and **y** in the plot command ('.', 'o', and '—' in the above examples) specifies the line type. it is also possible to specify the color of the graph. For more details, see "**help plot**".

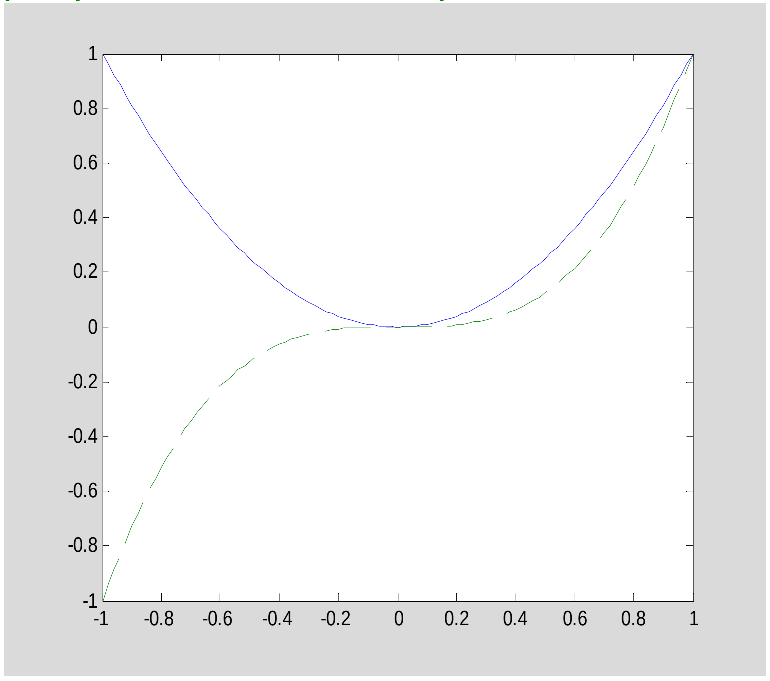
It is not much harder to plot two curves on the same graph. For example, I plot y=x^2 and y=x^3 together:

```
x=linspace(-1,1,101);
plot(x,x.^2,x,x.^3)
```



I can also give the lines different linetypes:

```
plot(x,x.^2,'-',x,x.^3,'--')
```



Symbolic computation in MATLAB

In addition to numerical computation, MATLAB can also perform symbolic computations. However, since, by default, variables are floating point, you must explicitly indicate that a variable is intended to be symbolic. One way to do this is using the **syms** command, which tells MATLAB that one or more variables are symbolic. For example, the following command defines **a** and **b** to be symbolic variables:

```
syms a b
```

I can now form symbolic expressions involving these variables:

```
2*a*b
ans =
2*a*b
```

Notice how the result is symbolic, not numeric as it would be if the variables were floating point variables. Also, the above calculation does not result in an error, even though **a** and **b** do not have values.

Another way to create a symbolic variable is to assign a symbolic value to a new symbol. Since numbers are, by default, floating point, it is necessary to use the sym function to tell MATLAB that a number is symbolic:

```
c=sym(2)
c =
2

whos
  Name       Size                Bytes  Class     Attributes

  A          3x2                    48  double
  B          2x4                    64  double
  a          1x1                    60  sym
  ans        1x1                    60  sym
  b          1x1                    60  sym
  c          1x1                    60  sym
  x          1x101                 808  double
  y          1x41                  328  double
  z          3x1                    24  double
```

I can do symbolic computations:

```
a=sqrt(c)
a =
2^(1/2)
```

You should notice the difference between the above result and the following:

```
a=sqrt(2)
a =
    1.4142

whos
  Name       Size                Bytes  Class     Attributes

  A          3x2                    48  double
  B          2x4                    64  double
  a          1x1                     8  double
  ans        1x1                    60  sym
  b          1x1                    60  sym
  c          1x1                    60  sym
  x          1x101                 808  double
  y          1x41                  328  double
  z          3x1                  24  double
```

Even though a was declared to be a symbolic variable, once I assign a floating point value to it, it becomes numeric. This example also emphasizes that **sym** must be used with literal constants if they are to interpreted as symbolic and not numeric:

```
a=sqrt(sym(2))
a =
2^(1/2)
```

As a further elementary example, consider the following two commands:

```
sin(sym(pi))
ans =
0

sin(pi)
ans =
  1.2246e-016
```

In the first case, since π is symbolic, MATLAB notices that the result is *exactly* zero; in the second, both π and the result are represented in floating point, so the result is not exactly zero (the error is just roundoff error).

Using symbolic variables, I can perform algebraic manipulations.

```
syms  x
p=(x-1)*(x-2)*(x-3)
p =
(x - 1)*(x - 2)*(x - 3)

expand(p)
ans =
x^3 - 6*x^2 + 11*x - 6

factor(ans)
ans =
(x - 3)*(x - 1)*(x - 2)
```

Integers can also be factored, though the form of the result is depends on whether the input is numeric or symbolic:

```
factor(144)
ans =
     2     2     2     2     3     3
factor(sym(144))
ans =
2^4*3^2
```

For a numeric (integer) input, the result is a list of the factors, repeated according to multiplicity. For a symbolic input, the output is a symbolic expression.

An important command for working with symbolic expressions is **simplify**, which tries to reduce an expression to a simpler one equal to the original. Here is an example:

```
clear
```

```
syms x a b c
p=(x-1)*(a*x^2+b*x+c)+x^2+3*x+a*x^2+b*x
p =
3*x + b*x + a*x^2 + (x - 1)*(a*x^2 + b*x + c) + x^2
```

```
simplify(p)
ans =
3*x - c + c*x + a*x^3 + b*x^2 + x^2
```

Since the concept of simplification is not precisely defined (which is simpler, a polynomial in factored form or in expanded form $a+bx+cx^2+\ldots$?), MATLAB has a number of specialized simplification commands. I have already used two of them, **factor** and **expand**. Another is **collect**, which "gathers like terms":

```
p
p =
3*x + b*x + a*x^2 + (x - 1)*(a*x^2 + b*x + c) + x^2
```

```
collect(p,x)
ans =
a*x^3 + (b + 1)*x^2 + (c + 3)*x - c
```

By the way, the display of symbolic output can be made more mathematical using the **pretty** command:

```
pretty(ans)

     3             2
  a x  + (b + 1) x  + (c + 3) x - c
```

Note: MATLAB's symbolic computation is based on Maple ™, a computer algebra system originally developed at the University of Waterloo, Canada, and marketed by Waterloo Maple, Inc. If you have the Extended Symbolic Math Toolbox with your installation of MATLAB, then you have access to *all* nongraphical Maple functions; see "help maple" for more details. However, these capabilities are not included in the standard Student Version of MATLAB, and so I will not emphasize them in this tutorial.

Manipulating functions in MATLAB

Symbolic expressions can be treated as functions. Here is an example:

```
syms x
p=x/(1+x^2)
p =
x/(x^2 + 1)
```

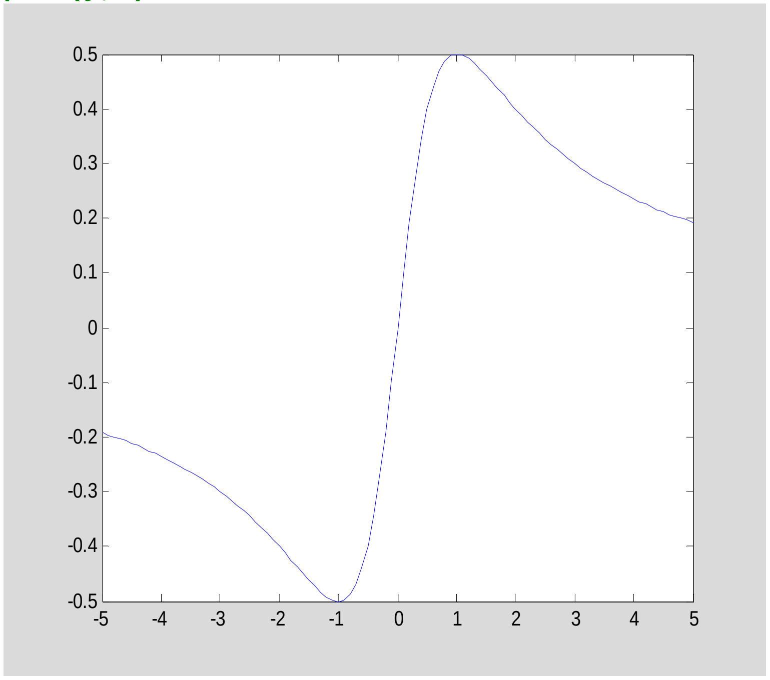
Using the **subs** command, I can evaluate the function **p** for a given value of **x**. The following command substitutes 3 for every occurrence of **x** in **p**:

```
subs(p,x,3)
ans =
    0.3000
```

The calculation is automatically vectorized:

```
y=linspace(-5,5,101);
z=subs(p,x,y);
```

```
plot(y,z)
```



One of the most powerful features of symbolic computation in MATLAB is that certain calculus operations, notably integration and differentiation, can be performed symbolically. These capabilities will be explained later in the tutorial.

Above I showed how to evaluate a function defined by a symbolic expression. It is also possible to explicitly define a function at the command line. Here is an example:

```
f=@(x)x^2
f =
    @(x)x^2
```

The function **f** can be evaluated in the expected fashion, and the input can be either floating point or symbolic:

```
f(1)
ans =
     1
f(sqrt(pi))
ans =
    3.1416

f(sqrt(sym(pi)))
ans =
pi
```

The @ operator can also be used to create a function of several variables:

```
g=@(x,y)x^2*y
g =
    @(x,y)x^2*y
g(1,2)
ans =
     2
g(pi,14)
ans =
  138.1745
```

A function defined by the @ operator is not automatically vectorized:

```
x=linspace(0,1,11)';
y=linspace(0,1,11)';
g(x,y)
??? Error using ==> mpower
Inputs must be a scalar and a square matrix.
To compute elementwise POWER, use POWER (.^) instead.

Error in ==> @(x,y)x^2*y
```

The function can be vectorized when it is defined:

```
g=@(x,y)x.^2.*y
g =
    @(x,y)x.^2.*y
g(x,y)
ans =
         0
    0.0010
    0.0080
    0.0270
    0.0640
    0.1250
    0.2160
    0.3430
    0.5120
    0.7290
    1.0000
```

There is a third way to define a function in MATLAB, namely, to write a program that evaluates the function. I will defer an explanation of this technique until Chapter 3, where I first discuss programming in MATLAB.

Saving your MATLAB session

When using MATLAB, you will frequently wish to end your session and return to it later. Using the **save** command, you can save all of the variables in the MATLAB workspace to a file. The variables are stored efficiently in a binary format to a file with a ".mat" extension. The next time you start MATLAB, you can load the data using the **load** command. See "help save" and "help load" for details.

About the rest of this tutorial

The remainder of this tutorial is organized in parallel with my textbook. Each section in the tutorial introduces any new MATLAB commands that would be useful in addressing the material in the corresponding section of the textbook. As I mentioned above, some sections of the textbook have no counterpart in the tutorial, since all of the necessary MATLAB commands have already been explained. For this reason, the tutorial is intended to be read from beginning to end, in conjunction with the textbook.

Chapter 1: Classification of differential equations

As I mentioned above, MATLAB can perform symbolic calculus on expressions. Consider the following example:

```
syms x
f=sin(x^2)
f =
sin(x^2)
```

I can differentiate this expression using the **diff** command:

```
diff(f,x)
ans =
2*x*cos(x^2)
```

The same techniques work with expressions involving two or more variables:

```
syms x y
q=x^2*y^3*exp(x)
q =
x^2*y^3*exp(x)
pretty(q)

   2  3
  x  y  exp(x)
diff(q,y)
ans =
3*x^2*y^2*exp(x)
```

Thus MATLAB can compute partial derivatives just as easily as ordinary derivatives.

One use of these capabilities to test whether a certain function is a solution of a given differential equation. For example, suppose I want to check whether the function $u(t)=e^{at}$ is a solution of the ODE

$$\frac{du}{dt} - au = 0.$$

I define

```
syms a t
u=exp(a*t)
u =
exp(a*t)
```

I can then compute the left side of the differential equation, and see if it agrees with the right side (zero):

```
diff(u,t)-a*u
ans =
0
```

Thus the given function u is a solution. Is the function $v(t)=at$ another solution? I can check it as follows:

```
v=a*t
v =
a*t
diff(v,t)-a*v
ans =
a - a^2*t
```

Since the result is not zero, the function v is not a solution.

It is no more difficult to check whether a function of several variables is the solution of a PDE. For example, is $w(x,y)=\sin(\pi x)+\sin(\pi y)$ a solution of the differential equation

$$\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}=0?$$

As before, I can answer this question by defining the function and substituting it into the differential equation:

```
syms x y
w=sin(pi*x)+sin(pi*y)
w =
sin(pi*x) + sin(pi*y)
diff(w,x,2)+diff(w,y,2)
ans =
- pi^2*sin(pi*x) - pi^2*sin(pi*y)
simplify(ans)
ans =
-pi^2*(sin(pi*x) + sin(pi*y))
```

Since the result is not zero, the function w is not a solution of the PDE.

The above example shows how to compute higher derivatives of an expression. For example, here is the fifth derivative of w with respect to x:

```
diff(w,x,5)
ans =
pi^5*cos(pi*x)
```

To compute a mixed partial derivative, we have to iterate the **diff** command. Here is the mixed partial derivative of $w(x,y)=x^2+xy^2$ with respect to x and then y:

```
syms x y
w=x^2*exp(y)+x*y^2
w =
x^2*exp(y) + x*y^2

diff(diff(w,x),y)
ans =
2*y + 2*x*exp(y)
```

Instead of using expressions in the above calculations, I can use functions. Consider the following:

```
clear
syms a x
f=@(x)exp(a*x)
f =
    @(x)exp(a*x)
```

```
diff(f(x),x)-a*f(x)
ans =
0
```

When defining a function that depends on a parameter (like the function **f** above, which depends on **a**), the value of the parametered is "captured" at the time when the function is defined.

```
clear
syms a
f=@(x)exp(a*x)
f =
    @(x)exp(a*x)

f(1)
ans =
exp(a)
a=1.5
a =
    1.5000
f(1)
ans =
exp(a)
a
a =
    1.5000
```

The same is true if the parameter has a numeric value:

```
clear
a=2
a =
     2
f=@(x)exp(a*x)
f =
    @(x)exp(a*x)
f(1)
ans =
    7.3891
a=3
a =
     3
f(1)
ans =
    7.3891
```

Chapter 2: Models in one dimension

Section 2.1: Heat flow in a bar; Fourier's Law

MATLAB can compute both indefinite and definite integrals. The command for computing an indefinite integral (antiderivative) is exactly analogous to that for computing a derivative:

```
syms x
f=x^2
f =
x^2
int(f,x)
ans =
x^3/3
```

As this example shows, MATLAB does not add the "constant of integration." It simply returns one antiderivative (when possible). If the integrand is too complicated, MATLAB just returns the integral unevaluated, and prints a warning message.

```
int(exp(cos(x)),x)
Warning: Explicit integral could not be found.
ans =
int(exp(cos(x)), x)
```

To compute a definite integral, we must specify the limits of integration:

```
int(x^2,x,0,1)
ans =
1/3
```

MATLAB has commands for estimating the value of a definite integral that cannot be computed analytically. Consider the following example:

```
int(exp(cos(x)),x,0,1)
Warning: Explicit integral could not be found.
ans =
int(exp(cos(x)), x = 0..1)
```

Since MATLAB could not find an explicit antiderivative, I can use the **quad** function to estimate the definite integral. The **quad** command takes, as input, a function rather than an expression (as does **int**). Therefore, I must first create a function:

```
f=@(x)exp(cos(x))
f =
    @(x)exp(cos(x))
```

Now I can invoke **quad**:

```
quad(f,0,1)
ans =
    2.3416
```

"quad" is short for quadrature, another term for numerical integration. For more information about the **quad** command, see "**help quad**".

As a further example of symbolic calculus, I will use the commands for integration and differentiation to test Theorem 2.1 from the text. The theorem states that (under certain conditions) a partial derivative can be moved under an integral sign:

$$\frac{d}{dx}\int_c^d F(x,y)\,dy = \int_c^d \frac{\partial F}{\partial x}(x,y)\,dy.$$

Here is a specific instance of the theorem:

```
syms  x y c d
f=x*y^3+x^2*y
f =
x^2*y + x*y^3

r1=diff(int(f,y,c,d),x)
r1 =
- (x*(c^2 - d^2))/2 - ((c^2 - d^2)*(c^2 + d^2 + 2*x))/4

r2=int(diff(f,x),y,c,d)
r2 =
-((c^2 - d^2)*(c^2 + d^2 + 4*x))/4

r1-r2
ans =
((c^2 - d^2)*(c^2 + d^2 + 4*x))/4 - ((c^2 - d^2)*(c^2 + d^2 + 2*x))/4 -
(x*(c^2 - d^2))/2
simplify(ans)
ans =
0
```

Solving simple boundary value problems by integration

Consider the following BVP:

$$-\frac{d^2u}{dt^2} = 1+x,\ 0<x<1,$$
$$u(0)=2,\ u(1)=0.$$

The solution can be found by two integrations (cf. Example 2.2 in the text). Remember, MATLAB does not add a constant of integration, so I do it explicitly:

```
clear
syms x C1 C2
int(-(1+x),x)+C1
ans =
C1 - (x + 1)^2/2

int(ans,x)+C2
ans =
C2 + C1*x - (x + 1)^3/6
```

```
u=ans
u =
C2 + C1*x - (x + 1)^3/6
```

The above function u, with the proper choice of C1 and C2, is the desired solution. To find the constants, I solve the (algebraic) equations implied by the boundary conditions. The MATLAB command **solve** can be used for this purpose.

The MATLAB solve command

Before completing the previous example, I will explain the solve command on some simpler examples. Suppose I wish to solve the linear equation $ax+b=0$ for x. I can regard this as a root-finding problem: I want the root of the function $f(x)=ax+b$. The **solve** command finds the root of a function with respect to a given independent variable:

```
syms f x a b
f=a*x+b
f =
b + a*x
solve(f,x)
ans =
-b/a
```

If the equation has more than one solution, **solve** returns the possibilities in an array:

```
syms f x
f=x^2-3*x+2;
solve(f,x)
ans =
 1
 2
```

As these examples show, **solve** is used to find solutions of equations of the form $f(x)=0$; only the expression $f(x)$ is input to **solve**.

solve can also be used to solve a system of equations in several variables. In this case, the equations are listed first, followed by the unknowns. For example, suppose I wish to solve the equations

$$x+y=1,$$
$$2x-y=1.$$

Here is the command:

```
syms x y
s=solve(x+y-1,2*x-y-1,x,y)
s =
    x: [1x1 sym]
    y: [1x1 sym]
```

What kind of variable is the output *s*? If we list the variables in the workspace,

```
whos
  Name       Size             Bytes  Class     Attributes

  C1         1x1                 60  sym
  C2         1x1                 60  sym
  a          1x1                 60  sym
  ans       2x1                60  sym
  b         1x1                60  sym
  f         1x1                60  sym
  s         1x1               368  struct
  u         1x1                60  sym
  x         1x1                60  sym
  y         1x1                60  sym
```

we see that **s** is a 1 by 1 struct array, that is, an array containing a single struct. A struct is a data type with named fields that can be accessed using the syntax **variable.field**. The variable **s** has two fields:

```
s
s =
    x: [1x1 sym]
    y: [1x1 sym]
```

The two fields hold the values of the unknowns *x* and *y*:

```
s.x
ans =
2/3
s.y
ans =
1/3
```

If the system has more than one solution, then the output of **solve** will be a struct, each of whose fields is an array containing the values of the unknowns. Here is an example:

```
s=solve(x^2+y^2-1,y-x^2,x,y)
s =
    x: [4x1 sym]
    y: [4x1 sym]
```

The first solution is

```
pretty(s.x(1))

  /  1/2       \1/2
  | 5          |
  | ---- - 1/2 |
  \  2         /
```

```
pretty(s.y(1))

   1/2
  5
  ---- - 1/2
   2
```

The second solution is

```
pretty(s.x(2))

  /    1/2       \1/2
  |   5          |
  | - ---- - 1/2 |
  \    2         /
```

```
pretty(s.y(2))

     1/2
    5
  - ---- - 1/2
     2
```

You might notice that the second solution is complex (at least, the value of **x** is complex). This might be easier to see from the numerical value of the expressions, which we can see with the **double** command (which converts a symbolic expression to a floating point value, if possible)

```
double(s.x(2))
ans =
        0 + 1.2720i
double(s.y(2))
ans =
   -1.6180
```

Here are the remaining solutions (given in floating point)

```
double(s.x(3))
ans =
   -0.7862
double(s.y(3))
ans =
    0.6180

double(s.x(4))
ans =
        0 - 1.2720i
double(s.y(4))
ans =
   -1.6180
```

If the equation to be solved cannot be solved exactly, then **solve** automatically tries to solve it numerically, using extended precision arithmetic (approximately 32 decimal digits):

```
syms x
solve(x^5+sym(4)*x^4-x^3+x^2-x+1,x)
ans =
                                                     -
4.3019656878883790704888463433324
   0.40236742000277868343510597733001*sqrt(-1) +
0.49445817238673908475778249075192
 - 0.67380934595293810995276180453239*sqrt(-1) -
0.34347532844254954951335931908572
   0.67380934595293810995276180453239*sqrt(-1) -
0.34347532844254954951335931908572
   0.49445817238673908475778249075192 -
0.40236742000277868343510597733001*sqrt(-1)
```

Finally, there is another way to call **solve**. The equation and unknowns can be given as strings, meaning that there is no need to define symbolic variables before calling **solve**:

```
solve('x^2-2*x-1=0','x')
ans =
 2^(1/2) + 1
 1 - 2^(1/2)
```

Notice that, when specifying the equation in symbolically form, it must be expressed with the right-hand side equal to 0, and only the left-hand side is passed to solve (that is, to solve $f(x)=0$ for x, we call **solve** as **solve(f(x),x)**. However, when calling **solve** using strings instead of symbolic expressions, we can use either form, $f(x)=0$ (as above) or $f(x)$:

```
solve('x^2-2*x-1','x')
ans =
 2^(1/2) + 1
 1 - 2^(1/2)
```

Back to the example

We can now solve for the constants of integrations in the solution to the BVP

$$-\frac{d^2u}{dt^2}=1+x,\ 0<x<1,$$
$$u(0)=2,\ u(1)=0.$$

Recall that the solution is of the form

```
u
u =
C2 + C1*x - (x + 1)^3/6
```

We must use the boundary conditions to compute **C1** and **C2**. The equations are $u(0)-2=0$ and $u(1)=0$. Notice how I use the **subs** command to form $u(0)$ and $u(1)$:

```
s=solve(subs(u,x,0)-2,subs(u,x,1),C1,C2)
s =
    C1: [1x1 sym]
    C2: [1x1 sym]
```

Here are the values of **C1** and **C2**:

```
s.C1
ans =
-5/6

s.C2
ans =
13/6
```

Here is the solution:

```
u=subs(u,{C1,C2},{s.C1,s.C2})
u =
13/6 - (x + 1)^3/6 - (5*x)/6
```

Notice how, when substituting for two or more variables, the variables and the values are enclosed in curly braces.

Let us check our solution:

```
-diff(u,x,2)
ans =
x + 1

subs(u,x,sym(0))
ans =
2
subs(u,x,sym(1))
ans =
0
```

The differential equation and the boundary conditions are satisfied.

Another example

Now consider the BVP with a nonconstant coefficient:

$$-\frac{d}{dx}\left[\left(1+\frac{x}{2}\right)\frac{du}{dx}\right]=0,\ 0<x<1,$$
$$u(0)=20,\ u(1)=25.$$

Integrating once yields

$$\frac{du}{dx}(x)=\frac{C1}{1+x/2}.$$

(It is easier to perform this calculation in my head than to ask MATLAB to integrate 0.) I now perform the second integration:

```
clear
syms C1 C2 x
u=int(C1/(1+x/2),x)+C2
u =
C2 + 2*C1*log(x + 2)
```

Now I use **solve** to find C1 and C2:

```
s=solve(subs(u,x,0)-20,subs(u,x,1)-25,C1,C2)
s =
    C1: [1x1 sym]
    C2: [1x1 sym]
```

Here is the solution:

```
u=subs(u,{C1,C2},{s.C1,s.C2})
u =
(45035996273704960*log(x + 2))/3652105019575333 +
41825526550679865/3652105019575333
```

Notice the unexpected answer; the problem is that MATLAB did not interpret the constants in the equations (0, 1, 20, 25) as symbolic quantities, but rather as numerical (floating point) values. As a result, it used extended precision arithmetic while finding a solution. The desired solution can be found by specifying that the various numbers be treated as symbolic:

```
clear u
syms C1 C2 x
u=int(C1/(1+x/2),x)+C2
u =
C2 + 2*C1*log(x + 2)

s=solve(subs(u,x,sym(0))-sym(20),subs(u,x,sym(1))-sym(25),C1,C2)
s =
    C1: [1x1 sym]
    C2: [1x1 sym]

u=subs(u,{C1,C2},{s.C1,s.C2})
u =
(5*(5*log(2) - 4*log(3)))/(log(2) - log(3)) - (5*log(x + 2))/(log(2) -
log(3))
```

Now I will check the answer:

```
simplify(-diff((1+x/2)*diff(u,x),x))
ans =
0

subs(u,x,0)
ans =
    20
subs(u,x,1)
ans =
   25.0000
```

Chapter 3: Essential linear algebra

Section 3.1 Linear systems as linear operator equations

I have already showed you how to enter matrices and vectors in MATLAB. I will now introduce a few more elementary operations on matrices and vectors, and explain how to extract components from a vector and entries, rows, or columns from a matrix. At the end of this section, I will describe how MATLAB can perform symbolic linear algebra; until then, the examples will use floating point arithmetic.

```
clear
```

Consider the matrix

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

The transpose is indicated by a single quote following the matrix name:

```
A'
ans =
     1     4     7
     2     5     8
     3     6     9
```

Recall that, if *x* and *y* are column vectors, then the dot product of *x* and *y* can be computed as $x^T y$:

```
x=[1;0;-1]
x =
     1
     0
    -1

y=[1;2;3]
y =
     1
     2
     3

x'*y
ans =
    -2
```

Alternatively, I can use the **dot** function:

```
dot(x,y)
ans =
    -2
```

I can extract a component of a vector,

```
x(1)
ans =
     1
```

or an entry of a matrix:

```
A(2,3)
ans =
     6
```

In place of an index, I can use a colon, which represents the entire range. For example, **A**(:,1) indicates all of the rows in the first column of **A**. This yields a column vector:

```
A(:,1)
ans =
     1
     4
     7
```

Similarly, I can extract a row:

```
A(2,:)
ans =
     4     5     6
```

Section 3.2: Existence and uniqueness of solutions to Ax=b

MATLAB can find a basis for the null space of a matrix. Consider the matrix

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

Here is a basis for the null space:

```
x=null(B)
x =
   -0.4082
    0.8165
   -0.4082
```

Since MATLAB returned a single vector, this indicates that the null space is one-dimensional. Here is a check of the result:

```
B*x
ans =
  1.0e-015 *
   -0.2220
   -0.4441
         0
```

Notice that, since the computation was done in floating point, the product Bx is not exactly zero, but just very close.

If a matrix is nonsingular, its null space is trivial:

```
A=[1,2;2,1]
A =
     1     2
     2     1

null(A)
ans =
   Empty matrix: 2-by-0
```

On the other hand, the null space can have dimension greater than one:

```
A=[1 1 1;2 2 2;3 3 3;]
A =
     1     1     1
     2     2     2
     3     3     3

null(A)
ans =
   -0.8165         0
    0.4082    0.7071
    0.4082   -0.7071
```

The matrix **A** has a two-dimensional null space.

MATLAB can compute the inverse of a nonsingular matrix:

```
A=[1 0 -1;3 2 1;2 -1 1]
A =
     1     0    -1
     3     2     1
     2    -1     1
```

The command is called **inv**:

```
Ainv=inv(A)
Ainv =
    0.3000    0.1000    0.2000
   -0.1000    0.3000   -0.4000
   -0.7000    0.1000    0.2000

A*Ainv
ans =
    1.0000   -0.0000    0.0000
         0    1.0000    0.0000
         0         0    1.0000
```

Using the inverse, you can solve a linear system

```
b=[1;1;1]
b =
     1
     1
     1

x=Ainv*b
x =
    0.6000
   -0.2000
   -0.4000

A*x
ans =
    1.0000
    1.0000
    1.0000
```

On the other hand, computing and using the inverse of a matrix A is not the most efficient way to solve $Ax=b$. It is preferable to solve the system directly using some variant of Gaussian elimination. The backslash operator indicates to MATLAB that a linear system is to be solved:

```
x1=A\b
x1 =
    0.6000
   -0.2000
   -0.4000
x-x1
ans =
     0
     0
     0
```

(To remember how the backslash operator works, just think of **A\b** as "dividing b on the left by A", or multiplying b on the left by A^{-1}. However, MATLAB does not compute the inverse.)

Section 3.3: Basis and dimension

In this section, I will further demonstrate some of the capabilities of MATLAB by repeating some of the examples from Section 3.3 of the text.

Example 3.25

Consider the three vectors **v1**, **v2**, **v3** defined as follows:

```
v1=[1/sqrt(3);1/sqrt(3);1/sqrt(3)]
v1 =
    0.5774
    0.5774
    0.5774
v2=[1/sqrt(2);0;-1/sqrt(2)]
v2 =
    0.7071
         0
   -0.7071
v3=[1/sqrt(6);-2/sqrt(6);1/sqrt(6)]
v3 =
    0.4082
   -0.8165
    0.4082
```

I will verify that these vectors are orthogonal:

```
v1'*v2
ans =
     0
v1'*v3
ans =
     0
v2'*v3
ans =
     0
```

Example 3.27

I will verify that the following three quadratic polynomials for a basis for P_2. Note that while I did the previous example in floating point arithmetic, this examples requires symbolic computation.

```
clear
syms p1 p2 p3 x
p1=1
p1 =
     1
p2=x-1/2
p2 =
x - 1/2
```

```
p3=x^2-x+1/6
p3 =
x^2 - x + 1/6
```

Now suppose that $q(x)$ is an arbitrary quadratic polynomial:

```
syms q a b c
q=a*x^2+b*x+c
q =
a*x^2 + b*x + c
```

I want to show that **q** can be written in terms of **p1**, **p2**, and **p3**:

```
syms c1 c2 c3

q-(c1*p1+c2*p2+c3*p3)
ans =
c - c1 + b*x + a*x^2 - c2*(x - 1/2) - c3*(x^2 - x + 1/6)
```

I need to gather like terms in this expression, which is accomplished with the **collect** command:

```
collect(ans,x)
ans =
(a - c3)*x^2 + (b - c2 + c3)*x + c - c1 + c2/2 - c3/6
```

I can now set each coefficient equal to zero and solve for the coefficients:

```
r=solve(a-c3,b-c2+c3,c-c1+c2/2-c3/6,c1,c2,c3)
r =
    c1: [1x1 sym]
    c2: [1x1 sym]
    c3: [1x1 sym]
r.c1
ans =
a/3 + b/2 + c

r.c2
ans =
a + b
r.c3
ans =
a
```

Check:

```
q-(r.c1*p1+r.c2*p2+r.c3*p3)
ans =
b*x - b/2 - a/3 - (a + b)*(x - 1/2) + a*x^2 - a*(x^2 - x + 1/6)
simplify(ans)
ans =
0
```

The fact that there is a unique solution to this system, regardless of the values of **a**, **b**, **c**, shows that every quadratic polynomial can be uniquely written as a linear combination of **p1**, **p2**, **p3**, and hence that these three polynomials form a basis for P_2.

Example

Here is a final example. Consider the following three vectors in R^3:

```
u1=[1;0;2];
u2=[0;1;1];
u3=[1;2;-1];
```

I will verify that {**u1**,**u2**,**u3**} is a basis for R^3, and express the vector

```
x=[8;2;-4];
```

in terms of the basis. As discussed in the text, {**u1**,**u2**,**u3**} is a basis if and only if the matrix A whose columns are **u1**, **u2**, **u3** is nonsingular. It is easy to form the matrix A:

```
A=[u1,u2,u3]
A =
     1     0     1
     0     1     2
     2     1    -1
```

One way (that works fine for such a small matrix, but is not a good method in general) to determine if **A** is nonsingular is to compute its determinant:

```
det(A)
ans =
    -5
```

Another way to determine whether **A** is nonsingular is to simply try to solve a system involving **A**, since MATLAB will print a warning or error message if **A** is singular:

```
c=A\x
c =
    3.6000
   -6.8000
    4.4000
```

Here is a verification of the results:

```
x-(c(1)*u1+c(2)*u2+c(3)*u3)
ans =
     0
     0
     0
```

Symbolic linear algebra

Recall that MATLAB performs its calculations in floating point arithmetic by default. However, if desired, we can perform them symbolically. For an illustration, I will repeat the previous example.

```
clear
syms u1 u2 u3
u1=sym([1;0;2]);
u2=sym([0;1;1]);
u3=sym([1;2;-1]);
A=[u1,u2,u3]
```

```
A =
[ 1, 0,  1]
[ 0, 1,  2]
[ 2, 1, -1]
x=sym([8;2;-4]);

c=A\x
c =
  18/5
 -34/5
  22/5
```

The solution is the same as before.

Programming in MATLAB, part I

Before I continue on to Section 3.4, I want to explain simple programming in MATLAB---specifically, how to define new MATLAB functions. I have already shown you one way to do this: using the **@** operator. (Also, a symbolic expression can be used in place of a function for many purposes. For instance, it can be evalutated using the **subs** command.) However, in addition to the **@** operator, there is another method for defining MATLAB functions.

Defining a MATLAB function in an M-file.

An M-file is a plain text file whose name ends in ".m" and which contains MATLAB commands. There are two types of M-files, scripts and functions. I will explain scripts later. A function is a MATLAB subprogram: it accepts inputs and computes outputs using local variables. The first line in a function must be of the form

```
function [output1,output2,…]=function_name(input1,input2,…)
```

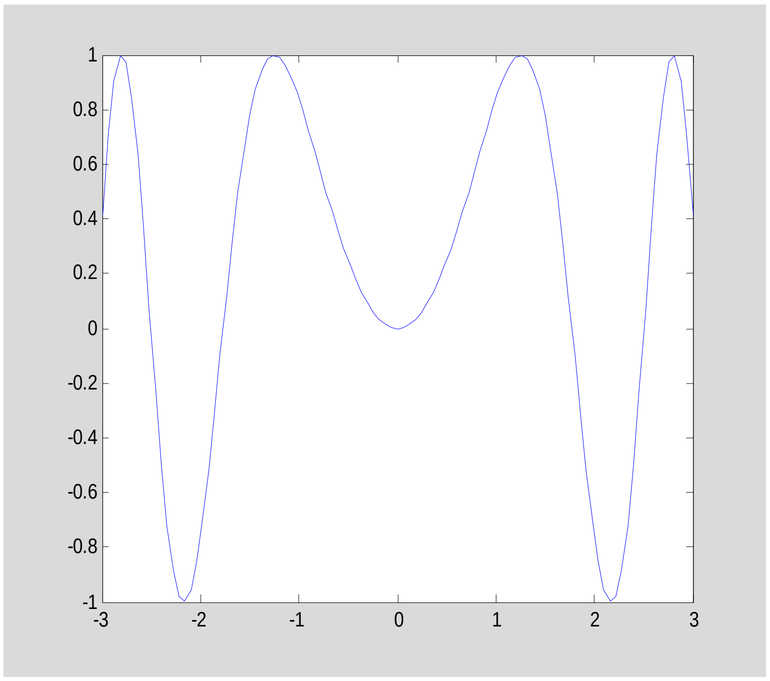
If there is a single output, the square brackets can be omitted. Also, a function can have zero inputs and/or zero outputs.

Here is the simplest type of example. Suppose I wish to define the function $f(x)=\sin(x^2)$. The following lines, stored in the M-file **f.m**, accomplish this:

```
function y=f(x)
y=sin(x.^2);
```

(Notice how I use the ".^" operator to vectorize the computation. Predefined MATLAB functions are always vectorized, and user-defined functions typically should be as well.) I can now use **f** just as a pre-defined function such as **sin**. For example:

```
clear
x=linspace(-3,3,101);
plot(x,f(x))
```



A few important points:

- The names of user-defined functions can be listed using the **what,** command (similar to **who**, but instead of listing variables, it lists M-files in the working directory).
- The contents of an M-file can be displayed using the **type** command.
- In order for you to use an M-file, MATLAB must be able to find it, which means that the M-file must be in a directory on the MATLAB path. The MATLAB path always includes the working directory, which can be determined using the **pwd** (print working directory) command. The MATLAB path can be listed using the **path** command. Other directories can be added to the MATLAB path using the **addpath** command. For more information, see "**help addpath**".

The current path is

```
path

     MATLABPATH

  fempack
  C:\Users\Guest\Documents\MATLAB
  C:\Program Files (x86)\MATLAB\R2011a\toolbox\matlab\general
  C:\Program Files (x86)\MATLAB\R2011a\toolbox\matlab\ops
  C:\Program Files (x86)\MATLAB\R2011a\toolbox\matlab\lang
  C:\Program Files (x86)\MATLAB\R2011a\toolbox\matlab\elmat
  C:\Program Files (x86)\MATLAB\R2011a\toolbox\matlab\randfun
  C:\Program Files (x86)\MATLAB\R2011a\toolbox\matlab\elfun
  C:\Program Files (x86)\MATLAB\R2011a\toolbox\matlab\specfun
  C:\Program Files (x86)\MATLAB\R2011a\toolbox\matlab\matfun
  C:\Program Files (x86)\MATLAB\R2011a\toolbox\matlab\datafun
  C:\Program Files (x86)\MATLAB\R2011a\toolbox\matlab\polyfun
  C:\Program Files (x86)\MATLAB\R2011a\toolbox\matlab\funfun
  C:\Program Files (x86)\MATLAB\R2011a\toolbox\matlab\sparfun
  C:\Program Files (x86)\MATLAB\R2011a\toolbox\matlab\scribe
  C:\Program Files (x86)\MATLAB\R2011a\toolbox\matlab\graph2d
  C:\Program Files (x86)\MATLAB\R2011a\toolbox\matlab\graph3d
  C:\Program Files (x86)\MATLAB\R2011a\toolbox\matlab\specgraph
  C:\Program Files (x86)\MATLAB\R2011a\toolbox\matlab\graphics
  C:\Program Files (x86)\MATLAB\R2011a\toolbox\matlab\uitools
  C:\Program Files (x86)\MATLAB\R2011a\toolbox\matlab\strfun
  C:\Program Files (x86)\MATLAB\R2011a\toolbox\matlab\imagesci
  C:\Program Files (x86)\MATLAB\R2011a\toolbox\matlab\iofun
  C:\Program Files (x86)\MATLAB\R2011a\toolbox\matlab\audiovideo
  C:\Program Files (x86)\MATLAB\R2011a\toolbox\matlab\timefun
  C:\Program Files (x86)\MATLAB\R2011a\toolbox\matlab\datatypes
  C:\Program Files (x86)\MATLAB\R2011a\toolbox\matlab\verctrl
  C:\Program Files (x86)\MATLAB\R2011a\toolbox\matlab\codetools
  C:\Program Files (x86)\MATLAB\R2011a\toolbox\matlab\helptools
  C:\Program Files (x86)\MATLAB\R2011a\toolbox\matlab\winfun
  C:\Program Files (x86)\MATLAB\R2011a\toolbox\matlab\winfun\NET
  C:\Program Files (x86)\MATLAB\R2011a\toolbox\matlab\demos
  C:\Program Files (x86)\MATLAB\R2011a\toolbox\matlab\timeseries
  C:\Program Files (x86)\MATLAB\R2011a\toolbox\matlab\hds
  C:\Program Files (x86)\MATLAB\R2011a\toolbox\matlab\guide
  C:\Program Files (x86)\MATLAB\R2011a\toolbox\matlab\plottools
  C:\Program Files (x86)\MATLAB\R2011a\toolbox\local
  C:\Program Files (x86)\MATLAB\R2011a\toolbox\matlab\datamanager
  C:\Program Files (x86)\MATLAB\R2011a\toolbox\shared\simulink
  C:\Program Files (x86)\MATLAB\R2011a\toolbox\shared\instrument
  C:\Program Files (x86)\MATLAB\R2011a\toolbox\shared\asynciolib
  C:\Program Files (x86)\MATLAB\R2011a\toolbox\shared\comparisons
  C:\Program Files
(x86)\MATLAB\R2011a\toolbox\shared\controllib\general
  C:\Program Files
(x86)\MATLAB\R2011a\toolbox\shared\controllib\graphics
  C:\Program Files (x86)\MATLAB\R2011a\toolbox\optim\optim
  C:\Program Files (x86)\MATLAB\R2011a\toolbox\optim\optimdemos
  C:\Program Files (x86)\MATLAB\R2011a\toolbox\shared\optimlib
  C:\Program Files (x86)\MATLAB\R2011a\toolbox\pde
  C:\Program Files (x86)\MATLAB\R2011a\toolbox\pde\pdedemos
  C:\Program Files (x86)\MATLAB\R2011a\toolbox\shared\eml\eml
  C:\Program Files (x86)\MATLAB\R2011a\toolbox\symbolic\symbolic
  C:\Program Files (x86)\MATLAB\R2011a\toolbox\symbolic\symbolicdemos
  C:\Program Files
(x86)\MATLAB\R2011a\toolbox\shared\testmeaslib\general
  C:\Program Files
(x86)\MATLAB\R2011a\toolbox\shared\testmeaslib\graphics
```

The working directory is

```
pwd
ans =
C:\Users\Guest\Documents\MATLAB\msgocken
```

The files associated with this tutorial are (on my computer) in C:\Users\Guest\Documents\MATLAB\msgocken. (When you install the tutorial on your computer, you will have to make sure that all of the files that come with the tutorial are in an accessible directory.) Here are all of the M-files in the directory called msgocken:

```
what msgocken

M-files in directory C:\Users\Guest\Documents\MATLAB\msgocken
```

```
HWProblem6Function     f1                     l2ip
scriptex
beuler                 f2                     l2norm
startup
eip                    f2a                    mkpl
testfun
euler                  f6                     myplot
testit
euler1                 g                      myplot1
testsym
euler2                 g1                     mysubs
f                      h                      mysubs1
```

I can look at **f.m**:

```
type f

function y=f(x)

y=sin(x.^2);
```

One important feature of functions defined in M-files is that variables defined in an M-file are *local*; that is, they exist only while the M-file is being executed. Moreover, variables defined in the MATLAB workspace (that is, the variables listed when you give the **who** command at the MATLAB prompt) are not accessible inside of an M-file function. Here is are examples:

```
type g

function y=g(x)

a=3;
y=sin(a*x);
clear
g(1)
ans =
    0.1411

a
??? Undefined function or variable 'a'.
```

The variable **a** is not defined in the MATLAB workspace after **g** executes, since a was local to the M-file **g.m**.

On the other hand, consider:

```
type h

function y=g(x)

y=sin(a*x);

clear

a=3
a =
     3
```

```
h(1)
??? Undefined function or variable 'a'.

Error in ==> h at 3
y=sin(a*x);
```

The M-file **h.m** cannot use the variable **a**, since the MATLAB workspace is not accessible within **h.m**. (In short, we say that **a** is not "in scope".)

Here is an example with two outputs:

```
type f1

function [y,dy]=f1(x)

y=3*x.^2-x+4;
dy=6*x-1;
```

The M-file function **f1.m** computes both $f(x)=3x^2-x+4$ and its derivative. It can be used as follows:

```
[v1,v2]=f1(1)
v1 =
     6
v2 =
     5
```

As another example, here is a function of two variables:

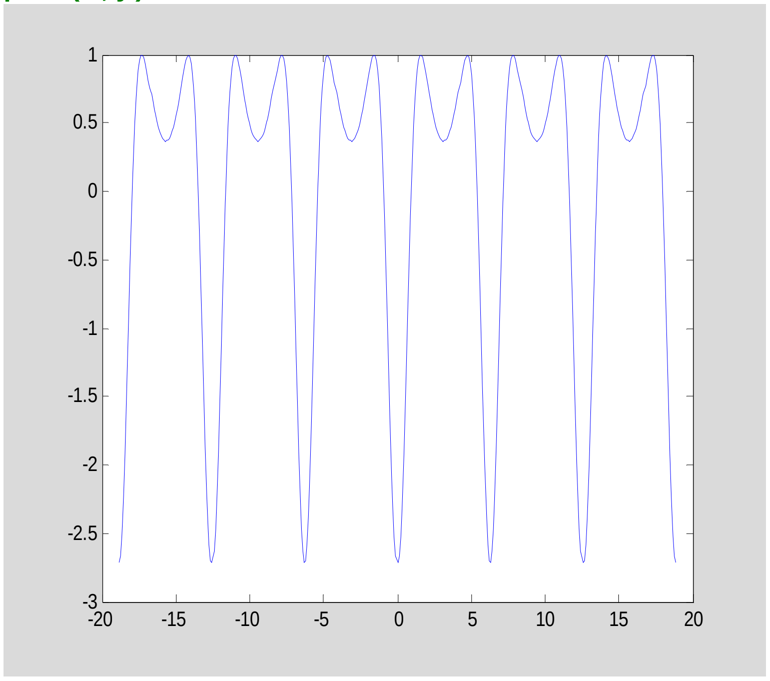
```
type g1

function z=g1(x,y)

z=2*x.^2+y.^2+x.*y;

g1(1,2)
ans =
     8
```

A function that is defined in an m-file can be given an alias---another name---inside of MATLAB. This is done by using the **@** operator to create a "function handle". This is useful because it allows you to give the file a meaningful name, such as **HWproblem6Function.m**, and then use a convenient alias, such as **f**, when typing MATLAB commands.

```
type HWProblem6Function

function y=HWProblem6Function(x)

y=exp(cos(x)).*sin(x).^2 - exp(cos(x)).*cos(x);

f=@HWProblem6Function
f =
    @HWProblem6Function

x=linspace(-6*pi,6*pi,401)';
```

```
y=f(x);
```

```
plot(x,y)
```



As we will see later, function handles are useful for passing one function as input to another function.

Optional inputs with default values

I now want to explain the use of optional inputs in M-files. Suppose you are going to be working with the function $f_2(x)=\sin(ax^2)$, and you know that, most of the time, the parameter **a** will have value 1. However, **a** will occasionally take other values. It would be nice if you only had to give the input **a** when its value is not the typical value of 1. The following M-file has this feature:

```
type f2

function y=f2(x,a)

if nargin<2
   a=1;
end

y=sin(a.*x.^2);
```

The first executable statement, "if nargin<2", checks to see if **f2** was invoked with one input or two. The keyword **nargin** is an automatic (local) variable whose value is the number of inputs. The M-file **f2.m** assigns the input **a** to have the value 1 if the user did not provide it. Now **f2** can be used with one or two parameters:

```
f2(pi)
ans =
   -0.4303
f2(pi,sqrt(2))
ans =
    0.9839
```

Comments in M-files

If the percent sign % is used in a MATLAB command, the remainder of the line is considered to be a comment and is ignored by MATLAB. Here is an example:

```
if sin(pi)==0   % Testing for roundoff error
  disp('No roundoff error')
else
  disp('Roundoff error detected')
end
```

(Notice the use of the **disp** command, which displays a string or the value of a variable. See "**help disp**" for more information. Notice also the use of the **if-else** block, which I discuss later in the tutorial.)

The most common use of comments is for documentation in M-files. Here is a second version of the function **f2** defined above:

```
type f2a

function y=f2a(x,a)

%y=f2a(x,a)
%
%  This function implements the function f2(x)=sin(a*x).  The parameter
%  a is optional; if it is not provided, it is taken to be 1.

if nargin<2
   a=1;
end

y=sin(a*x);
```

Notice how the block of comment lines explains the purpose and usage of the function. One of the convenient features of the MATLAB help system is that the first block of comments in an M-file is displayed when "help" is requested for that function:

```
help f2a
 y=f2a(x,a)

   This function implements the function f2(x)=sin(a*x).  The parameter
   a is optional; if it is not provided, it is taken to be 1.
```

I will explain more about MATLAB programming in Chapter 4.

M-files as scripts

An M-file that does not begin with the word **function** is regarded as a script, which is nothing more than a collection of MATLAB commands that are executed sequentially, just as if they had been typed at the MATLAB prompt. Scripts do not have local variables, and do not accept inputs or return outputs. A common use for a script is to collect the commands that define and solve a certain problem (e.g. a homework problem).

Here is a sample script. Its purpose is to graph the function

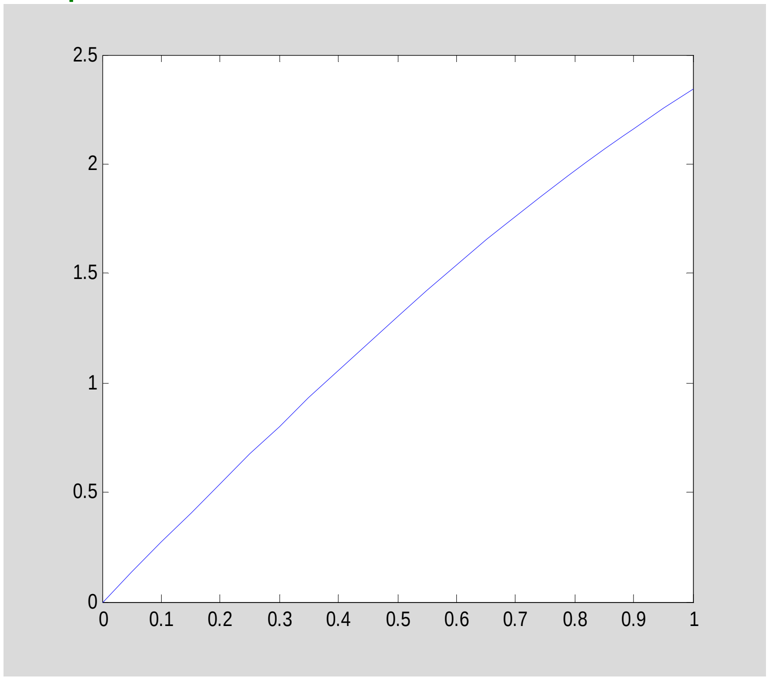
$$f(x) = \int_0^x e^{\cos(s)}\,ds$$

on the interval [0,1]. (Recall that MATLAB cannot compute the integral explicitly, so this is a nontrivial task.) (Caveat: I did not try to make the following script efficient; indeed, it is decidedly inefficient!)

```
type scriptex


% Define the integrand

g=@(s)exp(cos(s));

% Create a grid

x=linspace(0,1,21);
y=zeros(1,21);

% Evaluate the integral int(exp(cos(s)),s,0,x) for
% each value of x on the grid:

for ii=1:length(x)
   y(ii)=quad(g,0,x(ii));
end

% Now plot the result

plot(x,y)
```

Here is the result of running **scriptex.m**:

```
clear
scriptex
```



As I mentioned above, scripts do not have local variables. Any variables defined in scriptex exist in the MATLAB workspace:

```
whos
  Name       Size               Bytes  Class              Attributes

  g          1x1                   16  function_handle
  ii         1x1                    8  double
  x          1x21                 168  double
  y          1x21                 168  double
```

Section 3.4: Orthogonal bases and projection

Consider the following three vectors:

```
clear
v1=[1/sqrt(3);1/sqrt(3);1/sqrt(3)];
v2=[1/sqrt(2);0;-1/sqrt(2)];
v3=[1/sqrt(6);-2/sqrt(6);1/sqrt(6)];
```

These vectors are orthonormal, as can easily be checked. Therefore, I can easily express any vector as a linear combination of the three vectors, which form an orthonormal basis. To test this, I will use the **randn** command to generate a random vector (for more information, see "**help randn**"):

```
x=randn(3,1)
x =
   -0.2050
   -0.1241
    1.4897

y=(x'*v1)*v1+(x'*v2)*v2+(x'*v3)*v3
y =
   -0.2050
   -0.1241
    1.4897

y-x
ans =
  1.0e-015 *
    0.4718
   -0.0139
         0
```

Notice that the difference between **y** and **x** (which should be equal) is due to roundoff error, and is very small.

Working with the L^2 inner product

Since MATLAB can compute integrals symbolically, we can use it to compute the L^2 inner product and norm. For example:

```
clear
syms x
f=x
f =
x

g=x^2
g =
x^2

int(f*g,x,0,1)
ans =
1/4
```

If you are going to perform such calculations repeatedly, it is convenient to define a function to compute the L^2 inner product. The M-file **l2ip.m** does this:

```
help l2ip
 I=l2ip(f,g,a,b,x)

    This function computes the L^2 inner product of two functions
    f(x) and g(x), that is, it computes the integral from a to b of
    f(x)*g(x).  The two functions must be defined by symbolic
    expressions f and g.

    The variable of integration is assumed to be x.  A different
    variable can passed in as the (optional) fifth input.

    The inputs a and b, defining the interval [a,b] of integration,
    are optional.  The default values are a=0 and b=1.
```

Notice that I assigned the default interval to [0,1]. Here is an example:

```
syms x
l2ip(x,x^2)
ans =
1/4
```

For convenience, I also define a function computing the L^2 norm:

```
help l2norm
 I=l2norm(f,a,b,x)

    This function computes the L^2 inner product of the function f(x)
    The functions must be defined by the symbolic expressions f.

    The variable of integration is assumed to be x.  A different
    variable can passed in as the (optional) fourth input.

    The inputs a and b, defining the interval [a,b] of integration,
    are optional.  The default values are a=0 and b=1.

l2norm(x)
ans =
3^(1/2)/3
double(ans)
ans =
    0.5774
```

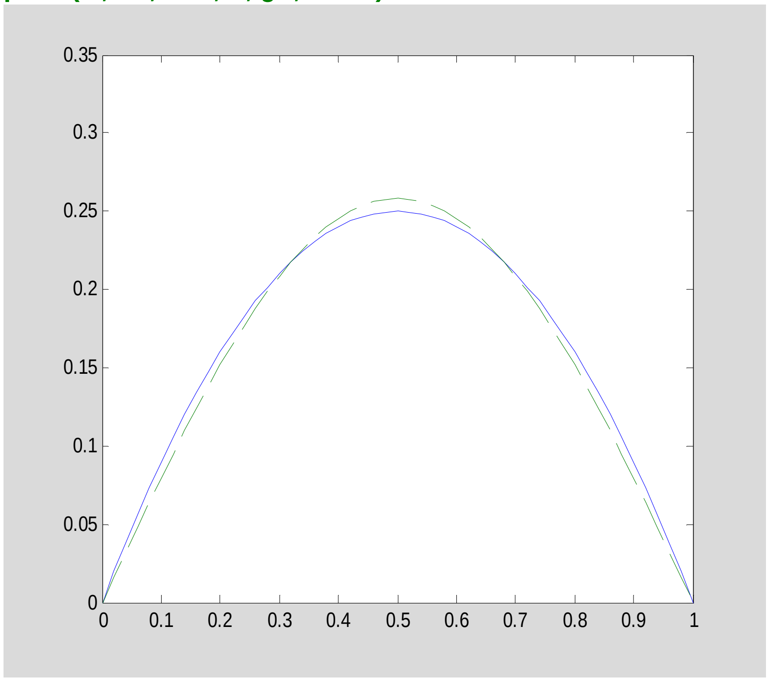
Example 3.35

Now consider the following two functions:

```
clear
syms x pi
f=x*(1-x)
f =
-x*(x - 1)

g=8/pi^3*sin(pi*x)
g =
(8*sin(pi*x))/pi^3
```

(I must tell MATLAB that **pi** is to be regarded as symbolic.) The following graph shows that the two functions are quite similar on the interval [0,1]:

```
t=linspace(0,1,51);
f1=subs(f,x,t);
g1=subs(g,x,t);
plot(t,f1,'-',t,g1,'--')
```



By how much do the two functions differ? One way to answer this question is to compute the relative difference in the L^2 norm:

```
l2norm(f-g)/l2norm(f)
ans =
30^(1/2)*(1/30 - 32/pi^6)^(1/2)
double(ans)
ans =
    0.0380
```

The difference is less than 4%.

Here are two more examples from Section 3.4 that illustrate some of the capabilities of MATLAB.

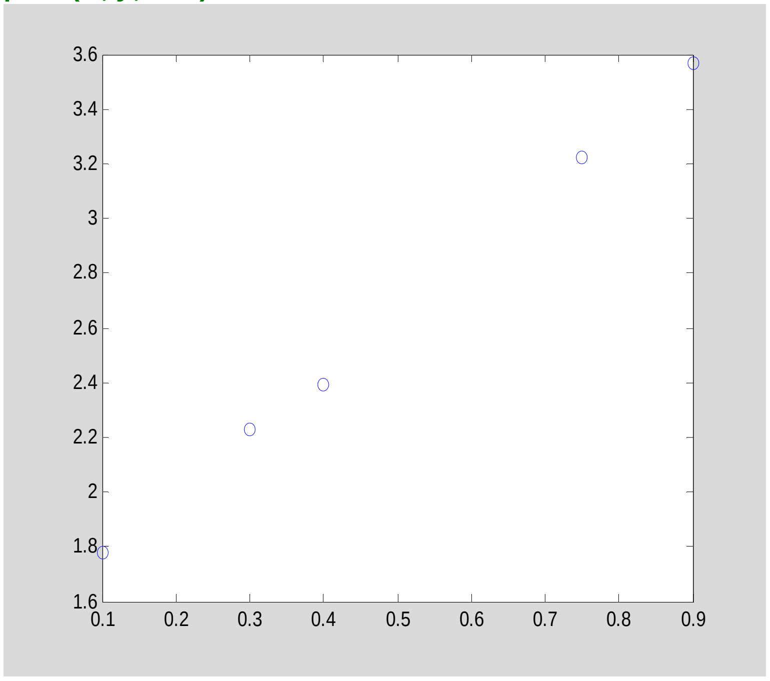
Example 3.37

The purpose of this example to compute the first-degree polynomial $f(x)=mx+b$ best fitting given data points (x_i,y_i). The data given in this example can be stored in two vectors:

```
clear
x=[0.1;0.3;0.4;0.75;0.9];

y=[1.7805;2.2285;2.3941;3.2226;3.5697];
```

Here is a plot of the data:

```
plot(x,y,'o')
```



To compute the first-order polynomial $f(x)=mx+b$ that best fits this data, I first form the Gram matrix. The **ones** command creates a vector of all ones:

```
e=ones(5,1)
e =
     1
     1
     1
     1
     1

G=[x'*x,x'*e;e'*x,e'*e]
G =
    1.6325    2.4500
    2.4500    5.0000
```

Next, I compute the right hand side of the normal equations:

```
z=[x'*y;e'*y]
z =
    7.4339
   13.1954
```

Now I can solve for the coefficients **c=[m;b]**:

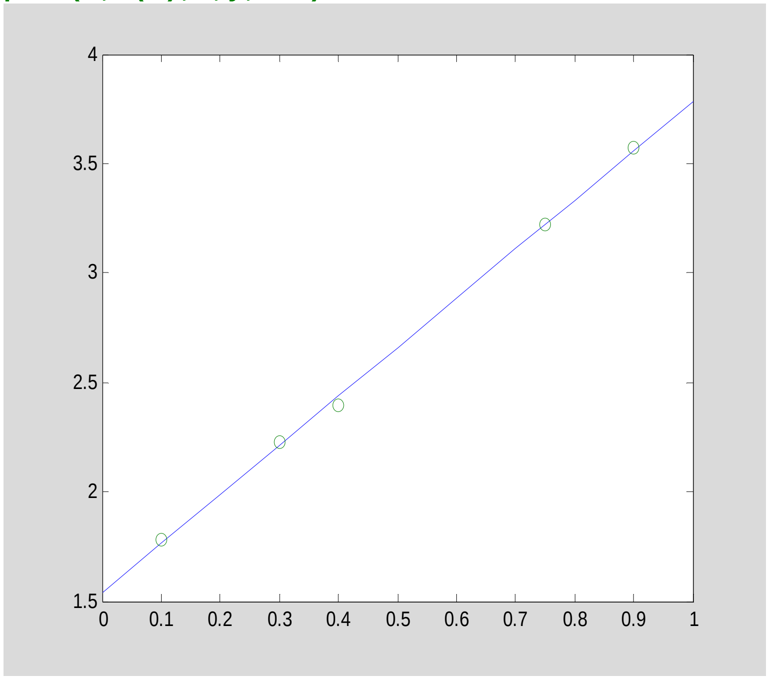
```
c=G\z
c =
    2.2411
    1.5409
```

I will define the solution as a function:

```
l=@(x)c(1)*x+c(2)
l =
    @(x)c(1)*x+c(2)
```

Here is a plot of the best fit line, together with the data:

```
t=linspace(0,1,11);
plot(t,l(t),x,y,'o')
```



The fit is not bad.

Example 3.38

In this example, I will compute the best quadratic approximation to the function e^x on the interval [0,1]. Here are the basis functions for the subspace P_2:

```
clear
syms x
p1=1;
p2=x;
p3=x^2;
```

I now compute the Gram matrix and the right hand side of the normal equations:

```
G=[l2ip(p1,p1),l2ip(p1,p2),l2ip(p1,p3)
l2ip(p2,p1),l2ip(p2,p2),l2ip(p2,p3)
l2ip(p3,p1),l2ip(p3,p2),l2ip(p3,p3)]
G =
[   1, 1/2, 1/3]
[ 1/2, 1/3, 1/4]
[ 1/3, 1/4, 1/5]


b=[l2ip(p1,exp(x));l2ip(p2,exp(x));l2ip(p3,exp(x))]
b =
 exp(1) - 1
          1
 exp(1) - 2
```

Now I solve the normal equations and find the best fit quadratic:

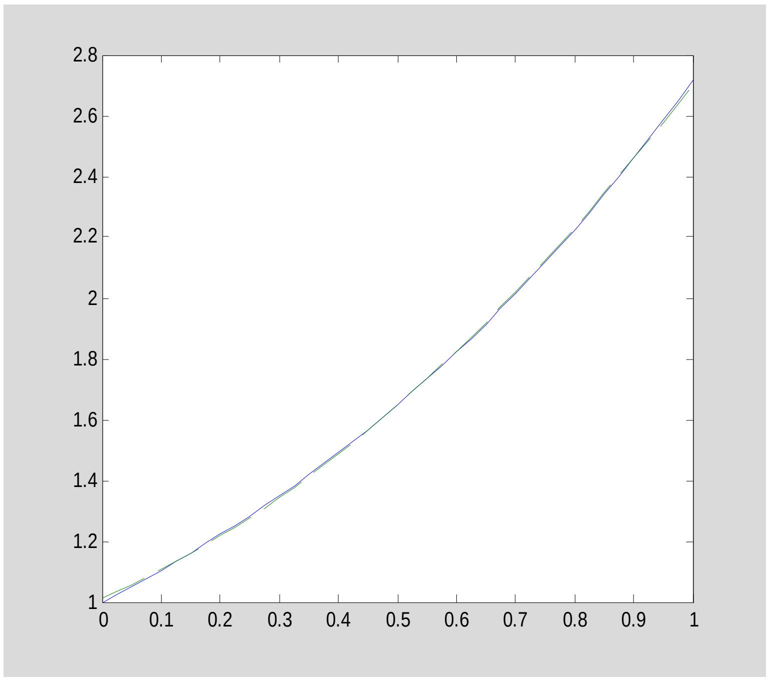
```
c=G\b
c =
  39*exp(1) - 105
 588 - 216*exp(1)
 210*exp(1) - 570
```

Here is the solution:

```
q=c(1)*p1+c(2)*p2+c(3)*p3
q =
(210*exp(1) - 570)*x^2 + (588 - 216*exp(1))*x + 39*exp(1) - 105
```

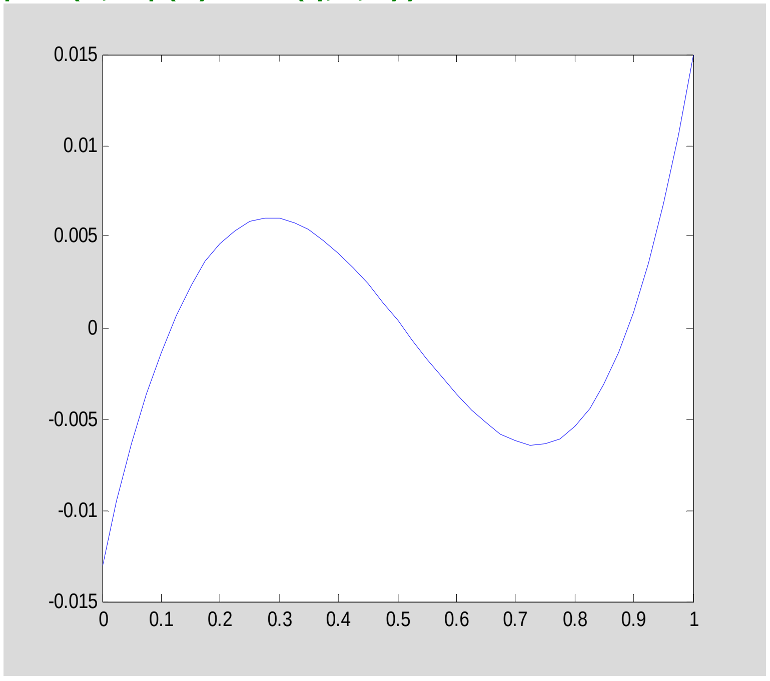
Here is the graph of $y=e^x$ and the quadratic approximation:

```
t=linspace(0,1,41)';
plot(t,exp(t),'-',t,subs(q,x,t),'--')
```



Since the approximation is so accurate, it is more informative to graph the error:

```
plot(t,exp(t)-subs(q,x,t))
```



We can also judge the fit by computing the relative error in the L^2 norm:

```
l2norm(exp(x)-q)/l2norm(exp(x))
ans =
(1350*exp(1) - (497*exp(2))/2 - 3667/2)^(1/2)/(exp(2)/2 - 1/2)^(1/2)
double(ans)
ans =
    0.0030
```

The error is less than 0.3%.

Section 3.5: Eigenvalues and eigenvectors of a symmetric matrix

MATLAB can compute eigenvalues and eigenvectors of a square matrix, either numerically or symbolically (when possible).

Numerical eigenvalues and eigenvectors

Recall that a matrix or any other quantity is stored in floating point by default:

```
clear
A=[1,2,-1;4,0,1;-7,-2,3]
A =
     1     2    -1
     4     0     1
    -7    -2     3
```

The **eig** command computes the eigenvalues and eigenvectors:

```
[V,D]=eig(A)
V =
   -0.4986    0.2554         0
    0.8056    0.0113    0.4472
   -0.3200   -0.9668    0.8944
D =
   -2.8730         0         0
         0    4.8730         0
         0         0    2.0000
```

The **eig** command returns two matrices. The first contains the eigenvectors as the columns of the matrix, while the second is a diagonal matrix with the eigenvalues on the diagonal. The eigenvectors and eigenvalues are given in the same order. For example:

```
A*V(:,1)-D(1,1)*V(:,1)
ans =
  1.0e-014 *
    0.1998
   -0.1332
    0.0222
```

(Notice how the colon is used to extract the first column of **A**.) The eigenvalue-eigenvector equation holds up to roundoff error.

It is also possible to call the **eig** command with a single output, in which case only the eigenvalues are returned, and in a vector instead of a matrix:

```
ev=eig(A)
ev =
   -2.8730
    4.8730
    2.0000
```

Symbolic eigenvalues and eigenvectors

To obtain symbolic (exact) eigenvalues and eigenvectors, it is only necessary to define the matrix to be symbolic:

```
clear
A=sym([1,2,-1;4,0,1;-7,-2,3])
A =
[  1,  2, -1]
[  4,  0,  1]
[ -7, -2,  3]
```

The computation then proceeds exactly as before:

```
[V,D]=eig(A)
V =
[    (4*15^(1/2))/17 + 11/17,  11/17 - (4*15^(1/2))/17,   0]
[ - (11*15^(1/2))/34 - 43/34, (11*15^(1/2))/34 - 43/34, 1/2]
[                          1,                        1,   1]
D =
[ 1 - 15^(1/2),            0, 0]
[            0, 15^(1/2) + 1, 0]
[            0,            0, 2]
```

Recall that you can use the **pretty** command to make output easier to read. For example, here is the second eigenvector:

```
pretty(V(:,2))

  +-                  -+
  |               1/2  |
  |           4 15     |
  |   11/17 - -------  |
  |             17     |
  |                    |
  |       1/2          |
  |  11 15             |
  |  -------- - 43/34  |
  |     34             |
  |                    |
  |          1         |
  +-                  -+
```

It is not always possible to compute eigenvalues and eigenvectors exactly. Indeed, the eigenvalues of an n by n matrix are the roots of the nth-degree characteristic polynomial. One of the most famous results of 19th century mathematics is that it is impossible to find a formula (analogous to the quadratic formula) expressing the roots of an arbitrary polynomial of degree 5 or more. For this reason, MATLAB cannot always find the eigenvalues of a symbolic matrix exactly. When it cannot, it automatically computes them approximately, using high precision arithmetic.

Here is a famous matrix, the Hilbert matrix:

```
clear
H=sym(hilb(5))
H =
[   1, 1/2, 1/3, 1/4, 1/5]
[ 1/2, 1/3, 1/4, 1/5, 1/6]
[ 1/3, 1/4, 1/5, 1/6, 1/7]
[ 1/4, 1/5, 1/6, 1/7, 1/8]
[ 1/5, 1/6, 1/7, 1/8, 1/9]
```

I will now try to compute the eigenvalues and eigenvectors symbolically:

```
[V,D]=eig(H);
```

Here is the first computed eigenvector:

```
V(:,1)
ans =
 0.016409133693801387188101700354447
 -0.31015050746487107774850800716253
   1.3453013982367055055233302583319
  -2.0390705016861528742291328798663
                                 1.0
```

Notice that the result appears to be in floating point, but with a large number of digits. In fact, MATLAB has returned a symbolic quantity, as the command **whos** shows:

```
whos
  Name         Size                Bytes  Class     Attributes

  D            5x5                    60  sym
  H            5x5                    60  sym
  V            5x5                    60  sym
  ans          5x1                    60  sym
```

The results are highly accurate, as the number of digits would suggest:

```
H*V(:,1)-D(1,1)*V(:,1)
ans =
   2.4839188867621254616420005487349*10^(-40)
  2.5796608527140381819915257178503*10^(-40)
 -3.2318146482723256106713975379415*10^(-41)
  1.8942051991510714756211514642121*10^(-40)
  8.1080530444298240540717917418738*10^(-41)
```

The explanation of these results is that when MATLAB could not compute the eigenpairs symbolically, it automatically switched to variable precision arithmetic, which, by default, uses 32 digits in all calculations. I will not have any need for variable precision arithmetic in this tutorial, but I wanted to mention it in case you encounter it unexpectedly, as in the above example.   You may wish to explore this capability of MATLAB further for your own personal use. You can start with "**help vpa**".

Since it is not usually possible to find eigenpairs symbolically (except for small matrices), it is typical to perform a floating point computation from the very beginning.

Example 3.49

I will now use the spectral method to solve **Ax**=**b**, where **A** and **b** are as defined below:

```
clear
A=[11,-4,-1;-4,14,-4;-1,-4,11]
A =
    11    -4    -1
    -4    14    -4
    -1    -4    11
b=[1;2;1]
b =
     1
     2
     1
```

The matrix **A** is symmetric, so its eigenvalues are necessarily real and its eigenvectors orthogonal. Moreover, MATLAB's **eig** returns normalized eigenvectors when the input is a floating point matrix.

```
[V,D]=eig(A)
V =
    0.5774    0.7071   -0.4082
    0.5774   -0.0000    0.8165
    0.5774   -0.7071   -0.4082
D =
    6.0000         0         0
         0   12.0000         0
         0         0   18.0000
```

The solution of **Ax**=**b** is then

```
x=(V(:,1)'*b/D(1,1))*V(:,1)+(V(:,2)'*b/D(2,2))*V(:,2)+(V(:,3)'*b/D(3,3)
)*V(:,3)
x =
    0.2037
    0.2593
    0.2037
```

Check:

```
A*x-b
ans =
  1.0e-015 *
    0.6661
    0.8882
   -0.2220
```

Review: Functions in MATLAB

I have presented three ways to define new functions in MATLAB. I will now review and compare these three mechanisms.

First of all, an expression in one or more variables can be used to represent a function. For example, to work with the function $f(x)=x^2$, I define

```
clear
syms x
f=x^2
f =
x^2
```

Now I can do whatever I wish with this function, including evaluate it, differentiate it, and integrate it. To evaluate **f**, I use the **subs** command:

```
subs(f,x,3)
ans =
     9
```

Differentiation and integration are performed with **diff** and **int**, respectively:

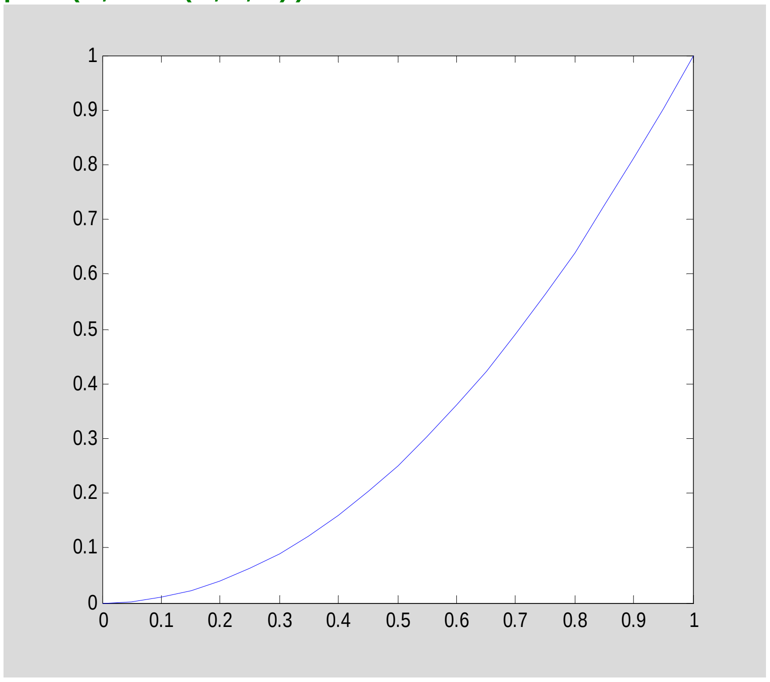
```
diff(f,x)
ans =
2*x
```

```
int(f,x)
ans =
x^3/3
```

As I have shown in earlier examples, a symbolic expression can be evaluated at a vector argument; that is, it is automatically vectorized:

```
t=linspace(0,1,21);
```

```
plot(t,subs(f,x,t))
```



The second way to define a function is using the **@** operator:

```
clear
f=@(x)x^2
f =
    @(x)x^2
```

An advantage of using a function is that functional notation is used for evaluation (instead of the **subs** command).

```
f(3)
ans =
     9
```

Functions can be evaluated with symbolic inputs:

```
syms a
f(a)
ans =
a^2
```

Symbolic calculus operations can be performed on functions indirectly, bearing in mind that **diff** and **int** operate on expressions, not functions.

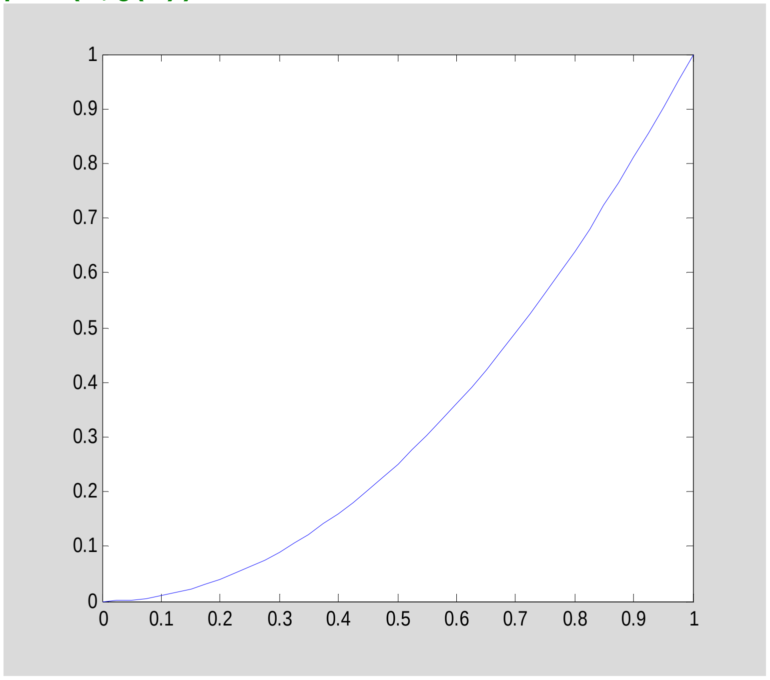
```
syms x
diff(f(x),x)
ans =
2*x
```

Here is an important fact to notice: functions are not automatically vectorized:

```
t=linspace(0,1,41);
plot(t,f(t))
??? Error using ==> mpower
Inputs must be a scalar and a square matrix.
To compute elementwise POWER, use POWER (.^) instead.

Error in ==> @(x)x^2
```

You can vectorize a function when you write it by using ".^", ".*", and "./".

```
g=@(x)x.^2;
plot(t,g(t))
```



Here is a common situation that I often face: Suppose we compute a complicated symbolic expression, perhaps using **diff** or another command, and we wish to make it into a function without retyping it. (We might wish to do this inside a script, for example, where it is not possible to retype the expression, or copy and paste it, because the calculations are being done automatically by the code.) Here is a trick: convert the expression to a string using the **char** command, construct a string containing the command to define the function, and then execute it using the **eval** command.

```
clear
syms x y
u=x^2*(1-x)*sin(pi*y)^2
u =
-x^2*sin(pi*y)^2*(x - 1)
expr=-diff(u,x,2)-diff(u,y,2)
expr =
2*sin(pi*y)^2*(x - 1) + 4*x*sin(pi*y)^2 + 2*pi^2*x^2*cos(pi*y)^2*(x -
1) - 2*pi^2*x^2*sin(pi*y)^2*(x - 1)
cmd=['f=@(x,y)',char(expr)]
cmd =
f=@(x,y)2*sin(pi*y)^2*(x - 1) + 4*x*sin(pi*y)^2 +
2*pi^2*x^2*cos(pi*y)^2*(x - 1) - 2*pi^2*x^2*sin(pi*y)^2*(x - 1)

eval(cmd)
f =
    @(x,y)2*sin(pi*y)^2*(x-
1)+4*x*sin(pi*y)^2+2*pi^2*x^2*cos(pi*y)^2*(x-1)-
2*pi^2*x^2*sin(pi*y)^2*(x-1)

f(.5,.75)
ans =
    0.5000
```

In the above calculation, notice that **"['f=@(x,y)',char(expr)]"** creates a character array---a string---that is the result of concatenating the two strings **'f=@(x,y)'** and **char(expr)**. Also note that single quotes are used to delimit an explicit string in MATLAB.

One problem with the previous example is that the resulting function **f** is not vectorized. This is easily remedied using the MATLAB command **vectorize**:

```
clear
syms x y
u=x^2*(1-x)*sin(pi*y)^2;
expr=-diff(u,x,2)-diff(u,y,2);
cmd=['f=@(x,y)',vectorize(char(expr))]
cmd =
f=@(x,y)2.*sin(pi.*y).^2.*(x - 1) + 4.*x.*sin(pi.*y).^2 +
2.*pi.^2.*x.^2.*cos(pi.*y).^2.*(x - 1) -
2.*pi.^2.*x.^2.*sin(pi.*y).^2.*(x - 1)
```

The third way to define a new function in MATLAB is to write an M-file. The major advantage of this method is that very complicated functions can be defined, in particular, functions that require several MATLAB commands to evaluate. You can review the section "Programming in MATLAB, part I" at this point, if necessary.

Chapter 4: Essential ordinary differential equations

Section 4.2: Solutions to some simple ODEs

Second-order linear homogeneous ODEs with constant coefficients

Suppose we wish to solve the following IVP:

$$\begin{aligned}
&\frac{d^2u}{dt^2}+4\frac{du}{dt}-3u=0,\ t>0,\\
&u(0)=1,\\
&\frac{du}{dt}(0)=0.
\end{aligned}$$

The characteristic polynomial is r^2+4r-3, which has the following roots:

```
clear
syms r
l=solve(r^2+4*r-3,r)
l =
   7^(1/2) - 2
 - 7^(1/2) - 2
```

The general solution of the ODE is then

```
syms t c1 c2
u=c1*exp(l(1)*t)+c2*exp(l(2)*t)
u =
c1*exp(t*(7^(1/2) - 2)) + c2/exp(t*(7^(1/2) + 2))
```

We can now solve for the unknown coefficients **c1**, **c2**:

```
c=solve(subs(u,t,sym(0))-1,subs(diff(u,t),t,sym(0)),c1,c2)
c =
    c1: [1x1 sym]
    c2: [1x1 sym]
c.c1,c.c2
ans =
7^(1/2)/7 + 1/2
ans =
(7^(1/2)*(7^(1/2) - 2))/14
```

The solution is found by substituting the correct values for **c1** and **c2**:

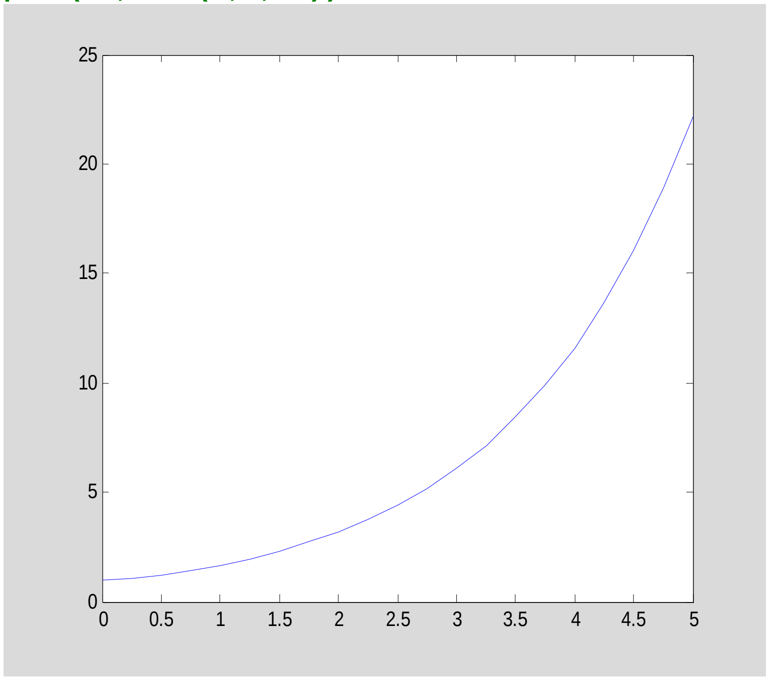
```
u=subs(u,{c1,c2},{c.c1,c.c2})
u =
exp(t*(7^(1/2) - 2))*(7^(1/2)/7 + 1/2) + (7^(1/2)*(7^(1/2) -
2))/(14*exp(t*(7^(1/2) + 2)))
```

Here is a better view of the solution:

```
pretty(u)

                    /  1/2       \      1/2   1/2
         1/2        | 7          |     7    (7    - 2)
  exp(t (7    - 2)) | ---- + 1/2 | + --------------------
                    \  7         /               1/2
                                      14 exp(t (7    + 2))
```

I can now plot the solution:

```
tt=linspace(0,5,21);
plot(tt,subs(u,t,tt))
```



A special inhomogeneous second-order linear ODE

Consider the IVP

$$\begin{aligned}&\frac{d^2u}{dt^2}+4u=\sin(\pi t),\ t>0,\\&u(0)=0,\\&\frac{du}{dt}(0)=0.\end{aligned}$$

The solution, as given in Section 4.2.3 of the text, is

```
clear
syms t s pi
(1/2)*int(sin(2*(t-s))*sin(pi*s),s,0,t)
ans =
-(2*sin(pi*t) - pi*sin(2*t))/(2*(pi^2 - 4))

u=simplify(ans)
u =
-(sin(pi*t) - (pi*sin(2*t))/2)/(pi^2 - 4)
pretty(u)

                 pi sin(2 t)
    sin(pi t) - -----------
                     2
```

```
  - -----------------------
              2
            pi   - 4
```

Let us check the solution:

```
diff(diff(u,t),t)+4*u
ans =
(pi^2*sin(pi*t) - 2*pi*sin(2*t))/(pi^2 - 4) - (4*(sin(pi*t) -
(pi*sin(2*t))/2))/(pi^2 - 4)
simplify(ans)
ans =
sin(pi*t)
```

The ODE is satisfied. How about the initial conditions?

```
subs(u,t,0)
ans =
     0
subs(diff(u,t),t,0)
ans =
     0
```

The initial conditions are also satisfied.

First-order linear ODEs

Now consider the following IVP:

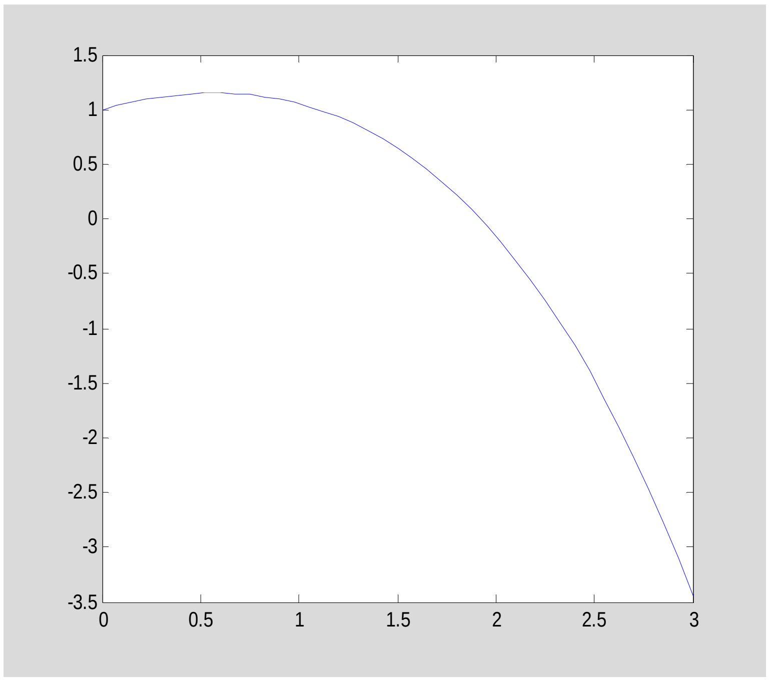
$$\frac{du}{dt}-\frac{1}{2}u=-t,\, t>0,$$
$$u(0)=1.$$

Section 4.2.4 contains an explicit formula for the solution:

```
clear
syms t s
exp(t/2)+int(exp((t-s)/2)*(-s),s,0,t)
ans =
2*t - 3*exp(t/2) + 4

u=ans
u =
2*t - 3*exp(t/2) + 4
```

Here is a graph of the solution:

```
tt=linspace(0,3,41);
plot(tt,subs(u,t,tt))
```



Just out of curiosity, let us determine the value of *t* for which the solution is zero.

```
solve(u,t)
ans =
- 2*lambertw(0, -3/(4*exp(1))) - 2
```

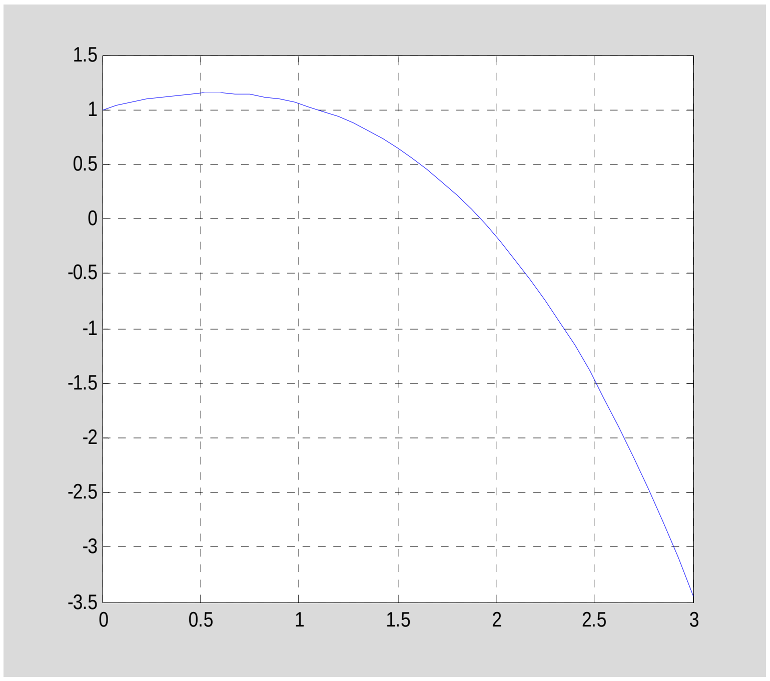
The **solve** command finds a solution and expresses in terms of the Lambert W function (see "**help lambertw**" for details).   We can convert the result to floating point:

```
double(ans)
ans =
   -1.1603
```

We see that, in this case, **solve** did not succeed in finding the desired root.

As an alternative to **solve**, MATLAB provides a command, **fzero**, that looks for a single root numerically. To use it, you must have an estimate of the desired root. To make it easier to find such an estimate, I will add a grid to the previous graph using the **grid** command:

```
grid
```



From the graph, we see that the desired root is a little less than 2. **fzero** requires a function, not just an expression, so I convert the expression **u** into a function:

```
eval(['u=@(t)',char(u)])
u =
    @(t)2*t-3*exp(t/2)+4
```

Now I call **fzero**, using 2.0 as the initial estimate of the root:

```
fzero(u,2.0)
ans =
    1.9226
```

Let's check the root:

```
u(ans)
ans =
 -8.8818e-016
```

The solution appears to be highly accurate.

Section 4.3: Linear systems with constant coefficients

Since MATLAB can compute eigenvalues and eigenvectors (either numerically or, when possible, symbolically), it can be used to solve linear systems with constant coefficients. I will begin with a simple example, solving the homogeneous IVP

$$\frac{dx}{dt} = Ax,\, t > 0,$$
$$x(0) = x_0,$$

where A and x_0 have the values given below. Notice that I define A and x_0 to be symbolic, and hence use symbolic computations throughout this example.

```
clear
A=sym([1 2;3 4])
A =
[ 1, 2]
[ 3, 4]
x0=sym([4;1])
x0 =
 4
 1
```

The first step is to find the eigenpairs of **A**:

```
[V,D]=eig(A)
V =
[ - 33^(1/2)/6 - 1/2, 33^(1/2)/6 - 1/2]
[                  1,                1]
D =
[ 5/2 - 33^(1/2)/2,                0]
[                0, 33^(1/2)/2 + 5/2]
```

Notice that the matrix **A** is not symmetric, and so the eigenvectors are not orthogonal. However, they are linearly independent, and so I can express the initial vector as a linear combination of the eigenvectors. The coefficients are found by solving the linear system **Vc=x0**:

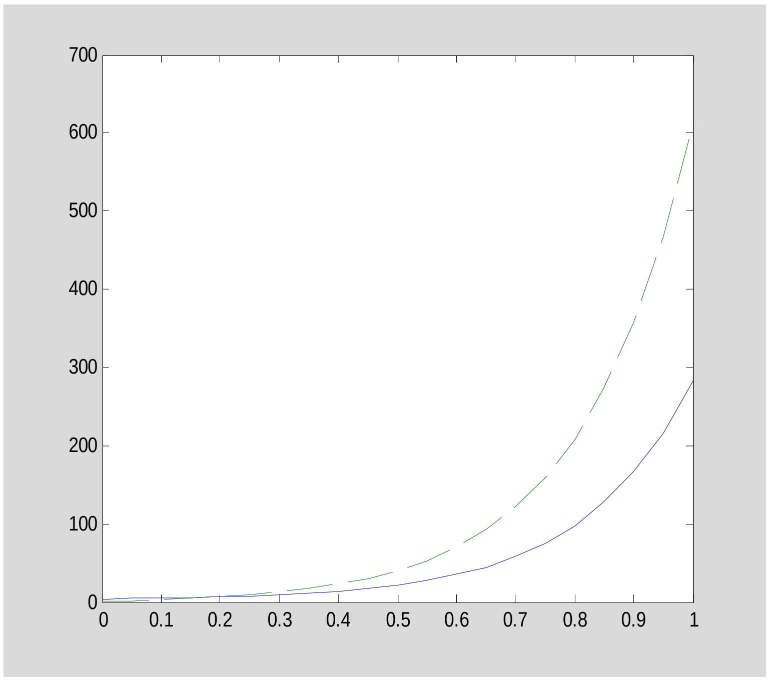
```
c=V\x0
c =
         1/2 - (9*33^(1/2))/22
 (33^(1/2)*(33^(1/2) + 27))/66
```

Now I can write down the solution:

```
syms t
x=c(1)*exp(D(1,1)*t)*V(:,1)+c(2)*exp(D(2,2)*t)*V(:,2)
x =
 ((33^(1/2)/6 + 1/2)*((9*33^(1/2))/22 - 1/2))/exp(t*(33^(1/2)/2 - 5/2))
+ (33^(1/2)*exp(t*(33^(1/2)/2 + 5/2))*(33^(1/2)/6 - 1/2)*(33^(1/2) +
27))/66
                                         (33^(1/2)*exp(t*(33^(1/2)/2 +
5/2))*(33^(1/2) + 27))/66 - ((9*33^(1/2))/22 - 1/2)/exp(t*(33^(1/2)/2 -
5/2))
```

Here are the graphs of the two components of the solutions:

```
tt=linspace(0,1,21);
plot(tt,subs(x(1),t,tt),'-',tt,subs(x(2),t,tt),'--')
```



Both components are dominated by a rapidly growing exponential, as a careful examination of the formulas confirms.

Inhomogeneous systems and variation of parameters

I will now show how to use MATLAB to solve the inhomogeneous system

$$\frac{dx}{dt} = Ax + f(t), t > 0,$$
$$x(0) = x_0.$$

Consider the matrix

```
clear
A=sym([1 2;2 1])
A =
[ 1, 2]
[ 2, 1]
```

and let

```
syms t
f=[sin(t);0]
f =
 sin(t)
      0
x0=sym([0;1])
x0 =
 0
 1
```

Notice that the matrix **A** is symmetric. First, I find the eigenvalues and eigenvectors of **A**:

```
[V,D]=eig(A)
V =
[ -1, 1]
[  1, 1]
D =
[ -1, 0]
[  0, 3]
```

When operating on a symbolic matrix, MATLAB does not normalize the eigenvectors, so I normalize them and call them **v1** and **v2**:

```
v1=V(:,1)/sqrt(V(:,1)'*V(:,1))
v1 =
 -2^(1/2)/2
  2^(1/2)/2
v2=V(:,2)/sqrt(V(:,2)'*V(:,2))
v2 =
 2^(1/2)/2
 2^(1/2)/2
```

For convenience, I will call the eigenvalues **l1** and **l2**:

```
l1=D(1,1)
l1 =
-1
l2=D(2,2)
l2 =
3
```

We have **f(t)=c1(t)v1+c2(t)v2** and **x0=b1*v1+b2*v2**, where

```
c1=v1'*f
c1 =
-(2^(1/2)*sin(t))/2
c2=v2'*f
c2 =
(2^(1/2)*sin(t))/2
b1=v1'*x0
b1 =
2^(1/2)/2
b2=v2'*x0
b2 =
2^(1/2)/2
```

I now solve the two decoupled IVPs

$$\frac{da_1}{dt} = l_1 a_1 + c_1(t),\ a_1(0) = b_1,$$
$$\frac{da_2}{dt} = l_2 a_2 + c_2(t),\ a_2(0) = b_2$$

using the methods of Section 4.2. The solution to the first is

```
syms s
a1=b1*exp(l1*t)+int(exp(l1*(t-s))*subs(c1,t,s),s,0,t)
a1 =
(2^(1/2)*cos(t))/4 - (2^(1/2)*sin(t))/4 + 2^(1/2)/(4*exp(t))
```

(Notice how I used the **subs** command to change the variable in the expression for **c1** from **t** to **s**.)
The solution to the second IVP is

```
a2=b2*exp(l2*t)+int(exp(l2*(t-s))*subs(c2,t,s),s,0,t)
a2 =
(11*2^(1/2)*exp(3*t))/20 - (3*2^(1/2)*sin(t))/20 - (2^(1/2)*cos(t))/20
```

The solution to the original system is then

```
x=simplify(a1*v1+a2*v2)
x =
 (11*exp(3*t))/20 - 1/(4*exp(t)) - (3*cos(t))/10 + sin(t)/10
   1/(4*exp(t)) + (11*exp(3*t))/20 + cos(t)/5 - (2*sin(t))/5
```

Let us check this result:

```
diff(x,t)-A*x-f
ans =
 0
 0
```

Thus we see that the ODE is satisfied. We also have

```
subs(x,t,sym(0))-x0
ans =
 0
 0
```

so the initial condition is satisfied as well.

Programming in MATLAB, Part II

In preparation for the next section, in which I discuss the implementation of numerical methods for IVPs in MATLAB, I want to present more of the mechanisms for programming in MATLAB. In an earlier section of this tutorial, I explained how to create a new function in an M-file. An M-file function need not implement a simple mathematical function $f(x)$. Indeed, a common use of M-files is to implement algorithms, in which case the M-file should be thought of as a subprogram.

To program any nontrivial algorithm requires the common program control mechanisms for implementing loops and conditionals.

Loops in MATLAB

Here is an example illustrating an indexed loop:

```
for jj=1:4
   disp(jj^2)
end
     1
     4
     9
    16
```

This example illustrates the **for** loop in MATLAB, which has the form

```
for j=j1:j2
    statement1
    statement2
    …
    statementn
end
```

The sequence of statements statement$_1$, statement$_2$,…,statement$_n$ is executed j_2-j_1+1 times, first with j=j_1, then with j=j_1+1, and so forth until j=$\mathbf{j_2}$.

MATLAB also has a **while** loop, which executes as long as a given logical condition is true. I will not need the **while** loop in this tutorial, so I will not discuss it. The interested reader can consult "**help while**".

Conditional execution

The other common program control mechanism that I will often use in this tutorial is the **if-elseif-else** block. I already used it once (see the M-file **f2.m** above). The general form is

```
if condition1
    statement(s)
elseif condition2
    statements(s)
.
.
.
elseif conditionk
    statement(s)
else
    statement(s)
end
```

When an **if-elseif-else** block is executed, MATLAB checks whether condition$_1$ is true; if it is, the first block of statements is executed, and then the program proceeds after the end statement. If condition$_1$ is false, MATLAB checks condition$_2$, condition$_3$, and so on until one of the conditions is true. If conditions 1 through k are all false, then the else branch executes.

Logical conditions in MATLAB evaluate to 0 or 1:

```
abs(0.5)<1
ans =
     1
abs(1.5)<1
ans =
     0
```

Therefore, the condition in an **if** or **elseif** statement can be any expression with a numerical value. MATLAB regards a nonzero value as true, and zero as false.

To form complicated conditions, one can use the logical operators == (equality), && (logical and), and || (logical or). Here are some examples:

```
clear
x=0.5
x =
    0.5000

if x<-1 || x>1 % true if x is less than -1 or x is greater than 1
   disp('True!')
else
   disp('False!')
end
False!

if x>-1 && x<1  % true if x is between -1 and 1
   disp('True!')
else
   disp('False!')
end
True!
```

Passing one function into another

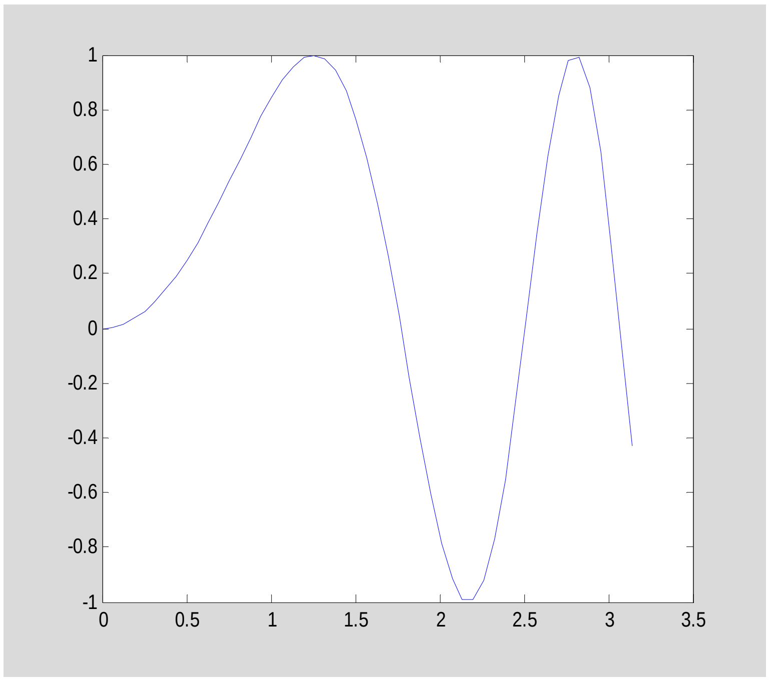
Often a program (which in MATLAB is a function) takes as input a function. For example, suppose I wish to write a program that will plot a given function *f* on a given interval [*a*,*b*]. (This is just for the purpose of illustration, since the program I produce below will not be much easier to use than the **plot** command itself.)
The function is written in the obvious fashion; when calling the function, a function handle is created and passed.

Here is the desired program, which I will call **myplot**:

```
type myplot

function myplot(f,a,b)

x=linspace(a,b,51);
y=f(x);
plot(x,y)
```

To call this function, a function handle is given as the first input argument. For example, to plot the function **g(x)=sin(x^2)** , I first define the function **g** and then call myplot in the obvious fashion:

```
clear
g=@(x)sin(x.^2);
myplot(g,0,pi)
```



Now recall the function **f** that I defined earlier:

```
type f

function y=f(x)

y=sin(x.^2);
```

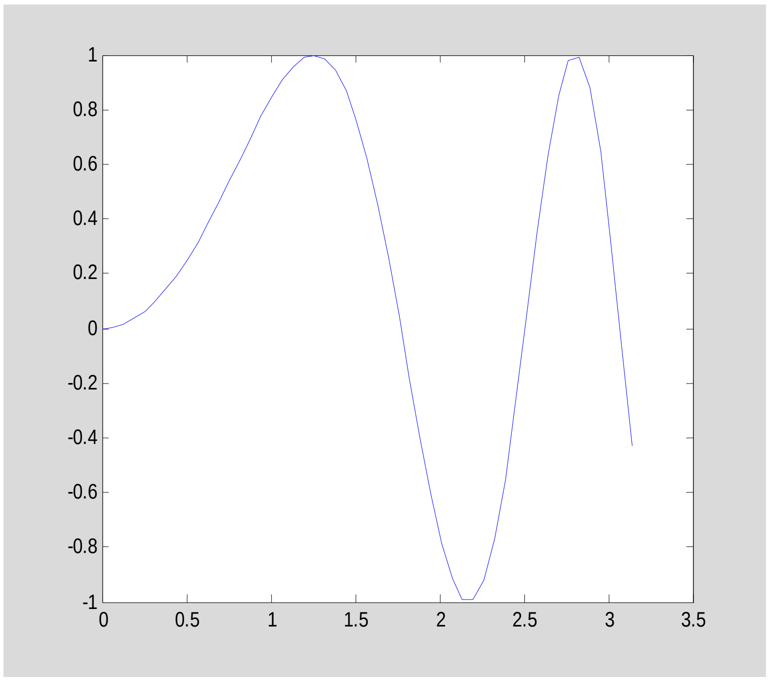
One might be tempted to call **myplot** like this:

```
myplot(f,0,pi)
??? Input argument "x" is undefined.

Error in ==> f at 3
y=sin(x.^2);
```

The above error occurs before MATLAB reaches the executable statements in **myplot.m**. In parsing the command "**myplot(f,0,pi)**", MATLAB must evaluate all of the inputs, so that it can pass their values to the M-file **myplot1.m**. However, in trying to evaluate **f**, MATLAB calls the M-file **f.m** and encounters an error since the input to **f** has not been given.

To correctly pass **f** to **myplot**, we must create a function handle referring to **f.m**:

```
f1=@f
f1 =
    @f
myplot(f1,0,pi)
```



Now the code works fine.

Section 4.4: Numerical methods for initial value problems

Now I will show how to implement numerical methods, such as those discussed in the text, in MATLAB programs. I will focus on Euler's method; you should easily be able to extend these techniques to implement a Runge-Kutta method.

Consider the IVP

$$\frac{du}{dt} = f(t,u),\ t > t_0,$$
$$u(t_0) = u_0.$$

A program that applies Euler's method to this problem should take, as input, f, t_0, and u_0, as well as information that allows it to determine the step size and the number of steps, and return, as output, the approximate solution on the resulting grid. I will assume that the user will specify the time interval $[t_0,t_n]$ and the number of steps n (then the program can compute the time step).

As an example, I will solve the IVP

$$\frac{du}{dt} = \frac{u}{1+t^2},\ t > 0,$$
$$u(0) = 1.$$

on the interval [0,10]. I will do it first interactively, and then collect the commands in an M-file.

The first step is to define the grid and allocate space to save the computed solution. I will use 100 steps, so the grid and the solution vector will each contain 101 components ($t_0,t_1,t_2,\ldots,t_{100}$ and $u_0,u_1,u_2,\ldots,u_{100}$). (It is convenient to store the times $t_0,t_1,t_2,\ldots,t_{100}$ in a vector for later use, such as graphing the solution.)

```
clear
t=linspace(0,10,101);
u=zeros(1,101);
```

Now I assign the initial value and compute the time step:

```
u(1)=1;
dt=10/100;
```

Next I define the function that forms the right-hand side of the ODE:

```
f=@(t,u)u/(1+t^2)
f =
    @(t,u)u/(1+t^2)
```

Euler's method is now implemented in a single loop:

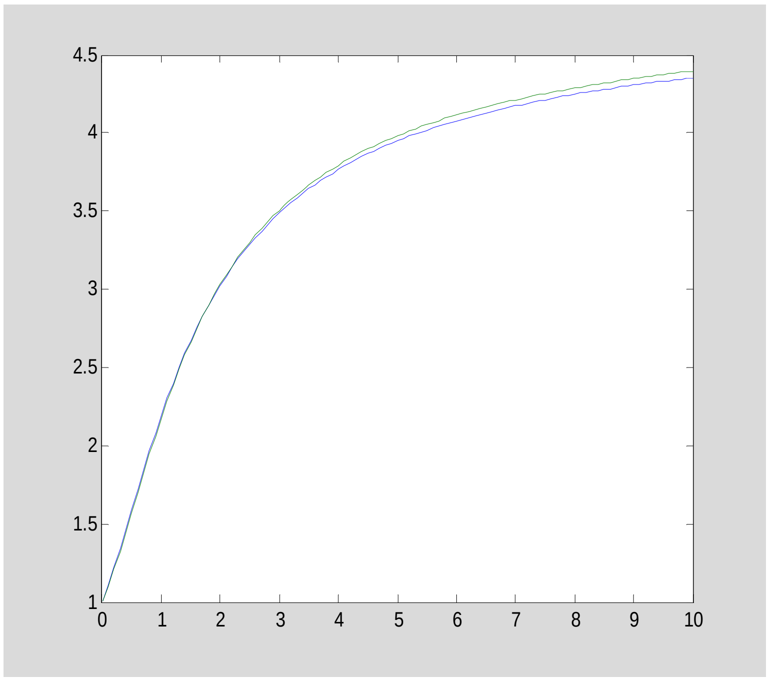
```
for ii=1:100
  u(ii+1)=u(ii)+dt*f(t(ii),u(ii));
end
```

The exact solution is $u(t)=\exp(\tan^{-1}(t))$:

```
 U=@(t)exp(atan(t))
U =
    @(t)exp(atan(t))
```

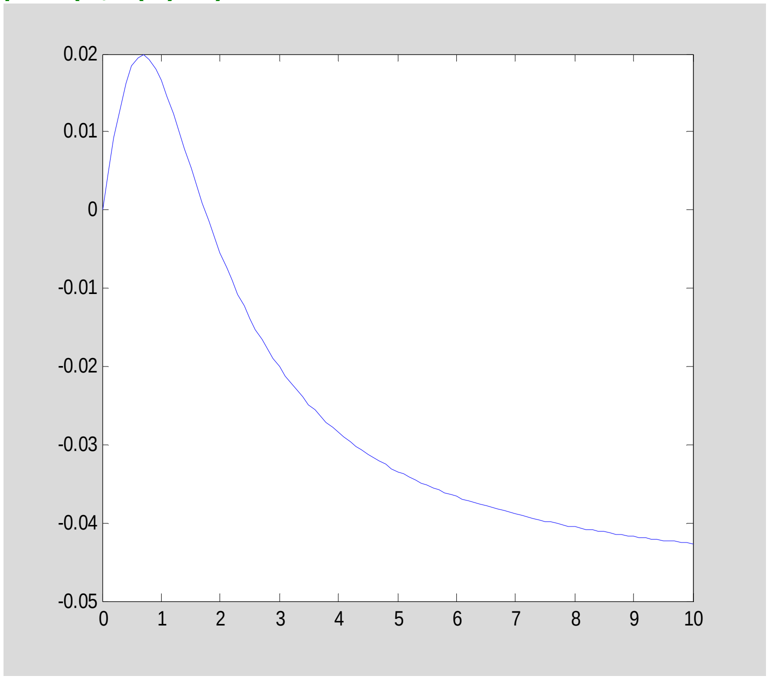
Here is a graph of the exact solution and the computed solution:

```
plot(t,U(t),t,u)
```



The computed solution is not too different from the exact solution. As is often the case, it is more informative to graph the error:

```
plot(t,U(t)-u)
```



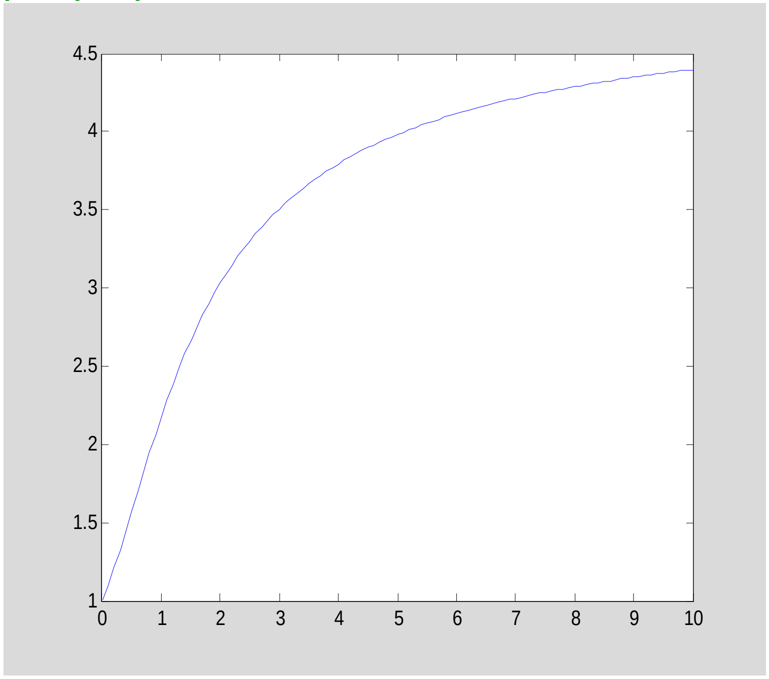
It is now easy to gather the above steps in an M-file that implements Euler's method:

```
type euler1
```

```
function [u,t]=euler1(f,t0,tf,u0,n)

%[u,t]=euler1(f,t0,tf,u0,n)
%
%   This function implements Euler's method for solving the IVP
%
%         du/dt=f(t,u), u(t0)=u0
%
%   on the interval [t0,tf].  n steps of Euler's method are taken;
%   the step size is dt=(tf-t0)/n.

% Compute the grid and allocate space for the solution

t=linspace(t0,tf,n+1);
u=zeros(1,n+1);

% Assign the initial value and compute the time step

u(1)=u0;
dt=(tf-t0)/n;

% Now do the computations in a loop

for ii=1:n
   u(ii+1)=u(ii)+dt*f(t(ii),u(ii));
end
```

The entire computation I did above now requires a single line:

```
[u,t]=euler1(f,0,10,1,100);
plot(t,u)
```



Vectorizing the euler program

Euler's method applies equally well to systems of ODEs, and the **euler1** program requires only minor changes to handle a system. Mainly we just need to take into account that, at each time t_i, the computed solution u_i is a vector. Here is the program, which works for either scalar or vector systems (since a scalar can be viewed as a vector of length 1):

```
type euler2

function [u,t]=euler2(f,t0,tf,u0,n)

%[u,t]=euler2(f,t0,tf,u0,n)
%
%   This function implements Euler's method for solving the IVP
%
%         du/dt=f(t,u), u(t0)=u0
%
%   on the interval [t0,tf].  n steps of Euler's method are taken;
%   the step size is dt=(tf-t0)/n.
%
%   The solution u can be a scalar or a vector.  In the vector case,
%   the initial value must be a kx1 vector, and the function f must
%   return a kx1 vector.

% Figure out the dimension of the system by examining the initial
value:

k=length(u0);

% Compute the grid and allocate space for the solution

t=linspace(t0,tf,n+1);
u=zeros(k,n+1);

% Assign the initial value and compute the time step

u(:,1)=u0;
dt=(tf-t0)/n;

% Now do the computations in a loop

for ii=1:n
   u(:,ii+1)=u(:,ii)+dt*f(t(ii),u(:,ii));
end
```

You should notice the use of the **length** command to determine the number of components in the initial value **u0**. A related command is **size**, which returns the dimensions of an array:

```
A=randn(4,3);
size(A)
ans =
     4     3
```

The length of an array is defined simply to be the maximum dimension:

```
length(A)
```

```
ans =
     4
max(size(A))
ans =
     4
```

The only other difference between **euler1** and **euler2** is that the vector version stores each computed value u_i as a column in a matrix.

As an example of solving a system, I will apply Euler's method to the system

$$\frac{du_1}{dt} = u_2, u_1(0) = 0,$$
$$\frac{du_2}{dt} = -u_1, u_2(0) = 1.$$

The right-hand side of the system is defined by the vector-valued function

$$f(t,u) = \begin{bmatrix} u_2 \\ -u_1 \end{bmatrix},$$

which I define as follows:

```
f=@(t,u)[u(2);-u(1)]
f =
    @(t,u)[u(2);-u(1)]
```

Notice that the function must return a column vector, not a row vector. Notice also that, even though this particular function f is independent of t, I wrote **euler2** to expect a function $f(t,u)$ of two variables. Therefore, I had to define f above as a function of t and u, even though it is constant with respect to t.

The initial value is

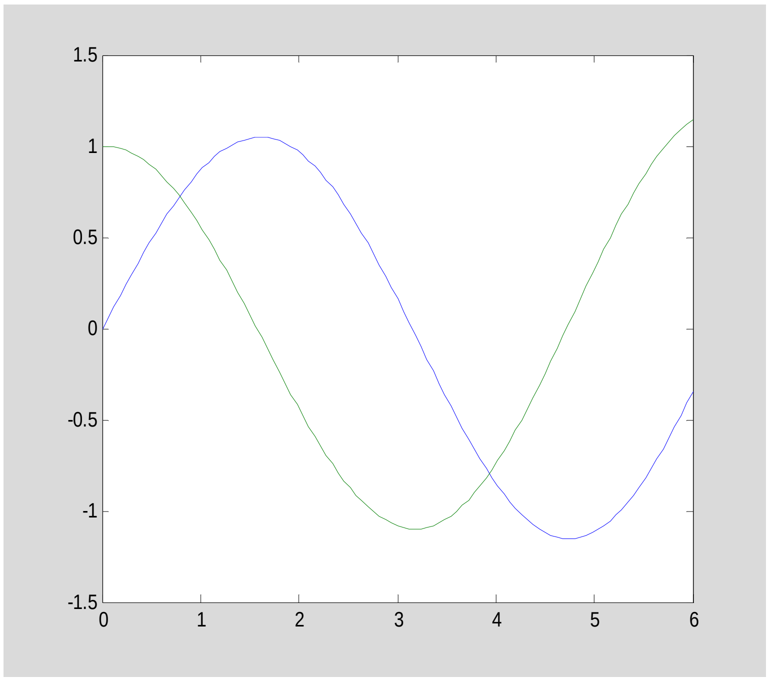
```
u0=[0;1];
```

I will solve the system on the interval [0,6].

```
[u,t]=euler2(f,0,6,u0,100);
```

Here I plot both components of the solution:

```
plot(t,u(1,:),t,u(2,:))
```



Programming in MATLAB, part III

There is a subtle improvement we can make in the above program implementing Euler's method. Often a function, which is destined to be input to a program like **euler2**, depends not only on independent variables, but also on one or more parameters. For example, suppose I wish to solve the following IVP for a several different values of *a*:

$$\frac{du}{dt} = au,\ u(0) = 1.$$

One way to handle this is to define a different function *f* for each different value of *a*:

```
f1=@(t,u)u
f1 =
    @(t,u)u
f2=@(t,u)2*u
f2 =
    @(t,u)2*u
```

(and so forth). However, this is obviously tedious and inelegant. A better technique is to make *f* depend directly on the parameter *a*; in effect, make *f* a function of three variables:

```
clear
f=@(t,u,a)a*u
f =
    @(t,u,a)a*u
```

The question is: How can a program implementing Euler's method recognize when the function f depends on one or more parameters? The answer lies in taking advantage of MATLAB's ability to count the number of arguments to a function. Recall that, inside an M-file function, the **nargin** variable records the number of inputs. The program can then do different things, depending on the number of inputs. (I already showed an example of the use of **nargin** above in the first section on MATLAB programming.)

Here is an M-file function implementing Euler's method and allowing for optional parameters:

```
type euler

function [u,t]=euler(f,t0,tf,u0,n,varargin)

%[u,t]=euler(f,t0,tf,u0,n,p1,p2,...)
%
%   This function implements Euler's method for solving the IVP
%
%         du/dt=f(t,u), u(t0)=u0
%
%   on the interval [t0,tf].  n steps of Euler's method are taken;
%   the step size is dt=(tf-t0)/n.
%
%   If the optional arguments p1,p2,... are given, they are passed
%   to the function f.

% Figure out the dimension of the system by examining the initial
value:

k=length(u0);

% Compute the grid and allocate space for the solution

t=linspace(t0,tf,n+1);
u=zeros(k,n+1);

% Assign the initial value and compute the time step

u(:,1)=u0;
dt=(tf-t0)/n;

% Now do the computations in a loop

for ii=1:n
   u(:,ii+1)=u(:,ii)+dt*f(t(ii),u(:,ii),varargin{:});
end
```

There are only two differences between **euler2** and **euler**. The new program accepts an additional input, **varargin**, and it passes this argument, in the form **varargin{:}**, to f. The symbol **varargin** is a special variable in MATLAB (like **nargin**) that is used only in M-file functions. It is initialized, when the M-file is invoked, to contain any additional arguments beyond those explicitly listed in the function statement. These additional arguments, if they are provided, are stored in a cell array---an indexed array of possibly different types.

I do not wish to explain cell arrays in any detail, since I do not need them in this tutorial except in this one context. It is enough to know that **varargin{:}** turns the fields of **varargin** into a list that can be passed to another function. Moreover, if there were no additional parameters when the M-file was invoked, the **varargin** is empty and passing **varargin{:}** to another function has absolutely no effect.

I will now do an example, solving

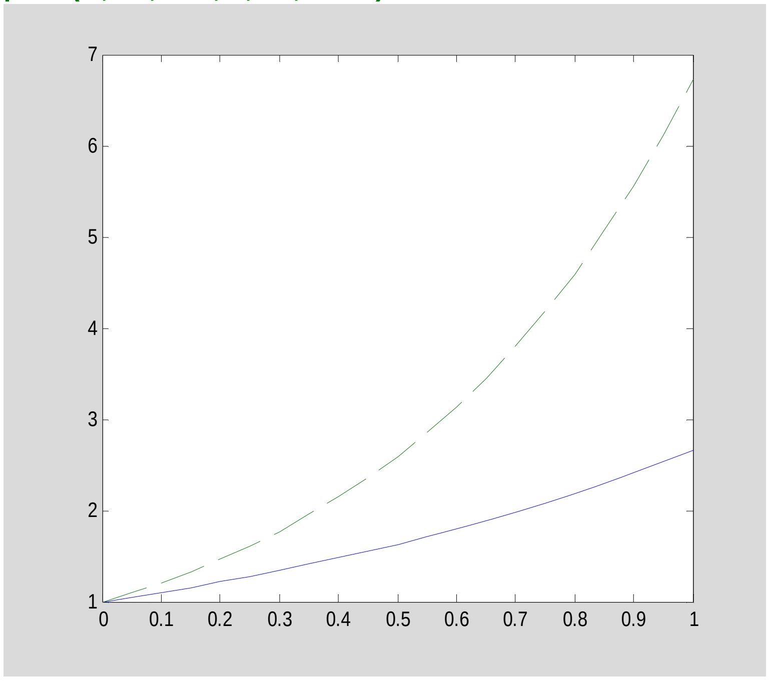
$$\frac{du}{dt} = au, u(0) = 1$$

on the interval [0,1] for two different values of *a*.

```
clear
f=@(t,u,a)a*u;
[u1,t]=euler(f,0,1,1,20,1);
[u2,t]=euler(f,0,1,1,20,2);
```

Now I plot the two solutions:

```
plot(t,u1,'-',t,u2,'--')
```



By the way, although the style of **euler** is the "right" way to implement a function that takes a mathematical function as an input, there is another way to accomplish the desired end. Suppose we have a function like **euler2** that does not have the flexibility of **euler**; specifically, it expects a function of the form f(t,u) and does not allow for an additional parameter, as in f(t,u,a). We can use the **@** operator to define, for example, f1(t,u)=f(t,u,1), and then pass f1 to **euler2**:

```
f1=@(t,u)f(t,u,1);
[u1a,t]=euler2(f1,0,1,1,20);
norm(u1a-u1)
ans =
     0
```

The result is now exactly the same as before.

Note: The point of the textbook and this tutorial is for you to learn how numerical methods work, and how to implement them in MATLAB. However, if you have an ODE to solve and need a numerical solution, you should be aware that MATLAB includes state-of-the-art algorithms implemented in **ode45** and other routines. In particular, **ode45** implements a variable step fourth-fifth order scheme, similar in spirit to the RKF45 scheme mentioned in the text (page 119), though using a different pair of formulas. MATLAB also has solvers, such as **ode23s**, for stiff systems. For more information, see "**help ode45**" or "**help ode23s**" (the documentation contains a list of other related routines).

Efficient MATLAB programming

Programs written in the MATLAB programming language are interpreted, not compiled. This means that you pay a certain performance penalty for using the MATLAB interactive programming environment instead of programming in a high-level programming language like C or Fortran. However, you can make your MATLAB programs run very fast, in many cases, by adhering to the following simple rule: whenever possible, use MATLAB commands, especially vectorization, rather then user-defined code.

Here is a simple example. Suppose I wish to evaluate the sine function on the grid

0, 0.000001, 0.000002, ... ,1.0

(1,000,001 points). Below are two ways to accomplish this; I time each using the "tic-toc" commands (see "help tic" for more information).

```
clear
x=linspace(0,1,1000001);
tic
y=sin(x);
toc
Elapsed time is 0.008175 seconds.

tic
y=zeros(1000001,1);
for i=1:1000001,y(i)=sin(x(i));end
toc
Elapsed time is 1.151249 seconds.
```

The version using an explicit loop is much more time-consuming than the version using MATLAB vectorization. This is because the vectorized commands use compiled code, while the explicit loop must be interpreted.

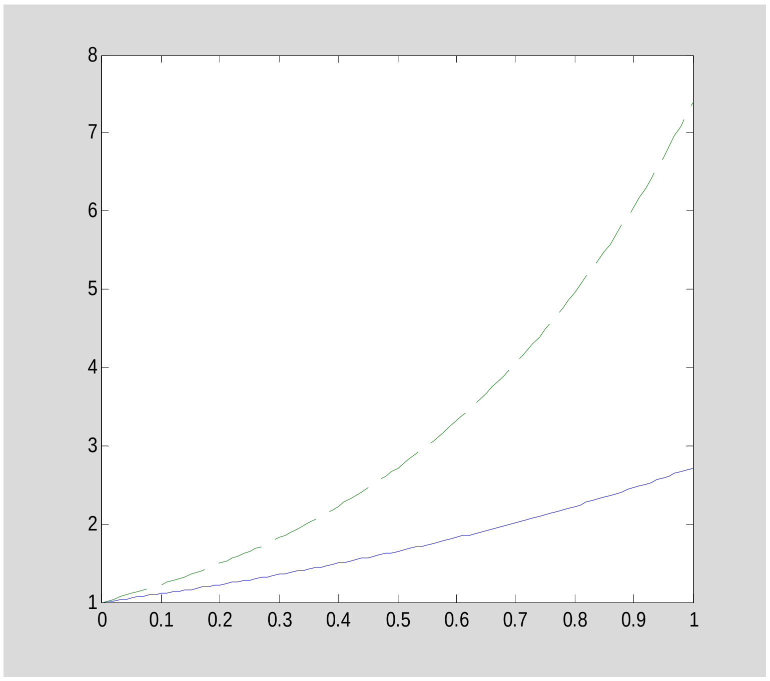
MATLAB commands use state-of-the-art algorithms and execute efficiently. Therefore, when you solve linear systems, evaluate functions, sort vectors, and perform other operations using MATLAB commands, your code will be very efficient. On the other hand, when you execute explicit loops, you pay a performance penalty. For many purposes, the convenience of MATLAB far outweighs any increase in computation time.

More about graphics in MATLAB

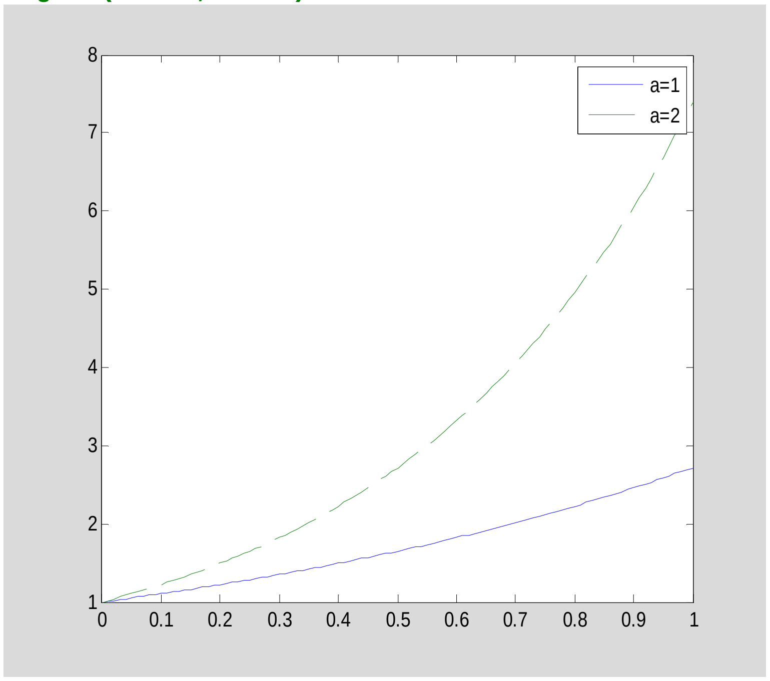
Adding a legend to a plot

When graphing two curves on the same plot, it is often helpful to label them so they can be distinguished. MATLAB has a command called **legend**, which adds a key with user-defined descriptions of the curves. The format is simple: just list the descriptions (as strings) in the same order as the curves were specified to the **plot** command:

```
x=linspace(0,1,101)';
plot(x,exp(x),'-',x,exp(2*x),'--')
```



```
legend('a=1','a=2')
```



It is possible to tell MATLAB where on the plot to place the legend; see "**help legend**" for more information.

You can also add a title and labels for the axes to a graph using **title**, **xlabel**, and **ylabel**. Use **help** to get more information.

Changing the font size, line width, and other properties of graphs

It is possible to increase the size of fonts, the widths of curves, and otherwise modify the default properties of MATLAB graphics objects. I do not want to explain how to do this here, but I wanted to make you aware of the fact that this can be done. You can find out how to modify default properties from the documentation on graphics, available through the help browser on the MATLAB desktop. Alternatively, you can consult "**help set**" and "**help plot**" to get started.

Chapter 5: Boundary value problems in statics

Section 5.2: Introduction to the spectral method; eigenfunctions

I will begin by verifying that the eigenfunctions of the negative second derivative operator (under Dirichlet conditions), sin($n\pi x/l$), are mutually orthogonal:

```
clear
syms m n pi x l
int(sin(n*pi*x/l)*sin(m*pi*x/l),x,0,l)
ans =
-(l*(m*cos(pi*m)*sin(pi*n) - n*cos(pi*n)*sin(pi*m)))/(pi*(m^2 - n^2))
simplify(ans)
ans =
-(l*(m*cos(pi*m)*sin(pi*n) - n*cos(pi*n)*sin(pi*m)))/(pi*(m^2 - n^2))
```

At first glance, this result is surprising: Why did MATLAB not obtain the expected result, 0? However, a moment's thought reveals the reason: The integral is not necessarily zero unless m and n are integers, and MATLAB has no way of knowing that the symbols m and n are intended to represent integers. When m is an integer, then $\sin(m\pi)=0$ and $\cos(m\pi)=(-1)^m$. We can use the **subs** command to replace each instance of $\sin(m\pi)$ by zero, and similarly for $\cos(m\pi)$, $\sin(n\pi)$, and $\cos(n\pi)$:

```
subs(ans,sin(m*pi),0)
ans =
-(l*m*cos(pi*m)*sin(pi*n))/(pi*(m^2 - n^2))
subs(ans,cos(m*pi),(-1)^m)
ans =
-((-1)^m*l*m*sin(pi*n))/(pi*(m^2 - n^2))
subs(ans,sin(n*pi),0)
ans =
0
```

These substitutions are rather tedious to apply, but they are often needed when doing Fourier series computations. Therefore, I wrote a simple program to apply them:

```
type mysubs

function expr=mysubs(expr,varargin)

%expr=mysubs(expr,m,n,...)
%
%  This function substitutes 0 for sin(m*pi) and (-1)^m
%  for cos(m*pi), and similarly for sin(n*pi) and cos(n*pi),
%  and any other symbols given as inputs.

syms pi
```

```
k=length(varargin);
for j=1:k
   expr=subs(expr,sin(varargin{j}*pi),0);
   expr=subs(expr,cos(varargin{j}*pi),(-1)^varargin{j});
end
expr=simplify(expr);
```

I will use **mysubs** often to simplify Fourier coefficient calculations.

Example 5.5

Let *f*(*x*) be defined as follows:

```
clear
syms x
f=x*(1-x)
f =
-x*(x - 1)
```

I can easily compute the Fourier sine coefficients of *f* on the interval [0,1]:

```
syms n pi
2*int(f*sin(n*pi*x),x,0,1)
ans =
(2*(4*sin((pi*n)/2)^2 - pi*n*sin(pi*n)))/(pi^3*n^3)
```

I simplify the result using **mysubs**:

```
a=mysubs(ans,n)
a =
(8*sin((pi*n)/2)^2)/(pi^3*n^3)
```

Here is the coefficient:

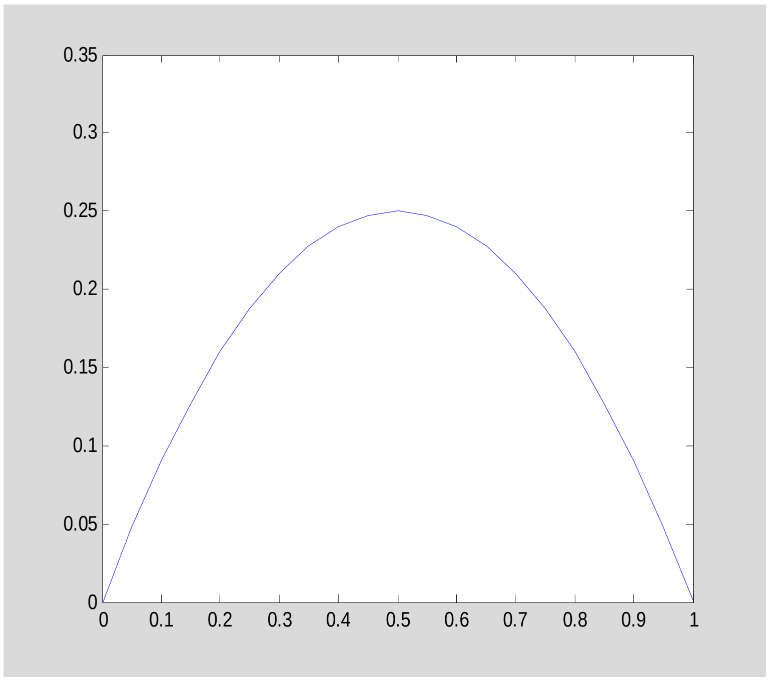
```
pretty(a)

       / pi n \2
  8 sin| ---- |
       \  2   /
  --------------
        3  3
      pi  n
```

Using the **symsum** (symbolic summation) command, I can now create a partial Fourier sine series with a given number of terms. For example, suppose I want the Fourier sine series with *n*=1,2,…,10. Here it is:

```
S=symsum(a*sin(n*pi*x),n,1,10)
S =
(8*sin(pi*x))/pi^3 + (8*sin(3*pi*x))/(27*pi^3) +
(8*sin(5*pi*x))/(125*pi^3) + (8*sin(7*pi*x))/(343*pi^3) +
(8*sin(9*pi*x))/(729*pi^3)
```

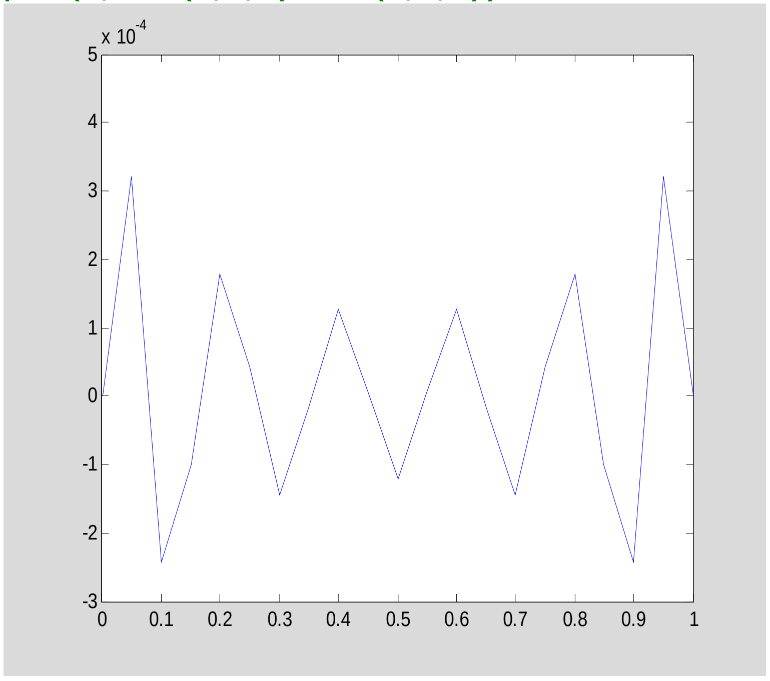
I can plot the partial sum:

```
t=linspace(0,1,21);
plot(t,subs(S,x,t))
```



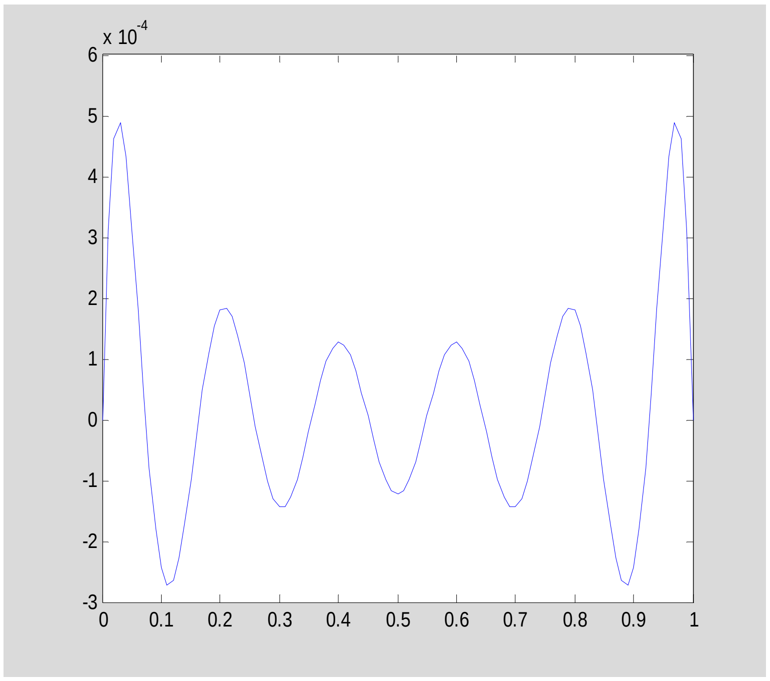
I can also plot the error in **S** as an approximation to **f**:

```
plot(t,subs(f,x,t)-subs(S,x,t))
```



The error looks jagged because I chose a coarse grid on which to perform the computations. Here is a more accurate graph:

```
t=linspace(0,1,101);
plot(t,subs(f,x,t)-subs(S,x,t))
```

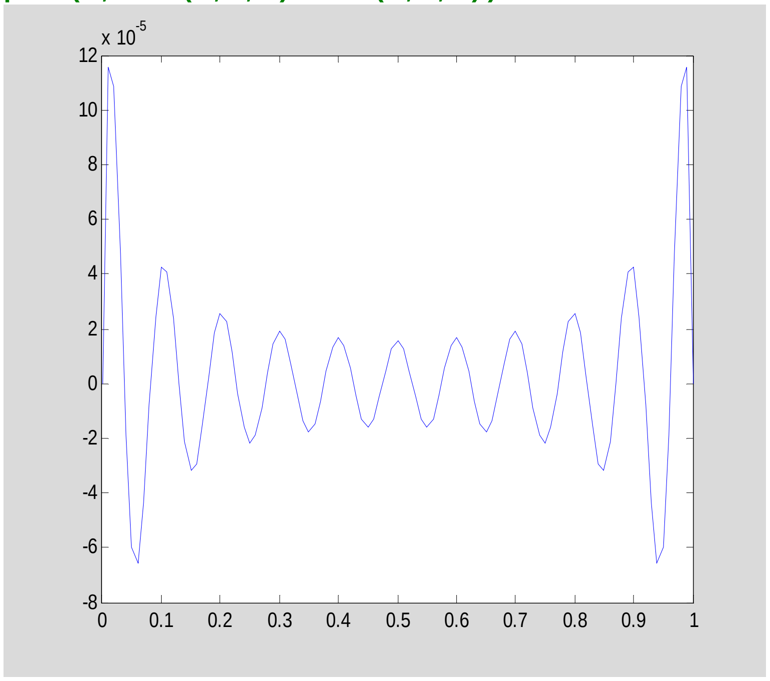


I can also investigate how the error decreases as the number of terms in the Fourier series is increased. For example, here is the partial Fourier series with 20 terms:

```
S=symsum(a*sin(n*pi*x),n,1,20);
```

Here is the error in **S** as an approximation to *f*:

```
plot(t,subs(f,x,t)-subs(S,x,t))
```



A few notes about symsum

The **symsum** command has the form "**symsum(expr,ii,m,n)**", where **ii** must be a symbolic variable without an assigned value. If **ii** has previously been assigned a value, then the above command will not work. Here are some errors to be avoid:

This does not work because **ii** is not symbolic:

```
ii=1;
symsum(ii,ii,1,10)
??? Undefined function or method 'symsum' for input arguments of type
'double'.
```

This does not work because **ii** has a value:

```
syms ii
ii=sym(1);
symsum(ii,ii,1,10)
??? Error using ==> mupadmex
Error in MuPAD command: summation variable must be an identifier or
indexed identifier [sum]

Error in ==> sym.symsum at 74
   r = mupadmex('symobj::map',f.s,'symobj::symsum',x.s,a.s,b.s);
```

This is correct:

```
clear ii
syms ii
symsum(ii,ii,1,10)
ans =
55
```

When **symsum(expr,ii,m,n)** is executed, the values m, m+1, ... ,n are substituted for **ii** in **expr**, and the results are summed. This substitution is automatic, and is part of the function of the **symsum** command. By contrast, consider the following loop:

```
clear
syms ii
expr=ii^2;
s=sym(0);
for ii=1:10
  s=s+expr;
end
s
s =
10*ii^2
```

Even though **ii** has a value when the line "**s=s+expr**" is executed, this value is not substituted into **expr** unless we specifically direct that it be. (The fact that the analogous substitution takes place during the execution of **symsum** is a special feature of **symsum**.) The **subs** command can be used to force this substitution:

```
clear
syms ii
expr=ii^2;
```

```
s=sym(0);
for ii=1:10
   s=s+subs(expr);
end
s
s =
385
```

This use of the **subs** command, **subs(expr)**, tells MATLAB that if a variable appears in **expr**, and that variable has a value in the MATLAB workspace, then the value should be substituted into **expr**.

Section 5.5: The Galerkin method

Since MATLAB can compute integrals that would be tedious to compute by hand, it is fairly easy to apply the Galerkin method with polynomial basis functions. (In the next section, I will discuss the finite element method, which uses *piecewise* polynomial basis functions.) Suppose we wish to approximate the solution to

$$-\frac{d}{dx}\left[(1+x)\frac{du}{dx}\right]=x^2,\ 0<x<1,$$
$$u(0)=0,\ u(1)=0,$$

using the subspace spanned by the following four polynomials:

```
clear
syms x
p1=x*(1-x)
p1 =
-x*(x - 1)
p2=x*(1/2-x)*(1-x)
p2 =
x*(x - 1)*(x - 1/2)
p3=x*(1/3-x)*(2/3-x)*(1-x)
p3 =
-x*(x - 1)*(x - 1/3)*(x - 2/3)
p4=x*(1/4-x)*(1/2-x)*(3/4-x)*(1-x)
p4 =
x*(x - 1)*(x - 1/2)*(x - 1/4)*(x - 3/4)
```

I have already defined the L^2 inner product in an M-file (**l2ip.m**). Here is an M-file function implementing the energy inner product:

```
type eip

function I=eip(f,g,k,a,b,x)

%I=eip(f,g,k,a,b,x)
%
%   This function computes the energy inner product of two functions
%   f(x) and g(x), that is, it computes the integral from a to b of
%   k(x)*f'(x)*g'(x).  The three functions must be defined by symbolic
%   expressions f, g, and k.
%
%   The variable of integration is assumed to be x.  A different
%   variable can passed in as the (optional) sixth input.
%
%   The inputs a and b, defining the interval [a,b] of integration,
%   are optional.  The default values are a=0 and b=1.

% Assign the default values to optional inputs, if necessary

if nargin<6
   syms x
end

if nargin<5
   b=1;
end

if nargin<4
   a=0;
end

% Compute the integral

I=int(k*diff(f,x)*diff(g,x),x,a,b);
```

Now the calculation is simple, although there is some repetitive typing required. (Below I will show how to eliminate most of the typing.) We just need to compute the stiffness matrix and the load vector, and solve the linear system. The stiffness matrix is

```
k=1+x;
K=[eip(p1,p1,k),eip(p1,p2,k),eip(p1,p3,k),eip(p1,p4,k)
eip(p2,p1,k),eip(p2,p2,k),eip(p2,p3,k),eip(p2,p4,k)
eip(p3,p1,k),eip(p3,p2,k),eip(p3,p3,k),eip(p3,p4,k)
eip(p4,p1,k),eip(p4,p2,k),eip(p4,p3,k),eip(p4,p4,k)]

K =
[     1/2,     -1/30,       1/90,     -1/672]
[   -1/30,      3/40,  -19/3780,      3/896]
[    1/90,  -19/3780,     5/567,  -41/60480]
[  -1/672,     3/896, -41/60480,   43/43008]
```

and the load vector is

```
f=x^2
f =
x^2
F=[l2ip(p1,f);l2ip(p2,f);l2ip(p3,f);l2ip(p4,f)]
F =
    1/20
  -1/120
   1/630
 -1/2688
```

I can now solve for the coefficients defining the (approximate) solution:

```
c=K\F
c =
   3325/34997
 -9507/139988
   1575/69994
```

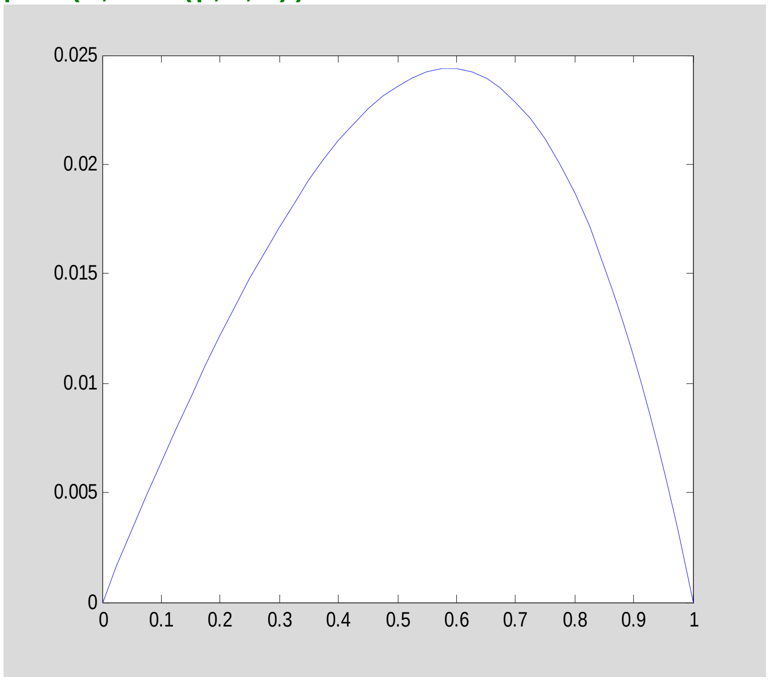
```
    420/34997
```

The approximate solution is

```
p=c(1)*p1+c(2)*p2+c(3)*p3+c(4)*p4
p =
(420*x*(x - 1)*(x - 1/2)*(x - 1/4)*(x - 3/4))/34997 - (9507*x*(x -
1)*(x - 1/2))/139988 - (1575*x*(x - 1)*(x - 1/3)*(x - 2/3))/69994 -
(3325*x*(x - 1))/34997
```

Here is a graph:

```
t=linspace(0,1,41);
plot(t,subs(p,x,t))
```



The exact solution can be found by integration:

```
syms c1 c2
int(-x^2,x)+c1
ans =
c1 - x^3/3
u=int(ans/k,x)+c2
u =
c2 - x/3 + log(x + 1)*(c1 + 1/3) + x^2/6 - x^3/9
```

Now I solve for the constants **c1** and **c2**:

```
c=solve(subs(u,x,sym(0)),subs(u,x,sym(1)),c1,c2)
c =
    c1: [1x1 sym]
    c2: [1x1 sym]

subs(u,c1,c.c1)
ans =
c2 - x/3 - log(x + 1)*((6*log(2) - 5)/(18*log(2)) - 1/3) + x^2/6 -
x^3/9
subs(ans,c2,c.c2)
ans =
x^2/6 - log(x + 1)*((6*log(2) - 5)/(18*log(2)) - 1/3) - x/3 - x^3/9
u=simplify(ans)
u =
(5*log(x + 1))/(18*log(2)) - x/3 + x^2/6 - x^3/9
```

Let us check the solution:

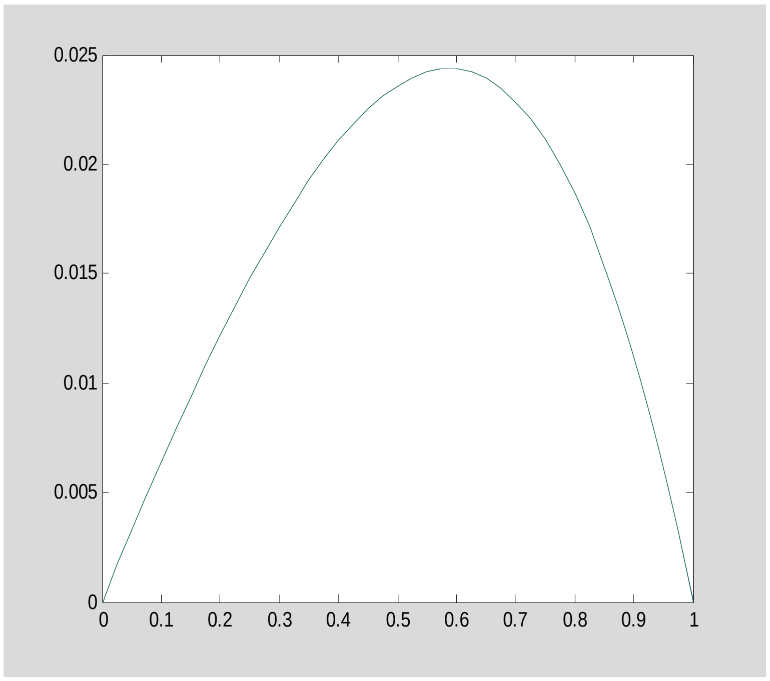
```
-diff(k*diff(u,x),x)
ans =
(x + 1)*((2*x)/3 + 5/(18*log(2)*(x + 1)^2) - 1/3) - 5/(18*log(2)*(x +
1)) - x/3 + x^2/3 + 1/3
simplify(ans)
ans =
x^2
subs(u,x,0)
ans =
     0
subs(u,x,1)
ans =
  2.7756e-017
```

This last result is a bit of a surprise. However, every quantity in MATLAB is floating point by default, so the "1" substituted into **u** is the floating point number 1, which causes the result to be computed in floating point. Here is what we really want:

```
subs(u,x,sym(1))
ans =
0
```

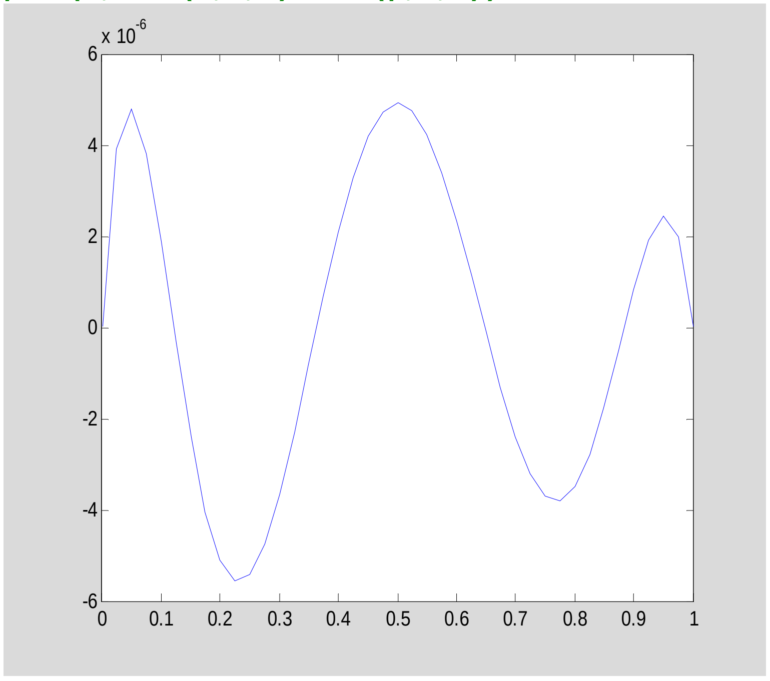
Since we have the exact solution, let us compare it with our approximate solution:

```
plot(t,subs(u,x,t),t,subs(p,x,t))
```



The computed solution is so close to the exact solution that the two graphs cannot be distinguished. Here is the error:

```
plot(t,subs(u,x,t)-subs(p,x,t))
```



Computing the stiffness matrix and load vector in loops

The above calculation is easy to perform, but it is rather annoying to have to type in the formulas for K and F in terms of all of the necessary inner products. It would be even worse were we to use an approximating subspace with a higher dimension. Fortunately, we can compute *K* and *F* in loops and greatly reduce the necessary typing. The key is to store the basis functions in a vector, so we can refer to them by index.

```
clear
syms x
p=[x*(1-x);x*(1/2-x)*(1-x);x*(1/3-x)*(2/3-x)*(1-x);x*(1/4-x)*(1/2-
x)*(3/4-x)*(1-x)];

pretty(p)

  +-                                           -+
  |                 -x (x - 1)                  |
  |                                             |
  |            x (x - 1) (x - 1/2)              |
  |                                             |
  |       -x (x - 1) (x - 1/3) (x - 2/3)        |
  |                                             |
  |  x (x - 1) (x - 1/2) (x - 1/4) (x - 3/4)    |
  +-                                           -+

k=1+x;
K=sym(zeros(4,4));
for ii=1:4
   for jj=1:4
      K(ii,jj)=eip(p(ii),p(jj),k);
   end
end
K
K =
[    1/2,    -1/30,       1/90,     -1/672]
[  -1/30,     3/40,  -19/3780,      3/896]
[   1/90, -19/3780,      5/567, -41/60480]
[ -1/672,    3/896, -41/60480,   43/43008]

f=x^2;
F=sym(zeros(4,1));
for ii=1:4
   F(ii)=l2ip(p(ii),f);
end
F
F =
    1/20
  -1/120
   1/630
 -1/2688
```

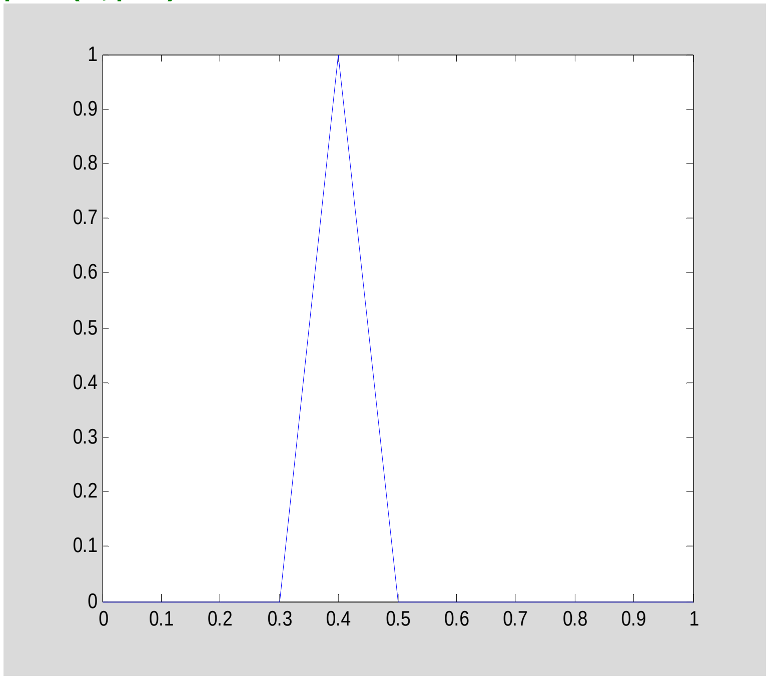
With this method, it is equally easy to compute **K** and **F** regardless of the number of basis functions. (Note that the above double loop that computes **K** is not as efficient as it could be, since **K** is known to be symmetric.)

Section 5.6: Piecewise polynomials and the finite element method

The finite element method is the Galerkin method with a piecewise linear basis. The computations are a bit more complicated than in the previous section, since a basis function is not defined by a single formula, as in the previous section.

Here is a typical basis function φ$_i$ on a mesh with *n* elements:

```
clear
n=10;
x=linspace(0,1,n+1)';
phi=zeros(n+1,1);
ii=5;
phi(ii)=1;
plot(x,phi)
```



The basis function φ$_i$ is defined by

$$\varphi_i(x)=\begin{cases}\dfrac{x-x_{i-1}}{h},\ x_{i-1}<x<x_i,\\ -\dfrac{x-x_i}{h},\ x_i<x<x_{i+1},\\ 0,\quad \text{otherwise.}\end{cases}$$

In these formulas, *h*=(*b*-*a*)/*n*, where the interval of interest is [*a*,*b*] and the number of elements is *n*, and

$$x_i=a+ih,\ i=0,1,2,\ldots,n.$$

The easiest way to represent φ$_i$ is to define two expressions representing the two nonzero pieces. I will assume that *a* (the left endpoint of the interval) is zero:

```
clear
syms ii h x
phi1=(x-(ii-1)*h)/h
phi1 =
(x - h*(ii - 1))/h
```

```
phi2=-(x-(ii+1)*h)/h
phi2 =
-(x - h*(ii + 1))/h
```

Computing the load vector

Here is how I would use the above expressions to compute a load vector. Consider the following BVP:

$$-\frac{d^2u}{dx^2} = x^2,\, 0 < x < 1,$$
$$u(0) = 0,\, u(1) = 0.$$

I define the right-hand side:

```
f=x^2;
```

Now I choose *n* and define *h*:

```
n=10;
h=1/n;
```

Here is the computation:

```
F=zeros(n-1,1);
for  ii=1:n-1
  F(ii)=int(subs(phi1)*f,x,ii*h-
h,ii*h)+int(subs(phi2)*f,x,ii*h,ii*h+h);
end


F
F =
    0.0012
    0.0042
    0.0092
    0.0162
    0.0252
    0.0362
    0.0492
    0.0642
    0.0812
```

Notice the use of **subs(phi1)** and **subs(phi2)** in the above loop. When the **subs** command is called with a single expression, it tells MATLAB to substitute, for variables in the expression, any value that exist in the workspace. In the above example, those variables were **h** and **ii**:

```
phi1
phi1 =
(x - h*(ii - 1))/h
ii=10;
subs(phi1)
ans =
10*x - 9
```

Computing the stiffness matrix

Computing the stiffness matrix is simple when the coefficient in the differential equation is a constant k, because then we already know the entries in the stiffness matrix (they were derived in the text).

The diagonal entries in the stiffness matrix are all $2k/h$, and the entries on the subdiagonal and superdiagonal are all $-k/h$. All other entries are zero.

As I explained in the text, one of the main advantages of the finite element method is that the stiffness matrix is sparse. One of the main advantages of MATLAB is that it is almost as easy to create and use sparse matrices as it is to work with ordinary (dense) matrices! (There are some exceptions to this statement. Some matrix functions cannot operate on sparse matrices.) Here is one way to create the stiffness matrix. Notice that I first allocate an n-1 by n-1 sparse matrix and then write two loops, one to fill the main diagonal, and the second to fill the subdiagonal and superdiagonal. (I take k=1 in this example.)

```
k=1;
K=sparse(n-1,n-1)
K =
   All zero sparse: 9-by-9
for ii=1:n-1
    K(ii,ii)=2*k/h;
end
for ii=1:n-2
   K(ii,ii+1)=-k/h;
   K(ii+1,ii)=-k/h;
end
```

Notice that I wrote two loops because the subdiagonal and superdiagonal have only n-2 entries, while the main diagonal has n-1 entries. (Note: It is more efficient to create the sparse matrix **K** use the spdiags command, which allows us to define a sparse matrix by specifying its diagonals. The spdiags command is explained below.)

Here is the matrix **K**:

```
K
K =
   (1,1)       20
   (2,1)      -10
   (1,2)      -10
   (2,2)       20
   (3,2)      -10
   (2,3)      -10
   (3,3)       20
   (4,3)      -10
   (3,4)      -10
   (4,4)       20
   (5,4)      -10
   (4,5)      -10
   (5,5)       20
   (6,5)      -10
   (5,6)      -10
   (6,6)       20
   (7,6)      -10
   (6,7)      -10
   (7,7)       20
   (8,7)      -10
   (7,8)      -10
   (8,8)       20
   (9,8)      -10
   (8,9)      -10
   (9,9)       20
```

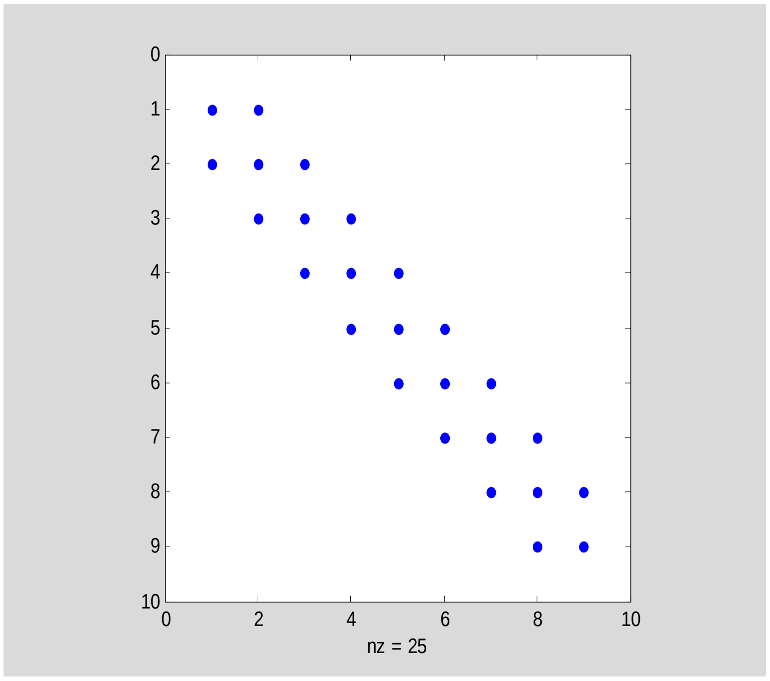
Notice that MATLAB only stores the nonzeros, so the above output is rather difficult to read. I can, if I wish, convert **K** to a full matrix to view it in the usual format:

```
full(K)
ans =
    20   -10     0     0     0     0     0     0     0
   -10    20   -10     0     0     0     0     0     0
     0   -10    20   -10     0     0     0     0     0
     0     0   -10    20   -10     0     0     0     0
     0     0     0   -10    20   -10     0     0     0
     0     0     0     0   -10    20   -10     0     0
     0     0     0     0     0   -10    20   -10     0
     0     0     0     0     0     0   -10    20   -10
     0     0     0     0     0     0     0   -10    20
```

However, this is usually not a good idea, since we tend to use sparse matrices when the size of the problem is large, in which case converting the sparse matrix to a dense matrix partially defeats the purpose.

The **spy** command is sometimes useful. It plots a schematic view of a matrix, showing where the nonzero entries are:

```
spy(K)
```



I can now solve **Ku=F** to get the nodal values (I computed **F** earlier):

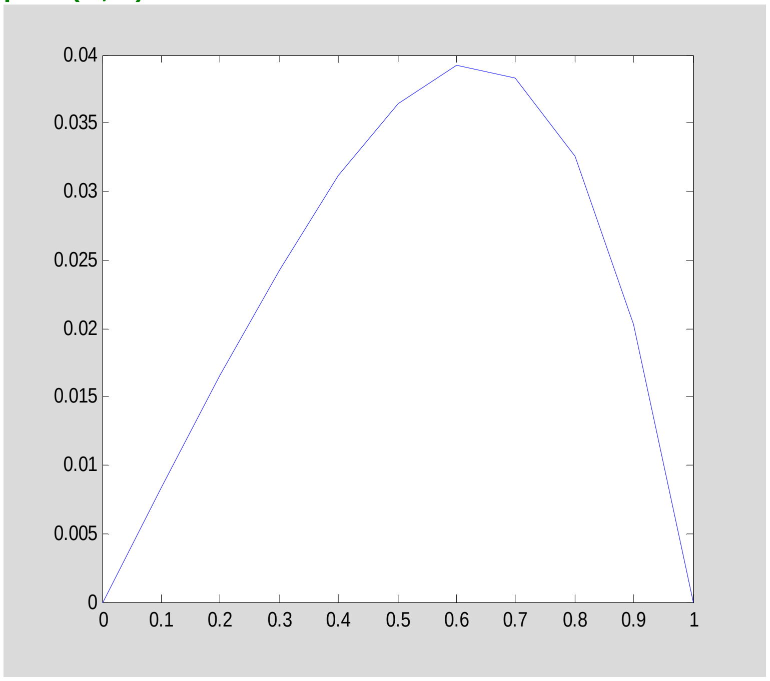
```
u=K\F
u =
    0.0083
    0.0165
    0.0243
    0.0312
    0.0365
    0.0392
    0.0383
    0.0325
    0.0203
```

For graphical purposes, I often explicitly put in the nodal values (zero) at the endpoints:

```
u=[0;u;0]
u =
         0
    0.0083
    0.0165
    0.0243
    0.0312
    0.0365
    0.0392
    0.0383
    0.0325
    0.0203
         0
```

If I now create the grid, I can easily plot the computed solution:

```
t=linspace(0,1,n+1)';
plot(t,u)
```



As I mentioned earlier, MATLAB's **plot** command simply graphs the given data points and connects them with straight line segments---that is, it automatically graphs the continuous piecewise linear function defined by the data!

The nonconstant coefficient case

Now consider the BVP

$$-\frac{d}{dx}\left[(1+x)\frac{du}{dx}\right]=x^2,\ 0<x<1,$$
$$u(0)=0,\ u(1)=0,$$

Now when I fill the stiffness matrix, I must actually compute the integrals, since their values are not known *a priori*. I can, however, simplify matters because I know the derivatives of **phi1** (1/h) and **phi2** (-1/h). Here is the computation of of the main diagonal of **K**:

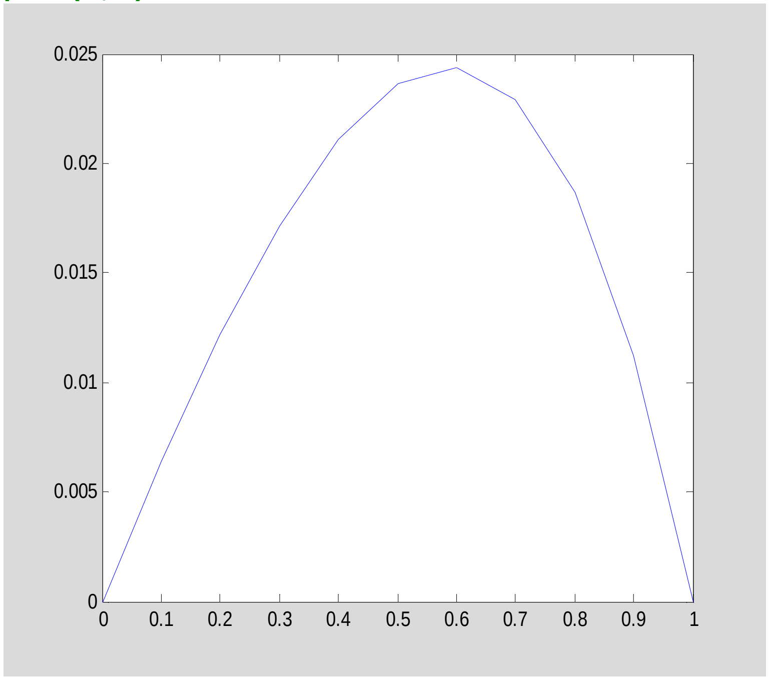
```
k=x+1;
K=sparse(n-1,n-1);
for ii=1:n-1
   K(ii,ii)=int(k/h^2,x,ii*h-h,ii*h)+int(k/h^2,x,ii*h,ii*h+h);
end
```

Here is the loop that computes the subdiagonal and super diagonal. Recall that **K** is symmetric, which means that I do not need to compute the subdiagonal once I have the superdiagonal:

```
for ii=1:n-2
   K(ii,ii+1)=double(int(-k/h^2,x,ii*h,ii*h+h));
   K(ii+1,ii)=K(ii,ii+1);
end
```

Now I can compute the nodal values and plot the result, as before:

```
u=K\F;
u=[0;u;0];
plot(t,u)
```



Creating a piecewise linear function from the nodal values

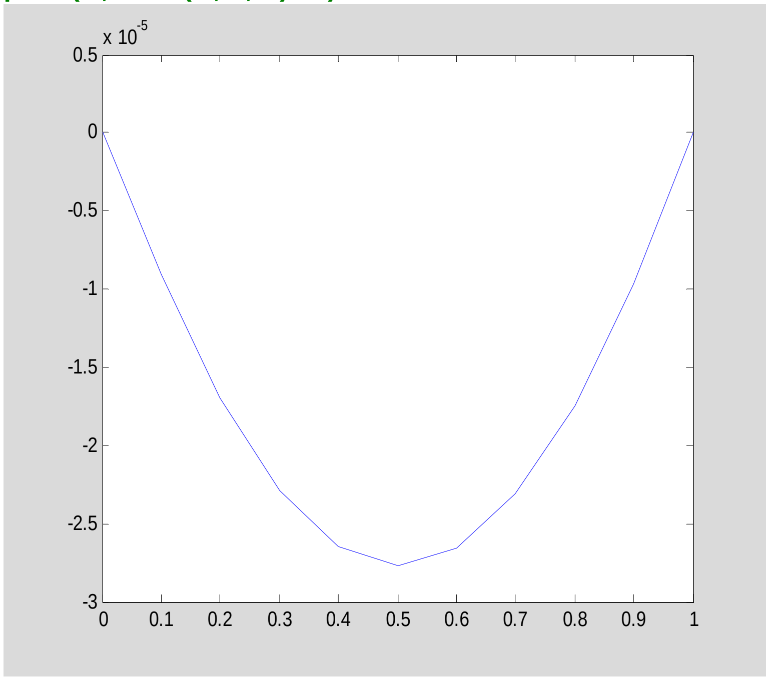
Here is the exact solution to the previous BVP:

```
syms c1 c2
int(-x^2,x)+c1;
U=int(ans/k,x)+c2;
c=solve(subs(U,x,sym(0)),subs(U,x,sym(1)),c1,c2);
subs(U,c1,c.c1);
subs(ans,c2,c.c2);
U=simplify(ans);
pretty(U)
```

```
                            2    3
  5 log(x + 1)    x    x     x
  ------------ - - + -- - --
   18 log(2)      3    6     9
```

Now I would like to compare this exact solution with the solution I computed above using the finite element method. I can plot the error as follows:

```
plot(t,subs(U,x,t)-u)
```



However, this is really a misleading graph, since I computed the errors at the nodes and (implicitly, by using the MATLAB **plot** command) that the error is linear in between the nodes. This is not true.

To see the true error, I must create a piecewise linear function from the grid and the nodal values, and then compare it to the true solution. MATLAB has some functions for working with piecewise polynomials, including **mkpp**, which creates an array describing a piecewise polynomial according to MATLAB's own data structure, and ppval, which evaluates a piecewise polynomial. Because **mkpp** is a little complicated, I wrote a function, **mkpl.m**, specifically for creating a piecewise linear function. It takes as inputs the grid and the nodal values:

```
help mkpl
```

```
  pl = mkpl(x,v)

    This function creates a continuous, piecewise linear function
    interpolating the data (x(i),v(i)), i=1,...,length(x).  The
    vectors x and v must be of the same length. Also, it is
    assumed that the components of x are increasing.
```

Here is the piecewise linear function we computed using the finite element method:

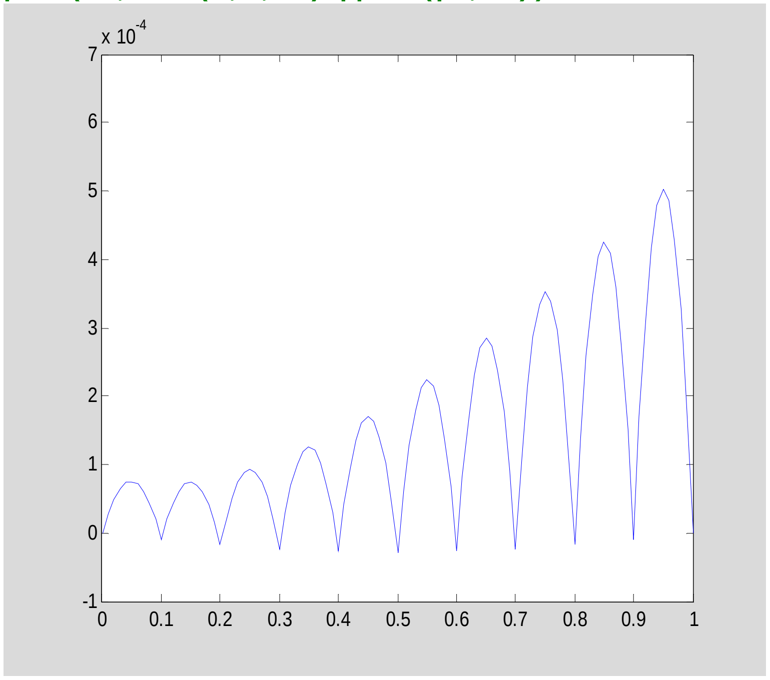
```
pu=mkpl(t,u);
```

I now create a fine grid:

```
tt=linspace(0,1,10*n+1)';
```

I can now use **ppval** to evaluate **pu** on the grid (and **subs** to evaluate **U**, as usual):

```
plot(tt,subs(U,x,tt)-ppval(pu,tt))
```



You should notice that the error is very small at the nodes---graphing only the nodal error gives a misleading impression of the size of the error.

More about sparse matrices

If a matrix **B** is stored in ordinary (dense) format, then the command **S =sparse(A)** creates a copy of the matrix stored in sparse format. For example:

```
clear
A = [0 0 1;1 0 2;0 -3 0]
A =
     0     0     1
     1     0     2
     0    -3     0
```

```
 S = sparse(A)
S =
   (2,1)        1
   (3,2)       -3
   (1,3)        1
   (2,3)        2

whos
  Name       Size              Bytes  Class     Attributes

  A          3x3                  72  double
  S          3x3                  64  double    sparse
```

Unfortunately, this form of the sparse command is not particularly useful, since if **A** is large, it can be very time-consuming to first create it in dense format. The command **S = sparse(m,n)** (as we have already seen) creates an *m* by *n* zero matrix in sparse format. Entries can then be added one-by-one:

```
A = sparse(3,2)
A =
   All zero sparse: 3-by-2

A(1,2)=1;
A(3,1)=4;
A(3,2)=-1;
A
A =
   (3,1)        4
   (1,2)        1
   (3,2)       -1
```

(Of course, for this to be truly useful, the nonzeros would be added in a loop or by some other method.)

Another version of the sparse command is **S = sparse(I,J,S,m,n,maxnz)**. This creates an *m* by *n* sparse matrix with entry **(I(k),J(k))** equal to **S(k)**. The optional argument **maxnz** causes MATLAB to pre-allocate storage for **maxnz** nonzero entries, which can increase efficiency in the case when more nonzeros will be later added to **S**.

There are still more versions of the **sparse** command. See "help sparse" for details.

The most common type of sparse matrix is a banded matrix, that is, a matrix with a few nonzero diagonals. Such a matrix can be created with the **spdiags** command. Consider the following matrix:

```
A =[
    64   -16     0   -16     0     0     0     0     0
   -16    64   -16     0   -16     0     0     0     0
     0   -16    64     0     0   -16     0     0     0
   -16     0     0    64   -16     0   -16     0     0
     0   -16     0   -16    64   -16     0   -16     0
     0     0   -16     0   -16    64     0     0   -16
     0     0     0   -16     0     0    64   -16     0
     0     0     0     0   -16     0   -16    64   -16
     0     0     0     0     0   -16     0   -16    64];
```

(notice the technique for entering the rows of a large matrix on several lines). This is a 9 by 9 matrix with 5 nonzero diagonals. In MATLAB's indexing scheme, the nonzero diagonals of **A** are numbers -3, -1, 0, 1, and 3 (the main diagonal is number 0, the first subdiagonal is number -1, the first superdiagonal is number 1, and so forth). To create the same matrix in sparse format, it is first necessary to create a 9 by 5 matrix containing the nonzero diagonals of A. Of course, the diagonals, regarded as column vectors, have different lengths; only the main diagonal has length 9. In order to gather the various diagonals in a single matrix, the shorter diagonals must be padded with zeros (or any other number---these extra entries are ignored). The rule is that the extra zeros go at the bottom for subdiagonals and at the top for superdiagonals. Thus we create the following matrix:

```
 B = [
   -16   -16    64     0     0
   -16   -16    64   -16     0
   -16     0    64   -16     0
   -16   -16    64     0   -16
   -16   -16    64   -16   -16
   -16     0    64   -16   -16
     0   -16    64     0   -16
     0   -16    64   -16   -16
     0     0    64   -16   -16];
```

The spdiags command also needs the indices of the diagonals:

```
d = [-3,-1,0,1,3];
```

The matrix is then created as follows:

```
S = spdiags(B,d,9,9);
```

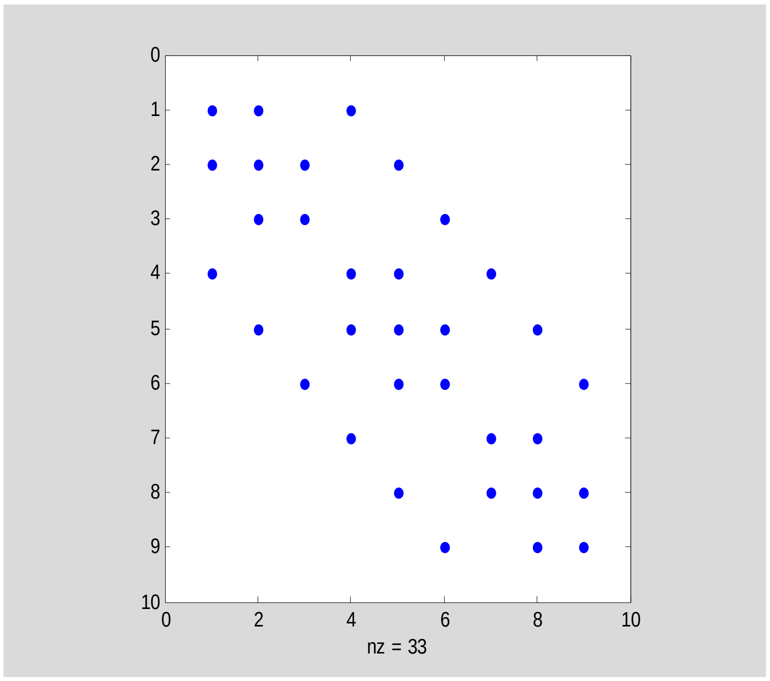
The last two arguments give the size of **S**.

Perhaps the most common sparse matrix is the identity. Recall that an identity matrix can be created, in dense format, using the command **eye**. To create the *n* by *n* identity matrix in sparse format, use **I** = **speye(n)**. For example:

```
I=speye(3)
I =
   (1,1)        1
   (2,2)        1
   (3,3)        1
```

Recall that the **spy** command is very useful for visualizing a sparse matrix:

```
spy(A)
```



Chapter 6: Heat flow and diffusion

Section 6.1: Fourier series methods for the heat equation

Example 6.2: An inhomogeneous example

Consider the IBVP

$$\begin{aligned} &\frac{\partial u}{\partial t} - A\frac{\partial^2 u}{\partial x^2} = 10^{-7},\ 0 < x < 100,\ t > 0,\\ &u(x,0) = 0,\ 0 < x < 100,\\ &u(0,t) = 0,\ t > 0,\\ &u(100,t) = 0,\ t > 0. \end{aligned}$$

The constant A has value 0.208 cm^2/s.

```
clear
A=0.208
A =
    0.2080
```

The solution can be written as

$$u(x,t) = \sum_{n=1}^{\infty} a_n(t)\sin\left(\frac{n\pi x}{100}\right)$$

where the coefficient $a_n(t)$ satisfies the IVP

$$\frac{da_n}{dt}+\frac{An^2\pi^2}{100^2}a_n=c_n,\ t>0,$$
$$a_n(0)=0.$$

The values c_1, c_2, c_3, ... are the Fourier sine coefficients of the constant function 10^{-7}:

```
syms n x pi
2/100*int(10^(-7)*sin(n*pi*x/100),x,0,100)
ans =
-(cos(pi*n) - 1)/(5000000*pi*n)
mysubs(ans,n)
ans =
-((-1)^n - 1)/(5000000*pi*n)
c=ans
c =
-((-1)^n - 1)/(5000000*pi*n)
```

The coefficient $a_n(t)$ is given by the following formula:

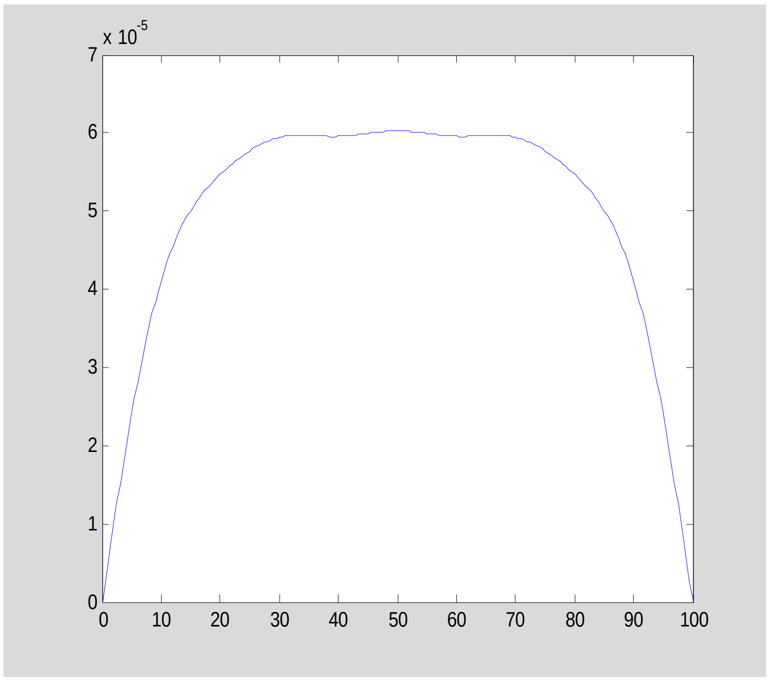
```
syms t s
int(exp(-A*n^2*pi^2*(t-s)/100^2)*c,s,0,t)
ans =
((-1)^n - 1)/(104*pi^3*n^3*exp((13*pi^2*n^2*t)/625000)) - ((-1)^n -
1)/(104*pi^3*n^3)
a=simplify(ans)
a =
((1/exp((13*pi^2*n^2*t)/625000) - 1)*((-1)^n - 1))/(104*pi^3*n^3)
```

Next I define the (partial) Fourier series of the solution:

```
S=symsum(a*sin(n*pi*x/100),n,1,10);
```

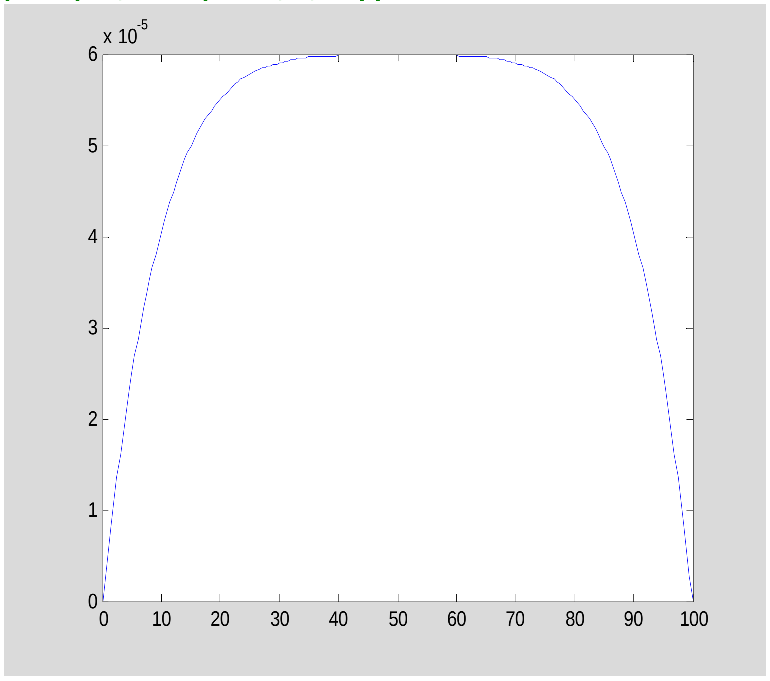
I can now look at some "snapshots" of the solution. For example, I will show the concentration distribution after 10 minutes (600 seconds). Some trial and error may be necessary to determine how many terms in the Fourier series are required for a qualitatively correct solution. (As discussed in the text, this number decreases as t increases.)

```
S600=subs(S,t,600);
xx=linspace(0,100,201);
plot(xx,subs(S600,x,xx))
```



The wiggles in the graph suggest that 10 terms is not enough for a correct graph (after 10 minutes, the concentration distribution ought to be quite smooth). Therefore, I will try again with 20 terms:

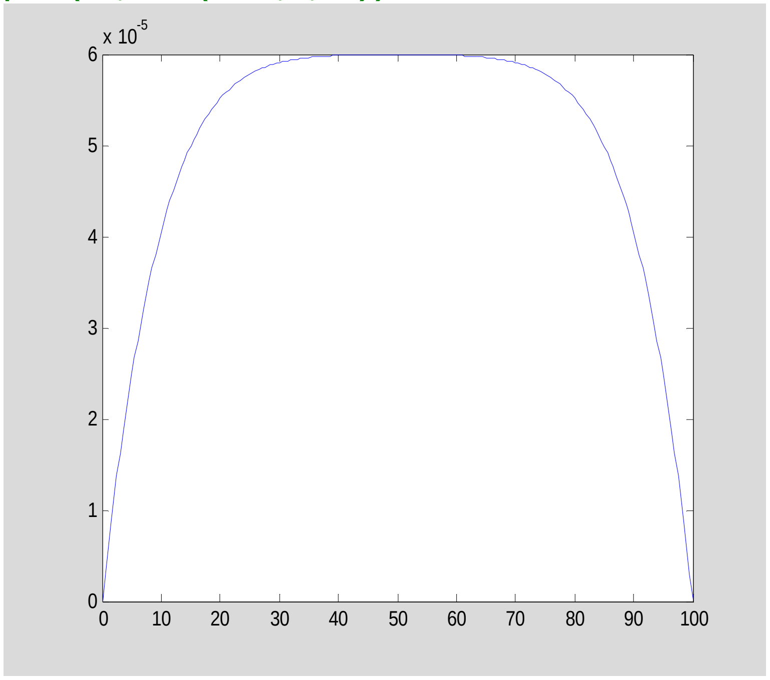
```
S=symsum(a*sin(n*pi*x/100),n,1,20);
S600=subs(S,t,600);
plot(xx,subs(S600,x,xx))
```



Now with 40 terms:

```
S=symsum(a*sin(n*pi*x/100),n,1,40);
S600=subs(S,t,600);
```

```
plot(xx,subs(S600,x,xx))
```



It appears that 20 terms is enough for a qualitatively correct graph at *t*=600.

Section 6.4: Finite element methods for the heat equation

Now I will show how to use the backward Euler method with finite element discretization to (approximately) solve the heat equation. Since the backward Euler method is implicit, it is necessary to solve an equation at each step. This makes it difficult to write a general-purpose program implementing backward Euler, and I will not attempt to do so. Instead, the function **beuler.m** applies the backward Euler method to the system of ODEs

$$M\frac{da}{dt}+Ka=f(t),\, t>0,$$
$$a(0)=a_0,$$

which is the result of applying the finite element to the heat equation.

```
type beuler

function [a,t]=beuler(M,K,f,a0,N,dt)

%[a,t]=beuler(M,K,f,a0,N,dt)
%
%   This function applies the backward Euler method to solve the
%   IVP
%
%       Ma'+Ka=f(t),
%       a(0)=a0,
%
%   where M and K are mxm matrices (with M invertible) and f is
%   a vector-valued function taking two arguments, t and m.
```

```
%
%   N steps are taken, each of length dt.

m=length(a0);
a=zeros(m,N+1);
a(:,1)=a0;
t=linspace(0,N*dt,N+1)';

L=M+dt*K;
for ii=1:N
   a(:,ii+1)=L\(M*a(:,ii)+dt*f(t(ii+1),m));
end
```

To solve a specific problem, I must compute the mass matrix M, the stiffness matrix K, the load vector $f(t)$, and the initial data a_0. The techniques should by now be familiar.

Example 6.9

A 100 cm iron bar, with ρ=7.88, c=0.437, and κ=0.836, is chilled to an initial temperature of 0 degrees, and then heated internally with both ends maintained at 0 degrees. The heat source is described by the function

$$F(x,t)=10^{-8}tx(100-x)^2.$$

The temperature distribution is the solution of

$$\begin{aligned}&\rho c\frac{\partial u}{\partial t}-\kappa\frac{\partial^2 u}{\partial x^2}=F(x,t),\ 0<x<100,\ t>0,\\&u(x,0)=0,\ 0<x<100,\\&u(0,t)=0,\ t>0,\\&u(100,t)=0,\ t>0.\end{aligned}$$

I will use standard piecewise linear basis functions and the techniques introduced in Section 5.6 of this tutorial to compute the mass and stiffness matrices:

```
clear
n=100
n =
   100
h=100/n
h =
     1
k=0.836
k =
    0.8360
p=7.88
p =
    7.8800
c=0.437
c =
    0.4370

K=spdiags([-k/h*ones(n-1,1),2*k/h*ones(n-1,1),-k/h*ones(n-1,1)],[-
1,0,1],n-1,n-1);
```

```
M=spdiags([p*c*h/6*ones(n-1,1),2*p*c*h/3*ones(n-1,1),p*c*h/6*ones(n-
1,1)],[-1,0,1],n-1,n-1);
```

(Note that, for this constant coefficient problem, we do not need to perform any integrations, as we already know the entries in the mass and stiffness matrices.)

Now I compute the load vector. Here is the typical entry:

```
clear ii
syms x t ii
phi1=(x-(ii-1)*h)/h
phi1 =
x - ii + 1
phi2=-(x-(ii+1)*h)/h
phi2 =
ii - x + 1
F=10^(-8)*t*x*(100-x)^2
F =
(t*x*(x - 100)^2)/100000000
int(F*subs(phi1),x,ii*h-h,ii*h)+int(F*subs(phi2),x,ii*h,ii*h+h)
ans =
(t*(30*ii^3 - 5970*ii^2 + 296015*ii + 99003))/6000000000 + (t*(30*ii^3
- 6030*ii^2 + 304015*ii - 101003))/6000000000
simplify(ans)
ans =
(t*(6*ii^3 - 1200*ii^2 + 60003*ii - 200))/600000000
```

Now I need to turn this formula into a vector-valued function that I can pass to **beuler**. I write an M-file function **f6.m**:

```
type f6

function y=f(t,n)

ii=(1:n)';
y=(t*(6*ii.^3 - 1200*ii.^2 + 60003*ii - 200))/600000000;
%y=-1/3000000*t+1/100000000*t*ii.^3-1/500000*t*ii.^2+...
%   20001/200000000*t*ii;
```

(Note the clever MATLAB programming in **f6**: I made **ii** a vector with components equal to 1,2 ,...,*n*-1. Then I can compute the entire vector in one command rather than filling it one component at a time in a loop.)

Next I create the initial vector **a0**. Since the initial value in the IBVP is zero, **a0** is the zero vector:

```
a0=zeros(n-1,1);
```

Now I choose the time step and invoke **beuler**:

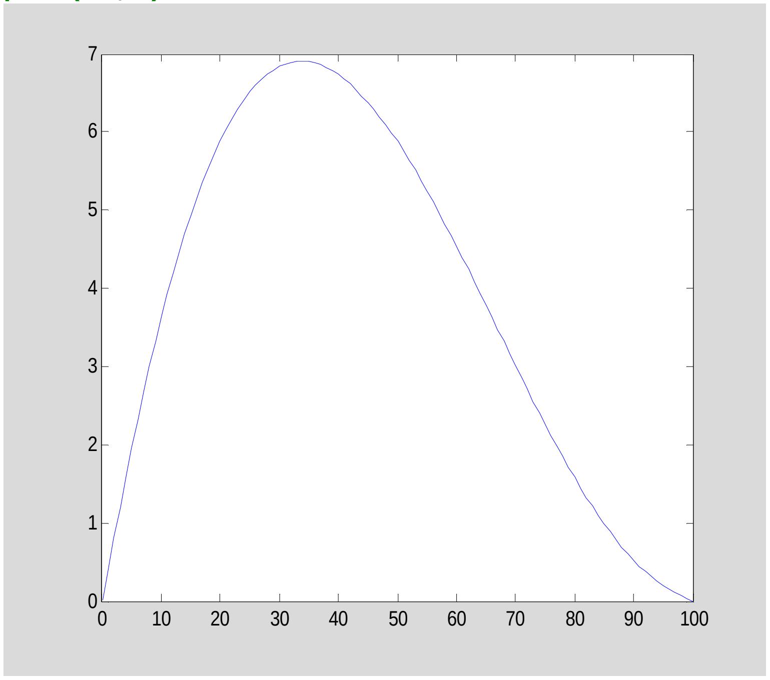
```
dt=2;
N=180/dt;
f=@f6;
[a,t]=beuler(M,K,f,a0,N,dt);
```

The last column of a gives the temperature distribution at time *t*=180 (seconds). I will put in the zeros at the beginning and end that represent the Dirichlet conditions:

```
T=[0;a(:,N+1);0];
xx=linspace(0,100,n+1)';
```

Here is a plot of the temperature after 3 minutes:

```
plot(xx,T)
```



Chapter 8: First-Order PDEs and the Method of Characteristics

Section 8.1: The simplest PDE and the method of characteristics

When solving PDEs in two variables, it is sometimes desirable to graph the solution as a function of two variables (that is, as a surface), rather than plotting snapshots of the solution. This is particularly appropriate when neither variable is time.

Two-dimensional graphics in MATLAB

Recall that to plot a function of one variable, we create a grid using the **linspace** command and then evaluate the desired function on the grid. We can then call the **plot** command. For a function of two variables, the procedure is similar. However, we need to create a grid on a rectangle rather than on an interval, which is a bit more complicated.

The **meshgrid** command takes two one-dimensional grids, on the intervals $a<x<b$ and $c<y<d$, and creates the necessary grid on the rectangle $\{(x,y) : a<x<b \text{ and } c<y<d\}$. This grid is represented as two matrices **X** and **Y**; the points in the grid are then (X_{ij},Y_{ij}). Evaluating $f(x,y)$ on the grid means producing a matrix **Z** such that

$$Z_{ij} = f(X_{ij}, Y_{ij}) \text{ for all } i, j.$$

This last step is easy, since MATLAB supports vectorized operations.

Here is how I created Figure 8.2 (page 314):

First, I create the two one-dimensional grids:

```
x=linspace(-5,5,41)';
y=linspace(0,10,41)';
```

Next, I invoke **meshgrid** to create the two-dimensional grid:

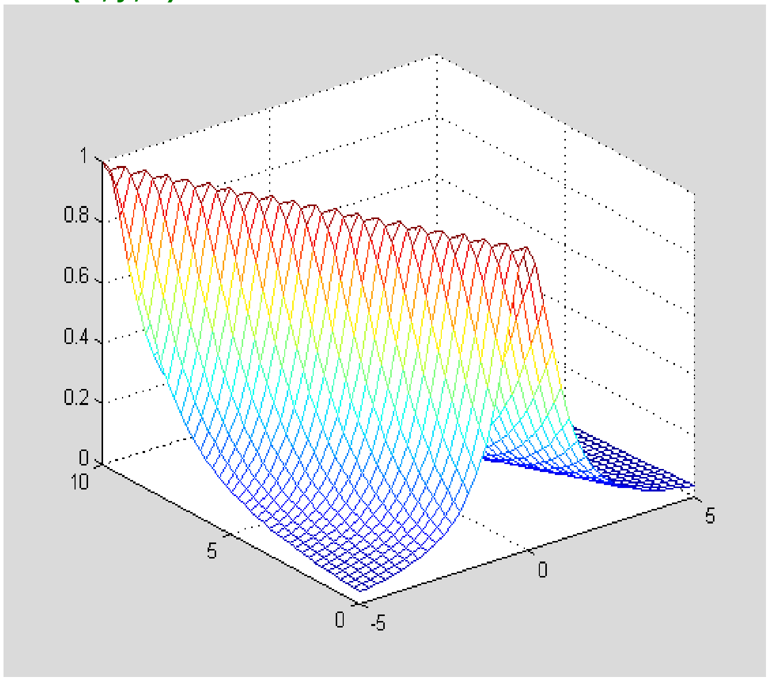
```
[X,Y]=meshgrid(x,y);
```

Finally, I compute the function u on the grid:

```
z=1./(1+(X+0.5*Y).^2);
```

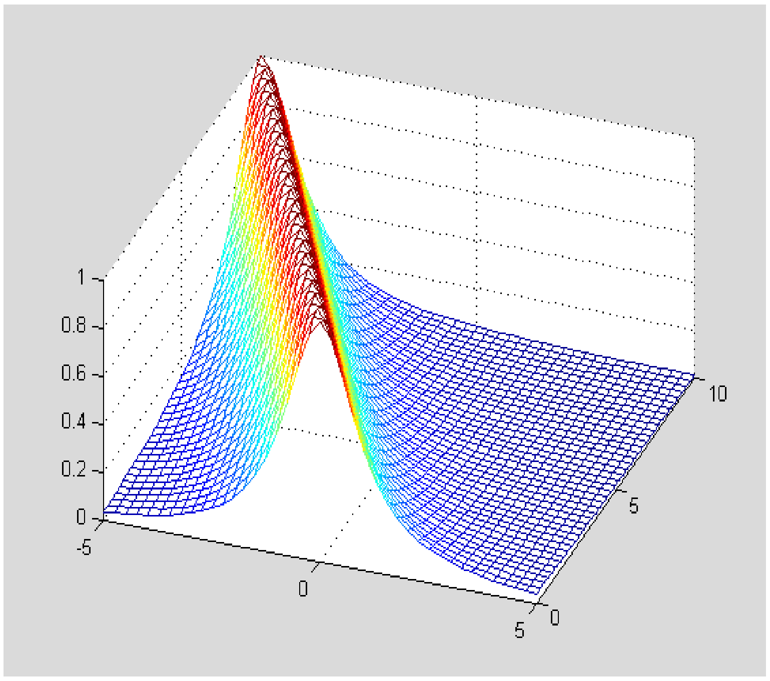
The command for plotting the surface *z*=*f*(*x*,*y*) is called **mesh**:

```
mesh(x,y,z)
```



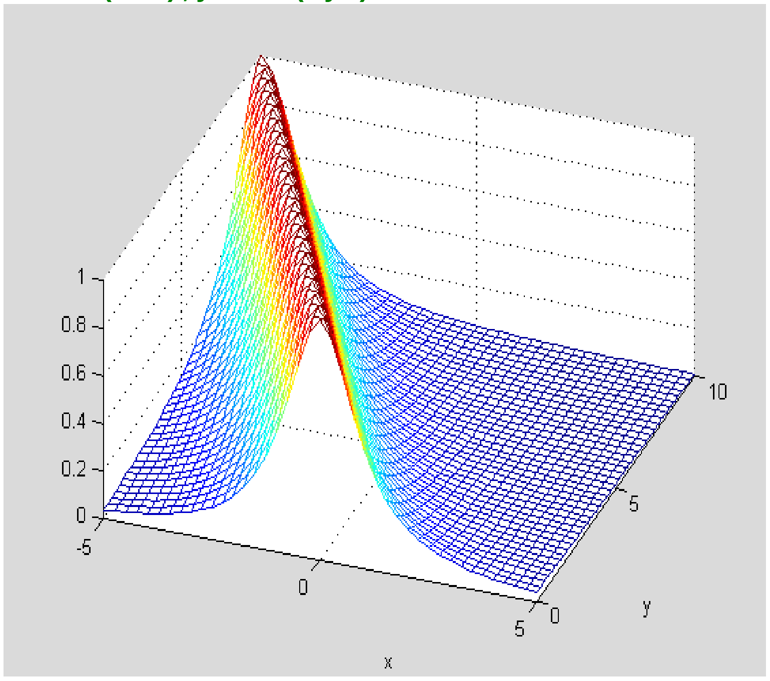
This surface looks better when viewed from a different angle (see "**help view**" for details):

```
view(20,45)
```



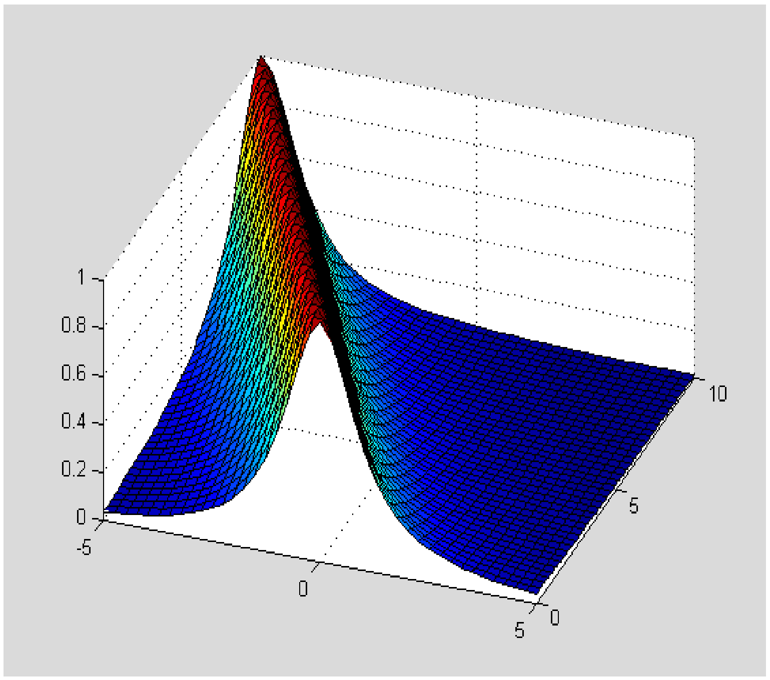
I can add the axis labels as follows:

```
xlabel('x');ylabel('y')
```



An alternative to **mesh** is **surf**:

```
surf(x,y,z);view(20,45)
```



Section 8.2: First-order quasi-linear PDEs

The purpose of the method of characteristics is to reduce a PDE to a family of ODEs. MATLAB has a function called **dsolve** that will solve many ordinary differential equations symbolically. I will illustrate the use of dsolve in the context of Example 8.6 (pages 323-324 in the text):

Example 8.6

The method of characteristics reduces the given PDE to the IVP

$$\frac{dv}{dt} = v^2, v(0) = \frac{1}{1+s^2}.$$

dsolve will solve this problem as follows:

```
dsolve('Dv=v^2','v(0)=1/(1+s^2)')
ans =
1/(s^2 - t + 1)
```

There are several points to notice about the use of **dsolve**:

The differential equation is specified as a string, in (single) quotation marks, and it is followed by the initial condion(s) (if any), also in single quotes. Notice that, unlike with the **solve** command, there is no option to give the equation explicitly using symbolic variables, rather than as a string. **dsolve** determines the name of the unknown function from the equation, and assumes that the independent variable is t.

Any derivatives are specified using "D" as an operator. If a higher-order derivative appears, it is specified using "D2", "D3", etc. For example:

```
dsolve('D2x=-x','x(0)=0','Dx(0)=1')
ans =
sin(t)
```

If no initial condition is given, dsolve returns the general solution of the ODE (when possible):

```
dsolve('Dv=v^2')
ans =
            0
 -1/(C13 + t)
```

Notice that, in this example, two solutions are returned. This is because the formula $v(t) = {}^{-1}/_{(C+t)}$ does not include the special solution $v(t) = 0$. This situation is always possible when the differential equation is nonlinear. Also notice that MATLAB chose a name for the constant in the general solution. If it solves another problem, it will choose another name for the constant:

```
dsolve('Dv=v')
ans =
C16*exp(t)
```

If desired, you can give dsolve a different name for the independent variable; this has to be the final input:

```
dsolve('Dv=v^2','x')
ans =
            0
 -1/(C19 + x)
```

dsolve will solve a system of ODEs (when possible); the equations in the system are given as multiple inputs to **dsolve**. This is illustrated in the context of Example 8.7 (pages 325-326 in the text):

Example 8.7

The method of characteristics, applied to the given PDE, yields the following system of ODEs:

dx/dt=v, x(0)=s,
dy/dt=y, y(0)=1,
dv/dt=x, v(0)=2s.

We solve this as follows:

```
sol=dsolve('Dx=v','Dy=y','Dv=x','x(0)=s','y(0)=1','v(0)=2*s')
sol =
    y: [1x1 sym]
    x: [1x1 sym]
    v: [1x1 sym]
sol.x
ans =
(3*s*exp(t))/2 - s/(2*exp(t))
sol.y  ans =
exp(t)
sol.v
ans =
s/(2*exp(t)) + (3*s*exp(t))/2
```

Chapter 11: Problems in multiple spatial dimensions

Section 11.2: Fourier series on a rectangular domain

Fourier series calculations on a rectangular domain proceed in almost the same fashion as in one-dimensional problems. The key difference is that we must compute double integrals and double sums in place of single integrals and single sums. Fortunately, this is not difficult, since a double integral over a rectangle is just an iterated integral.

As an example, I will compute the Fourier double sine series of the following function *f* on the unit square:

```
clear
syms x y
f=x*y*(1-x)*(1-y)
f =
x*y*(x - 1)*(y - 1)
```

The Fourier series has the form

$$\sum_{m=1}^{\infty}\sum_{n=1}^{\infty} a_{mn}\sin(m\pi x)\sin(n\pi y),$$

where a_{mn} is computed as follows:

```
syms m n pi
a=4*int(int(f*sin(m*pi*x)*sin(n*pi*y),y,0,1),x,0,1)
a =
(4*(16*cos((pi*m)/2)^2*cos((pi*n)/2)^2 +
8*pi*n*sin((pi*n)/2)*cos((pi*m)/2)^2*cos((pi*n)/2) - 16*cos((pi*m)/2)^2
+ 8*pi*m*sin((pi*m)/2)*cos((pi*m)/2)*cos((pi*n)/2)^2 +
4*pi^2*m*n*sin((pi*m)/2)*sin((pi*n)/2)*cos((pi*m)/2)*cos((pi*n)/2) -
8*pi*m*sin((pi*m)/2)*cos((pi*m)/2) - 16*cos((pi*n)/2)^2 -
8*pi*n*sin((pi*n)/2)*cos((pi*n)/2) + 16))/(pi^6*m^3*n^3)
```

Here is the partial Fourier series with a total of 100 terms:

```
S=symsum(symsum(a*sin(m*pi*x)*sin(n*pi*y),n,1,10),m,1,10);
```

To graph the (partial) Fourier series, I begin by creating, the two one-dimensional grids:

```
xx=linspace(0,1,21)';
yy=linspace(0,1,21)';
```

Next, I invoke **meshgrid** to create the two-dimensional grid:

```
[X,Y]=meshgrid(xx,yy);
```

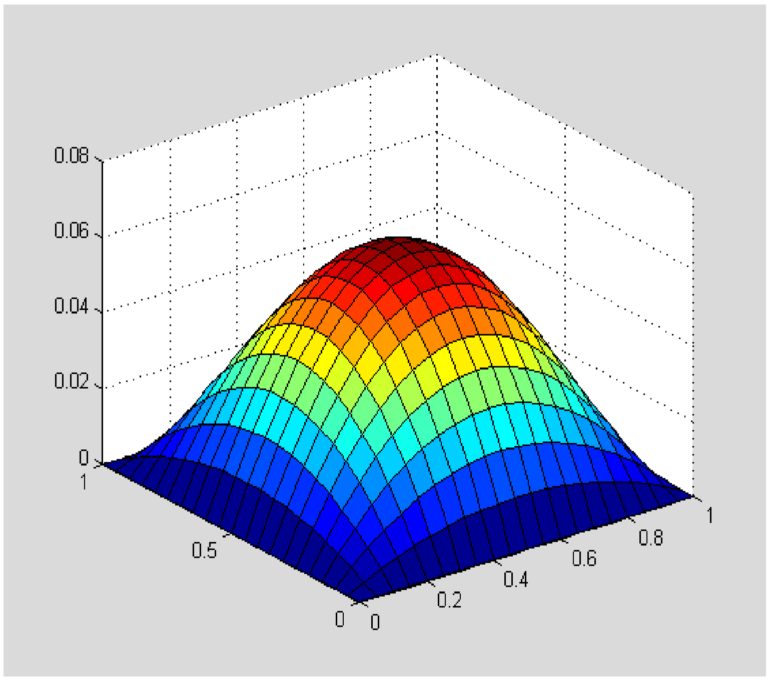
Finally, I compute the function *f*, from the previous example, on the grid:

```
Z=subs(f,{x,y},{X,Y});
```

(Notice how I can substitute for two variables at one time by listing the variables and the values between curly brackets.)

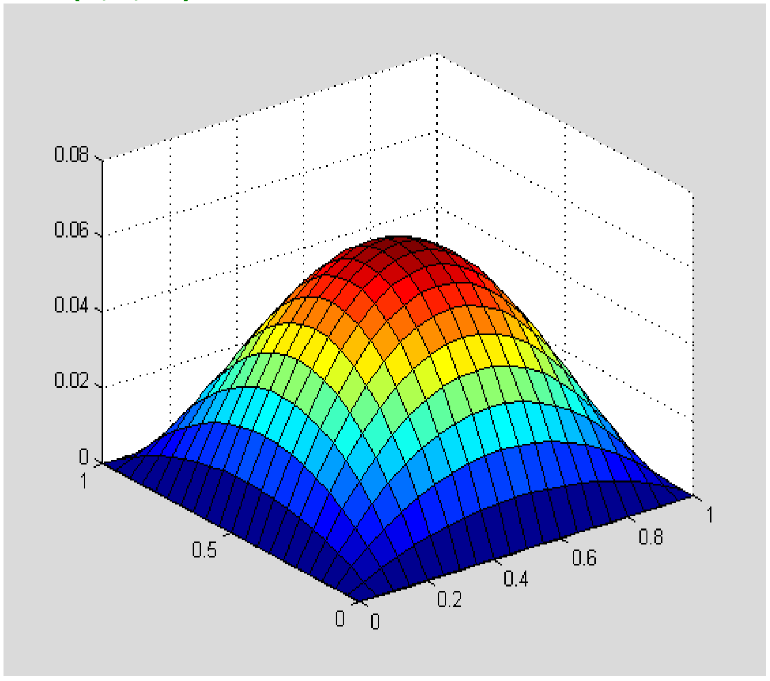
Now I invoke the **surf** command:

```
surf(X,Y,Z)
```



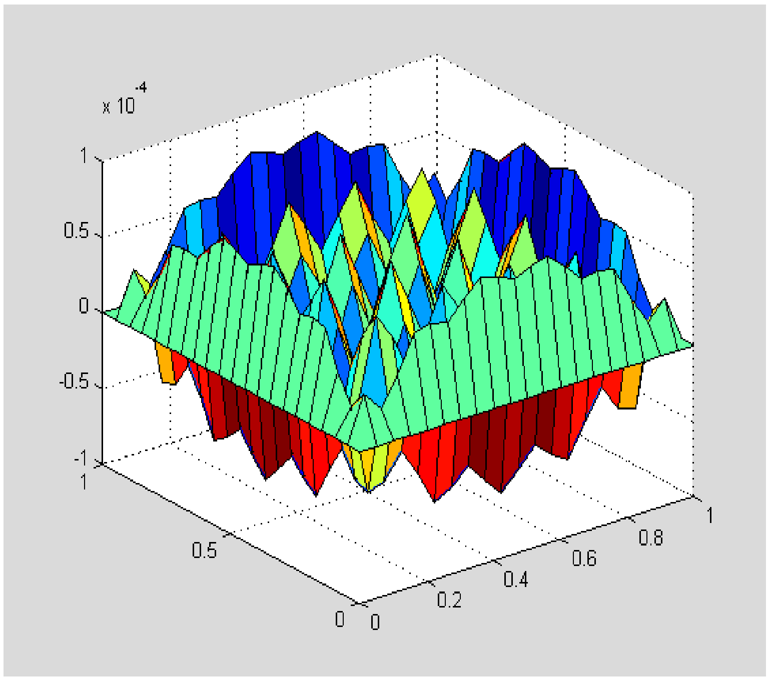
I can also evaluate the Fourier series on the two-dimensional grid. Notice that this may take some time if the grid is very fine or there are many terms in the series, or both.

```
Z1=subs(S,{x,y},{X,Y});
surf(X,Y,Z1)
```



Here is the approximation error:

```
surf(X,Y,Z-Z1)
```

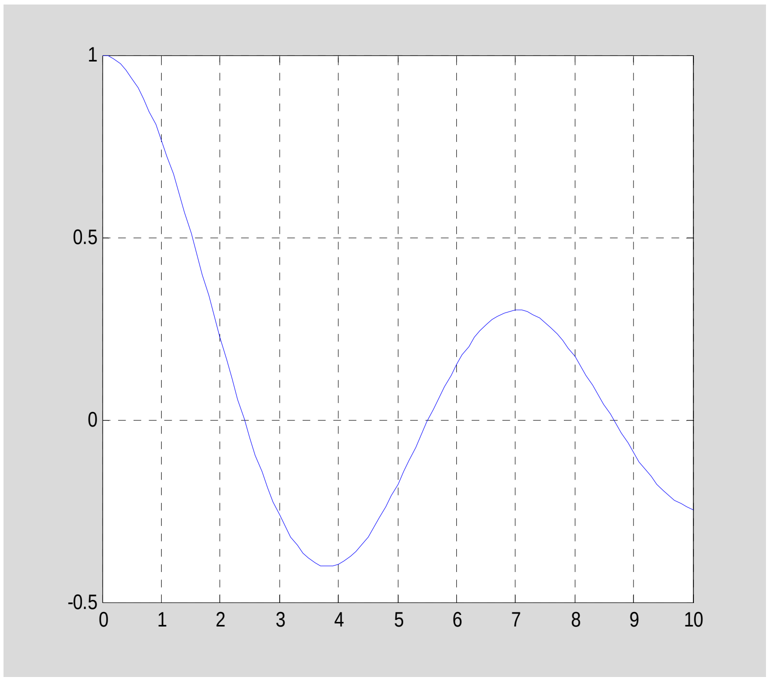


Looking at the vertical scale on the last two plots, we see that the error of approximation is only about one-half of one percent.

Section 8.3: Fourier series on a disk

The Bessel functions $J_n(s)$ are built-in functions in MATLAB, just as are the more common elementary function such as sine and cosine, and can be used just as conveniently. For example, here is the graph of J_0:

```
clear
t=linspace(0,10,101);
plot(t,besselj(0,t));grid
```



(Notice that the first argument to **besselj** is the index *n*.)

As an example, I will compute the smallest root s_{01}, s_{02}, s_{02} of J_0. The above graph shows that these roots are approximately 2.5, 5.5, and 8.5. Recall that the command **fzero** finds a (floating point estimate of) a root, near a given estimate, of a function. We have a slight difficulty in applying **fzero** to **besselj**, however, since **besselj** takes two arguments, and the first is the parameter *n*. Like most MATLAB functions that operate on functions, **fzero** will allow us to pass a parameter through to the user-defined function, but the parameter must come after the variable in the calling sequence. I will get around this problem by defining a function of one variable that represents J_0:

```
J0=@(x)besselj(0,x)
J0 =
    @(x)besselj(0,x)
```

Now I can invoke **fzero**:

```
fzero(J0,2.5)
ans =
    2.4048

fzero(J0,5.5)
ans =
    5.5201

fzero(J0,8.5)
ans =
    8.6537
```

Graphics on the disk

Functions on a disk are naturally described by cylindrical coordinates, that is, as $z = f(r,\theta)$, where (r,θ) are polar coordinates. We can use **surf** to graph such functions; however, it requires a little extra work to set up the grid. Here is how it works:

First, set up one-dimensional grids in *r* and θ (*A* is the radius of the disk):

```
clear
A=1;
r=linspace(0,A,21);
th=linspace(0,2*pi,21);
```

Next, use **meshgrid** to set up a rectangular grid in *r* and θ:

```
[R,Th]=meshgrid(r,th);
```

Now compute the function $z=f(r,\theta)$. I will use $f(r,\theta)=r\cos(\theta)$ for this example:

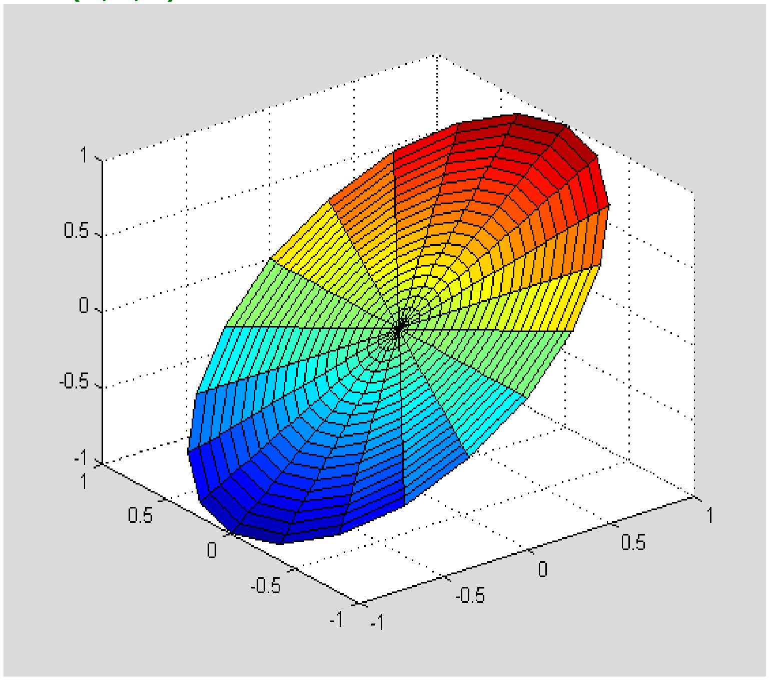
```
Z=R.*cos(Th);
```

Next, create matrices **X** and **Y** containing the (x,y) coordinates:

```
X=R.*cos(Th);
Y=R.*sin(Th);
```

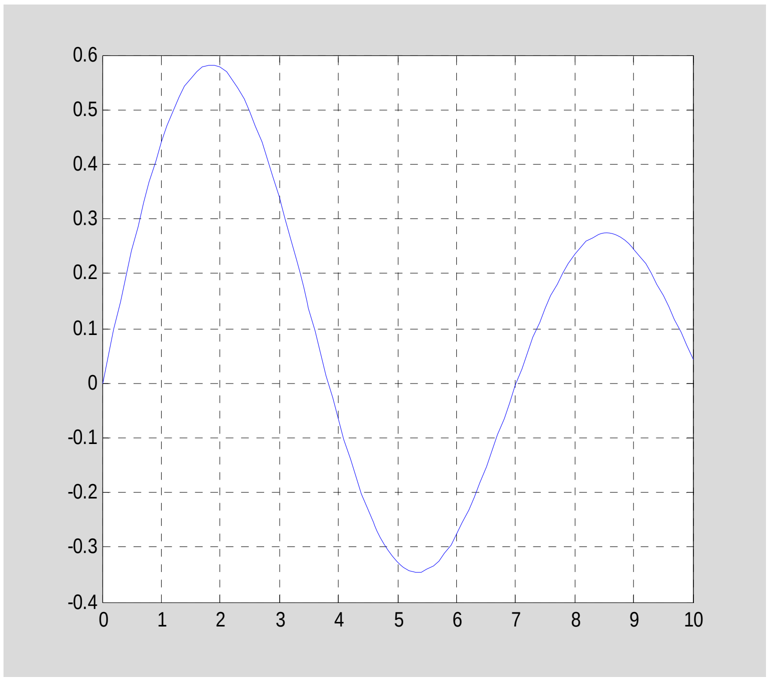
Now I can plot the surface:

```
surf(X,Y,Z)
```



Here is a more interesting example. I will graph the eigenfunction $\varphi(r,\theta)=J_1(s_{11}r)\cos(\theta)$ on the unit disk. First I must compute the root s_{11} of J_1:

```
clear
J1=@(x)besselj(1,x);
t=linspace(0,10,101)';
plot(t,J1(t));grid
```



```
s11=fzero(J1,4)
s11 =
    3.8317
```

Now I define the various grids:

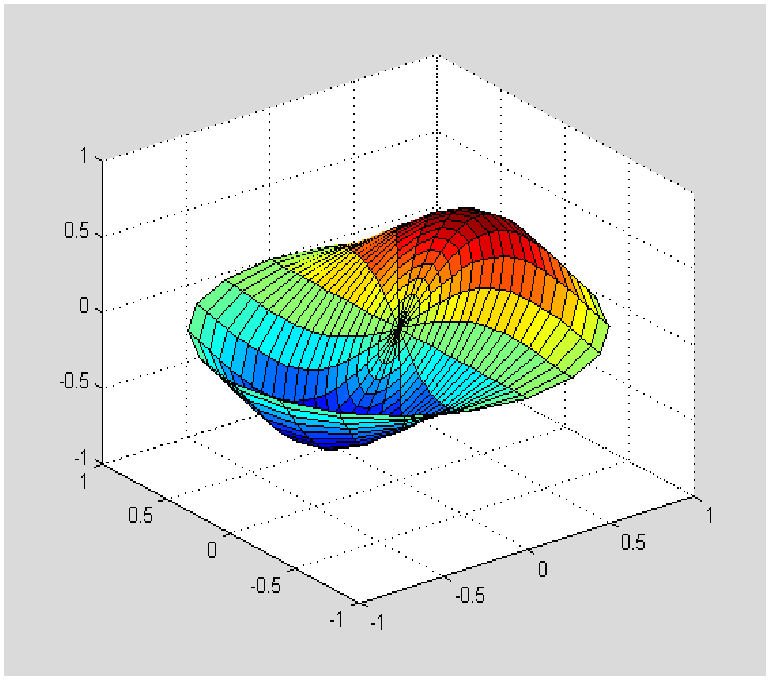
```
r=linspace(0,1,21);
th=linspace(0,2*pi,21);
[R,Th]=meshgrid(r,th);
X=R.*cos(Th);
Y=R.*sin(Th);
```

Next I define the function $z = \varphi(r,\theta)$.

```
Z=besselj(1,s11*R).*cos(Th);
```

Finally, I can graph the surface:

```
surf(X,Y,Z)
```



Chapter 12: More about Fourier series

Section 12.1: The complex Fourier series

It is no more difficult to compute complex Fourier series than the real Fourier series discussed earlier. You should recall that the imaginary unit sqrt(-1) is denoted by **i** (or **j**, but I will use **i**) in MATLAB. As an example, I will compute the complex Fourier series of $f(x)=x^2$ on the interval $[-1,1]$.

```
clear
f=inline('x.^2','x');
syms x pi n
1/2*int(f(x)*exp(-i*pi*n*x),x,-1,1)
ans =
- (exp(pi*n*sqrt(-1))*(pi^2*n^2 + 2*pi*n*sqrt(-1) - 2)*sqrt(-
1))/(2*pi^3*n^3) - ((1/exp(pi*n*sqrt(-1)))*(- pi^2*n^2 + 2*pi*n*sqrt(-
1) + 2)*sqrt(-1))/(2*pi^3*n^3)
```

We now have a simplification problem similar to what we first encountered in Chapter 5, and for which I wrote the function **mysubs**. In this case, since n is an integer, exp($-i\pi n$) equals $(-1)^n$, as does exp($i\pi n$). For this reason, I wrote another version of **mysubs**, namely **mysubs1**, to handle this case:

```
type mysubs1

function expr=mysubs1(expr,varargin)

%expr=mysubs1(expr,m,n,...)
%
%  This function substitutes (-1)^m for exp(-i*pi*m) and
%  for exp(i*pi*m), and similarly for exp(-i*pi*n) and
%  exp(i*pi*n), and any other symbols given as inputs.
```

```
syms pi i
k=length(varargin);
for jj=1:k
   expr=subs(expr,exp(-i*(pi*varargin{jj})),(-1)^varargin{jj});
   expr=subs(expr,exp(i*pi*varargin{jj}),(-1)^varargin{jj});
end
expr=simplify(expr);
```

Now I apply **mysubs1**:

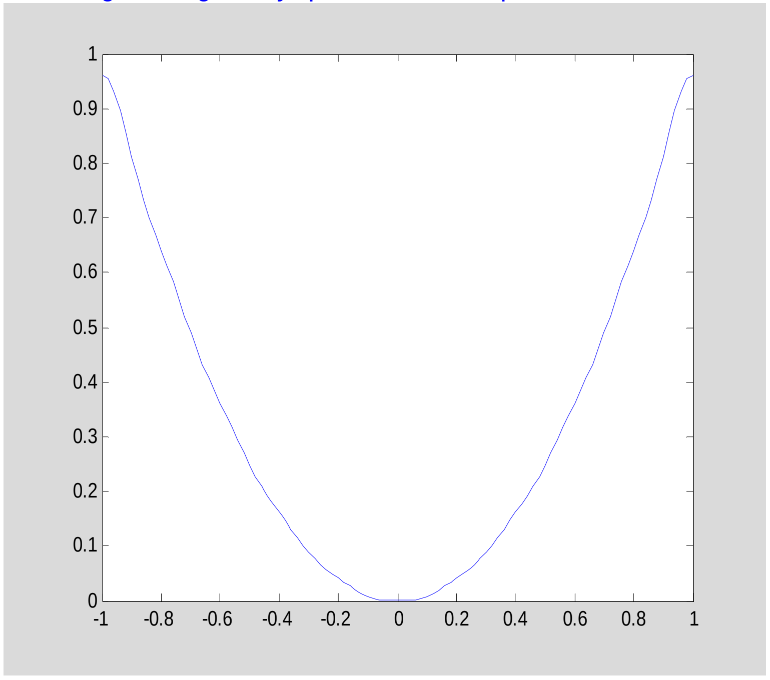
```
a=mysubs1(ans,n);
pretty(a)

        n
  2 (-1)
  -------
    2  2
  pi  n
```

This formula obviously does not hold for $n = 0$, and so I need to compute the coefficient a_0 separately:

```
a0=1/2*int(f(x),x,-1,1)
a0 =
1/3
```

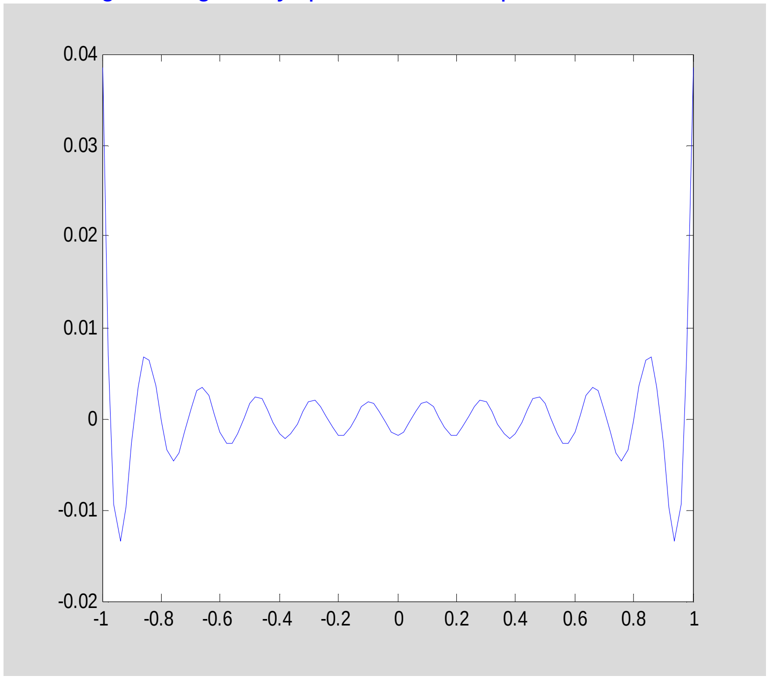
Now I can define the partial Fourier series. Recall from Section 9.1 of the text that, since *f* is real-valued, the following partial Fourier series is also real-valued (I choose to use 21 terms in the series):

```
S=a0+symsum(a*exp(i*n*pi*x),n,-10,-1)+symsum(a*exp(i*n*pi*x),n,1,10);
t=linspace(-1,1,101)';
plot(t,subs(S,x,t))
Warning: Imaginary parts of complex X and/or Y arguments ignored
```



The approximation is not bad, as the following graph of the error shows:

```
plot(t,f(t)-subs(S,x,t))
Warning: Imaginary parts of complex X and/or Y arguments ignored
```



You should notice the above warning messages complaining that the quantity I am trying to graph is complex. This is not important, because I know from the results in the text that the partial Fourier series must evaluate to a real number (since the underlying function *f* is real-valued). Therefore, any imaginary part must be due to roundoff error, and it is right to ignore it. Here is an example:

```
subs(S,x,1)
ans =
   0.9614 + 0.0000i
imag(ans)
ans =
  3.0815e-033
```

(The **imag** command extracts the imaginary part of a complex number.)

Section 9.2: Fourier series and the FFT

MATLAB implements the fast Fourier transform in the command **fft**. As mentioned in Section 9.2.3 in the text, there is more than one way to define the DFT (although all are essentially equivalent), and MATLAB's definition differs slightly from that used in the text: Instead of associating the factor of 1/M with the DFT, MATLAB associates it with the inverse DFT (cf. the formulas for the DFT and inverse DFT in Section 9.2.2 of the text). Otherwise, the definition used by MATLAB is the same as that in the text.

The inverse FFT is implemented in the command **ifft**.

As I explained in Section 9.2.2 of the text, a common operation when using the FFT is to swap the first and second halves of a finite sequence. Since this operation is so common, MATLAB implements it in the command **fftshift**:

```
help fftshift
 FFTSHIFT Shift zero-frequency component to center of spectrum.
    For vectors, FFTSHIFT(X) swaps the left and right halves of
    X.  For matrices, FFTSHIFT(X) swaps the first and third
    quadrants and the second and fourth quadrants.  For N-D
    arrays, FFTSHIFT(X) swaps "half-spaces" of X along each
    dimension.

    FFTSHIFT(X,DIM) applies the FFTSHIFT operation along the
    dimension DIM.

    FFTSHIFT is useful for visualizing the Fourier transform with
    the zero-frequency component in the middle of the spectrum.

    Class support for input X:
       float: double, single

    See also IFFTSHIFT, FFT, FFT2, FFTN, CIRCSHIFT.

    Reference page in Help browser
       doc fftshift
```

I can now reproduce Example 9.3 from the text to illustrate the use of these commands.

First, I define the function and the initial sequence:

```
clear
ff=inline('x.^3','x')
ff =
     Inline function:
     ff(x) = x.^3
x=linspace(-2/3,2/3,5);
f=[0,ff(x)]
f =
         0   -0.2963   -0.0370         0    0.0370    0.2963
```

Next, I shift the sequence using **fftshift**, and then apply **fft**. Recall that, to reproduce the results in the text, I must use the same definition for the DFT, which means that I must divide the output of **fft** by 6, the length of the sequence:

```
f1=fftshift(f)
f1 =
         0    0.0370    0.2963         0   -0.2963   -0.0370
F1=fft(f1)/6
F1 =
  Columns 1 through 4
        0                  0 - 0.0962i        0 + 0.0748i        0
  Columns 5 through 6
        0 - 0.0748i        0 + 0.0962i
```

Now I apply **fftshift** to F1 to obtain the desired sequence F:

```
F=fftshift(F1)
F =
  Columns 1 through 4
        0                  0 - 0.0748i        0 + 0.0962i        0
  Columns 5 through 6
```

```
        0 - 0.0962i        0 + 0.0748i
```

The result is the same as in the text, up to round-off error.

I can perform the entire computation above in one line as follows:

```
F=fftshift(fft(fftshift(f))/6)
F =
  Columns 1 through 4
        0                  0 - 0.0748i        0 + 0.0962i        0
  Columns 5 through 6
        0 - 0.0962i        0 + 0.0748i
```

Chapter 13: More about finite element methods

Section 13.1 Implementation of finite element methods

In this section, I do more than just explain MATLAB commands, as has been my policy up to this point. I define a collection of MATLAB M-file functions that implement piecewise linear finite elements on two-dimensional polygonal domains. The interested reader can experiment with and extend these functions to see how the finite element method works in practice. The implementation of the finite element method follows closely the explanation I give in Section 13.1 of the text, although the data structure has been extended to allow the code to handle inhomogeneous boundary conditions. (However, the code itself has not been extended to handle inhomogeneous boundary conditions. This extension has been left as an exercise.)

The main commands are **Stiffness1** and **Load1**, which assemble the stiffness matrix and load vectors for the BVP

$$-\nabla\cdot(a(x,y)\nabla u) = f(x,y) \text{ in } \Omega,$$
$$u = g \text{ on } \Gamma,$$
$$\frac{du}{dn} = h \text{ on } \partial\Omega\backslash\Gamma,$$

where Γ is part of the boundary of Ω and $\partial\Omega\backslash\Gamma$ is the rest of the boundary. However, before any computation can be done, the mesh must be described. I provide three routines for creating rectangular meshes, **RectangleMeshD1, RectangleMeshN1,** and **RectangleMeshTopD1** (these routines differ in the boundary conditions that are assumed).

Creating a mesh

The data structure is described in the M-file M**esh1.m** (this M-file contains only comments, which are displayed when you type "**help Mesh1**"):

```
addpath('fempack')
help Mesh1
  The Fem1 package implements piecewise linear finite elements for two
  dimensional problems and is intended to accompany "Partial
  Differential Equations: Analytical and Numerical Methods" (second
  edition) by Mark S. Gockenbach (SIAM 2010).  It uses the data
  structure described in Section 13.1 of the text (that is, the four
  arrays NodeList, NodePtrs, FNodePtrs, and ElList, augmented by
  three more arrays (CNodePtrs, ElEdgeList, and FBndyList) to allow
  the code to handle inhomogeneous boundary conditions (as hinted at
  in the text).  These seven arrays are collected into a structure.

     NodeList: Mx2 matrix, where M is the number of nodes in the mesh
               (including boundary nodes).  Each row corresponds to a
               node and contains the coordinates of the node.

     NodePtrs: Mx1 matrix.  Each entry corresponds to a node:
                if node i is free, NodePtrs(i) is the index of node i
                in FNodePtrs; else NodePtrs(i) is the negative of
                index of node i in CNodePtrs

     FNodePtrs: Nx1 vector, where N is the number of free nodes.
                NodePtrs(i) is the index into NodeList of the ith
                free node.  So, for example, NodeList(NodePtrs(i),1:2)
                are the coordinates of the ith free node.

     CNodePtrs: Kx1 vector, where K=M-N is the number of constrained
                nodes.  CNodePtrs(i) is the index into NodeList of the
                ith constrained node.

     ElList: Lx3 matrix, where L is the number of triangular elements
             in the mesh.  Each row corresponds to one element and
             contains pointers to the nodes of the triangle in
             NodeList.

     ElEdgeList: Lx3 matrix.  Each row contains flags indicating
                 whether each edge of the corresponding element is on
                 the boundary (flag is -1 if the edge is a constrained
                 boundary edge, otherwise it equals the index of the
                 edge in FBndyList) or not (flag is 0).  The edge
                 of the triangle are, in order, those joining vertices
                 1 and 2, 2 and 3, and 3 and 1.

     FBndyList: Bx2 matrix, where B is the number of boundary edges
                not constrained by Dirichlet conditions.  Each
                row corresponds to one edge and contains pointers
                into NodeList, yielding the two vertices of the edge.
```

The command **RectangleMeshD1** creates a regular triangulation of a rectangle

$$[0,l_x]\times[0,l_y],$$

assuming Dirichlet conditions on the boundary.

```
help RectangleMeshD1
  T=RectangleMeshD1(nx,ny,lx,ly)

    This function creates a regular, nx by ny finite element mesh for
    a Dirichlet problem on the rectangle [0,lx]x[0,ly].

    The last three arguments can be omitted; their default values are
    ny=nx, lx=1, ly=1.  Thus, the command "T=RectangleMeshD1(m)"
    creates a regular mesh with 2m^2 triangles on the unit square.

    For a description of the data structure describing T, see
    "help Mesh1".
```

The command **RectangleMeshN1** creates a regular triangulation of the same rectangle, assuming Neumann conditions on the boundary, while **RectangleMeshTopD1** imposes Dirichlet conditions on the top edge and Neumann conditions on the other three edges.

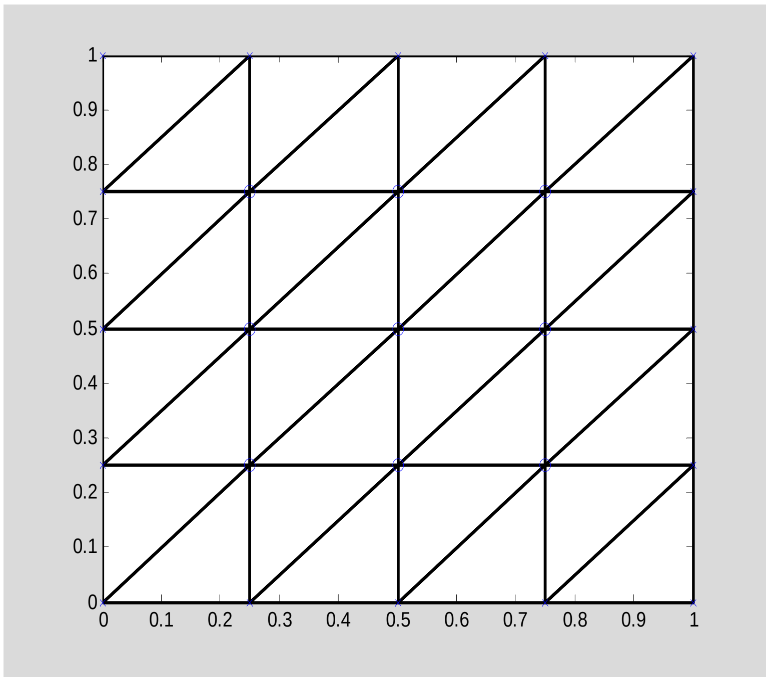
Thus I only provide the means to deal with a single domain shape (a rectangle), and only under three combinations of boundary conditions. To use my code to solve BVPs on another domain, you will have to write code to generate the mesh yourself. Note, however, that the rest of the code is general: it handles any polygonal domain, with any combination of homogeneous Dirichlet and Neumann conditions.

Here I create a mesh:

```
clear
T=RectangleMeshD1(4)
T =
      NodeList: [25x2 double]
      NodePtrs: [25x1 double]
     FNodePtrs: [9x1 double]
     CNodePtrs: [16x1 double]
        ElList: [32x3 double]
    ElEdgeList: [32x3 double]
     FBndyList: []
```

The mesh I just created is shown in Figure 10.1 in the text. I have provided a means of viewing a mesh:

```
help ShowMesh1
  ShowMesh1(T,flag)

    This function displays a triangular mesh T.  Unless the flag
    is nonzero, free nodes are indicated by a 'o', and
    constrained nodes by an 'x'.

    For a description of the data structure T, see "help Mesh1".

    The optional argument flag has the following effect:

       flag=1: the triangles are labeled by their indices
       flag=2: the nodes by their indices
       flag=3: both the nodes and triangles are labeled
       flag=4: the free nodes are labeled by their indices.
       flag=5: the free and constrained nodes are labeled by
               their indices.

ShowMesh1(T)
```



(Notice that it is also possible to label the triangles and/or nodes by their indices.)

Computing the stiffness matrix and the load vector

Here are the main commands:

```
help Stiffness1
  K=Stiffness1(T,fnk)

    Assembles the stiffness matrix for the PDE

         -div(k*grad u)=f in Omega,
              u=0 on Gamma,
              du/dn=0 on Bndy(Omega)-Gamma.

    The input fnk can be a (positive) number k or a function
    implementing a function k(x,y).  If fnk is omitted, it is
    assumed to be the constant 1.  T describes the triangulation
    of Omega.  For a description of the data structure T, see
    "help Mesh1".

help Load1
  F=Load1(T,fnf)

    Assembles the load vector for the BVP

         -div(a*grad u)=f in Omega,
              u=0 on Gamma,
              du/dn=0 on Bndy(Omega)-Gamma.

    The right hand side function f(x,y) must be implemented in the
    function fnf.
```

Thus, to apply the finite element method to the BVP given above (assuming homogeneous boundary conditions), I need to define the coefficient $a(x,y)$ and the forcing function $f(x,y)$. As an example, I will reproduce the computations from Example 8.10 from the text, in which case $a(x,y)=1$ and $f(x,y)=x$.

```
a=1;
f=@(x,y)x;
```

Now I compute the stiffness matrix **K** and the load vector **F**:

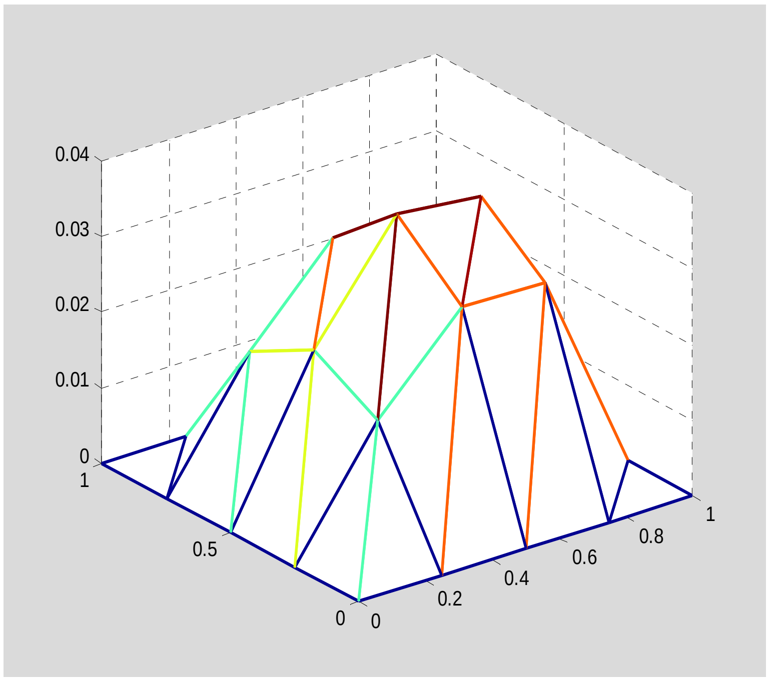
```
K=Stiffness1(T,a);
F=Load1(T,f);
```

Finally, I solve the system **Ku**=**F** to get the nodal values:

```
u=K\F;
```

Given the vector of nodal values (and the mesh), you can graph the computed solution using the **ShowPWLinFcn1** command:

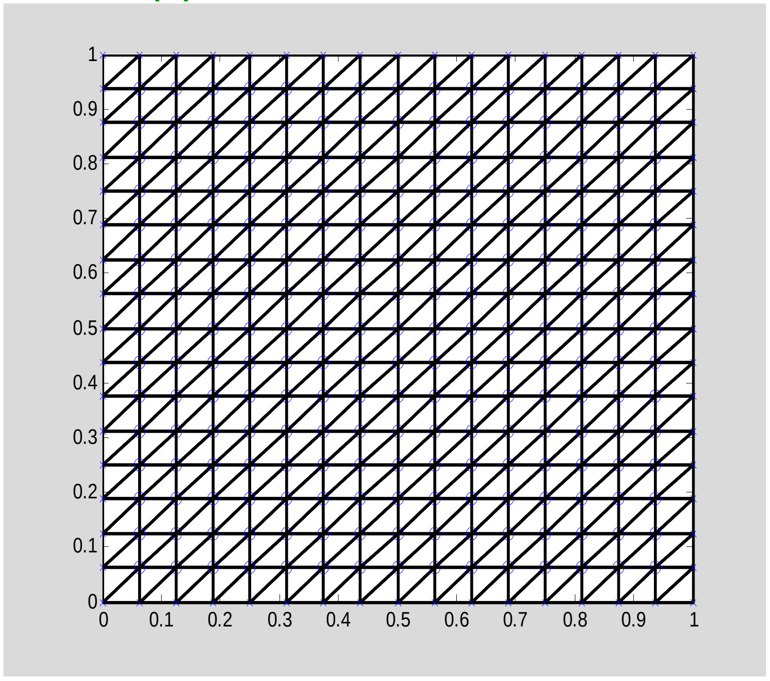
```
help ShowPWLinFcn1
  ShowPWLinFcn1(T,U,g)

    This function draws a surface plot of a piecewise linear
    function defined on a triangular mesh.  The inputs are T,
    the mesh (see "help Mesh1" for details about this data structure)
    and the vector U, giving the nodal values of the function
    (typically U would be obtained by solving a finite element
    equation).

    The optional input g is a vector of nodal values at the
    constrained nodes.  If g is not given, this vector is taken
    to be zero.

ShowPWLinFcn1(T,u)
```



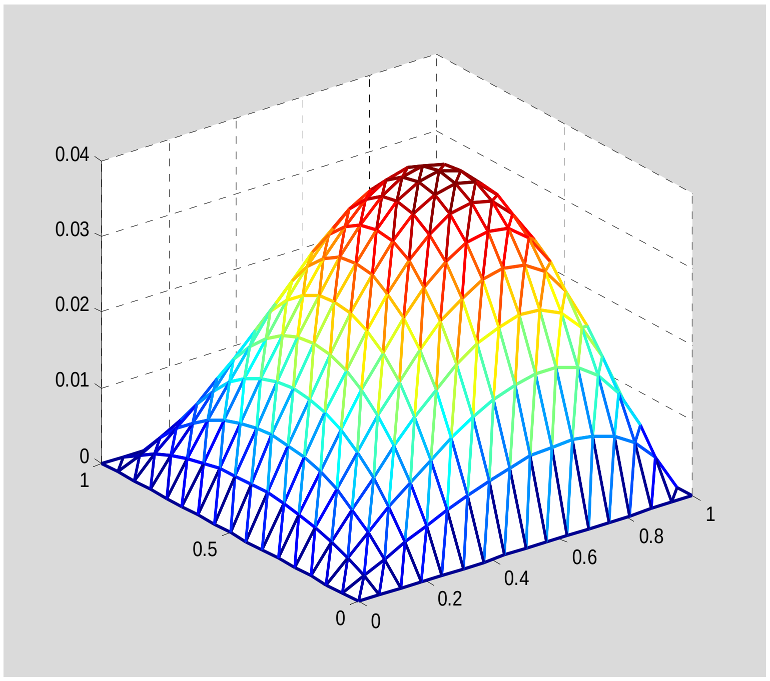
The above solution does not look very good (not smooth, for instance); this is because the mesh is rather coarse. I will now repeat the calculation on a finer mesh.

```
T=RectangleMeshD1(16);
```

```
ShowMesh1(T)
```

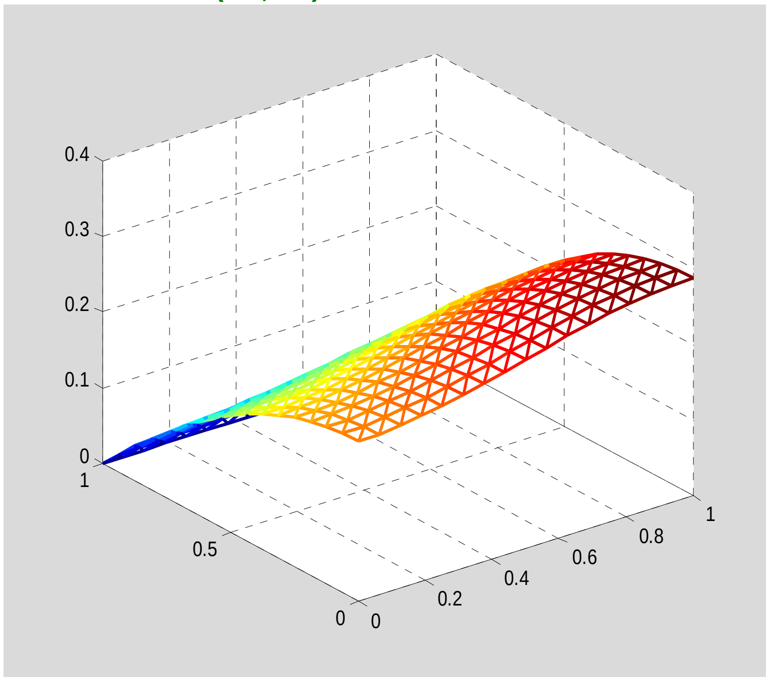


```
K=Stiffness1(T,a);
 F=Load1(T,f);
 u=K\F;
ShowPWLinFcn1(T,u)
```



For the sake of illustrating mixed boundary conditions, I will solve the same PDE with mixed boundary conditions:

```
T1=RectangleMeshTopD1(16);
K1=Stiffness1(T1,a);
F1=Load1(T1,f);
u1=K1\F1;

ShowPWLinFcn1(T1,u1)
```



There are other utility routines included, such as routines for estimating the L^2 and energy norms. For a complete list of routines provided, type "**help Fem1**".

Testing the code

To see how well the code is working, I can solve a problem with a known solution, and compare the computed solution with the exact solution. I can easily create a problem with a known solution; I just choose $a(x,y)$ and any $u(x,y)$ satisfying the boundary conditions, and then compute

$$f(x,y) = -\nabla \cdot \left(a(x,y)\nabla u\right)$$

to get the right-hand side f. For example, suppose I take

```
clear
a=@(x,y)1+x.^2;
u=@(x,y)x.*(1-x).*sin(pi*y);
```

(Notice that u satisfies homogeneous Dirichlet conditions on the unit square.) Then I can compute f:

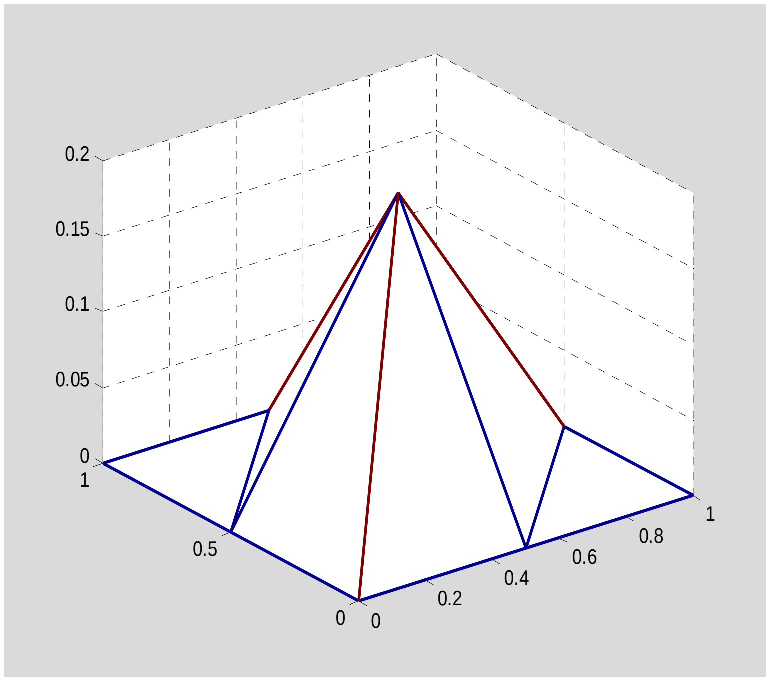
```
syms x y
-diff(a(x,y)*diff(u(x,y),x),x)-diff(a(x,y)*diff(u(x,y),y),y)
ans =
2*sin(pi*y)*(x^2 + 1) + 2*x*(x*sin(pi*y) + sin(pi*y)*(x - 1)) -
pi^2*x*sin(pi*y)*(x^2 + 1)*(x - 1)
cmd=['f=@(x,y)',char(ans)];
eval(cmd)
f =
    @(x,y)2*sin(pi*y)*(x^2+1)+2*x*(x*sin(pi*y)+sin(pi*y)*(x-1))-
pi^2*x*sin(pi*y)*(x^2+1)*(x-1)
```

Now I will create a coarse mesh and compute the finite element approximation:

```
T=RectangleMeshD1(2);
K=Stiffness1(T,a);
F=Load1(T,f);
 U=K\F;
```

Here is the approximate solution:

```
ShowPWLinFcn1(T,U)
```



(The mesh is so coarse that there is only one free node!)

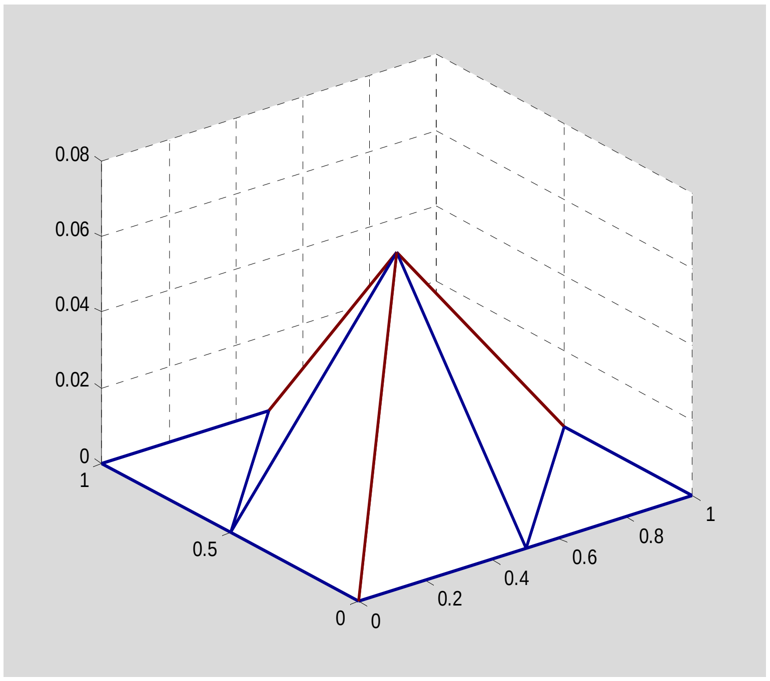
For comparison purposes, let me compute the nodal values of the exact solution on the same mesh. I have provided a command to do this:

```
help NodalValues1
 [v,g]=NodalValues1(T,u)

   This function sets v equal to the vector of
   values of u(x,y) at the free nodes of the mesh T,
   and g equal to the vector of values of u(x,y)
   at the constrained nodes of the mesh T.
   The function implementing u must be vectorized.

   See "help Mesh1" for a description of the
   data structure for T.
```

```
V=NodalValues1(T,u);
```

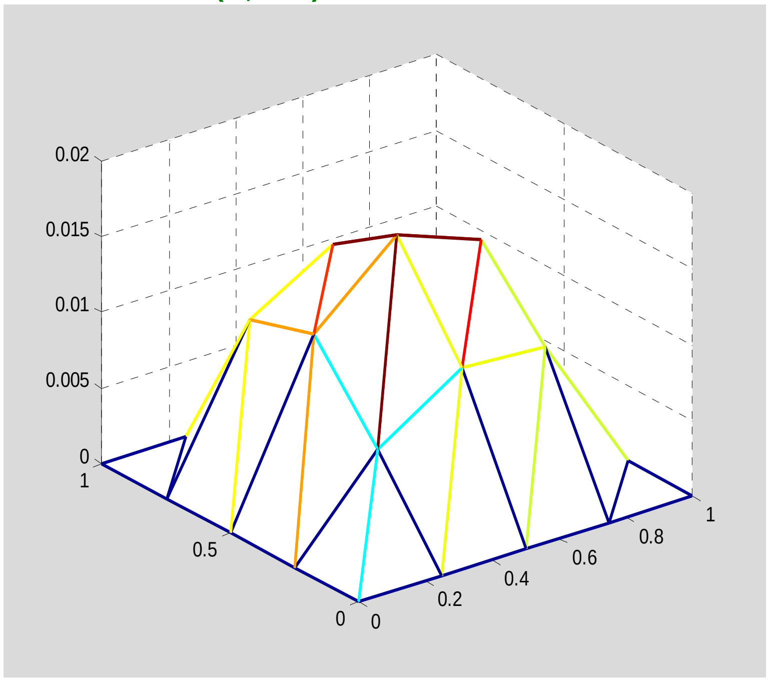
Now I will plot the difference between the computed solution and the piecewise linear interpolant of the exact solution (notice the scale on the graph):

```
ShowPWLinFcn1(T,V-U)
```



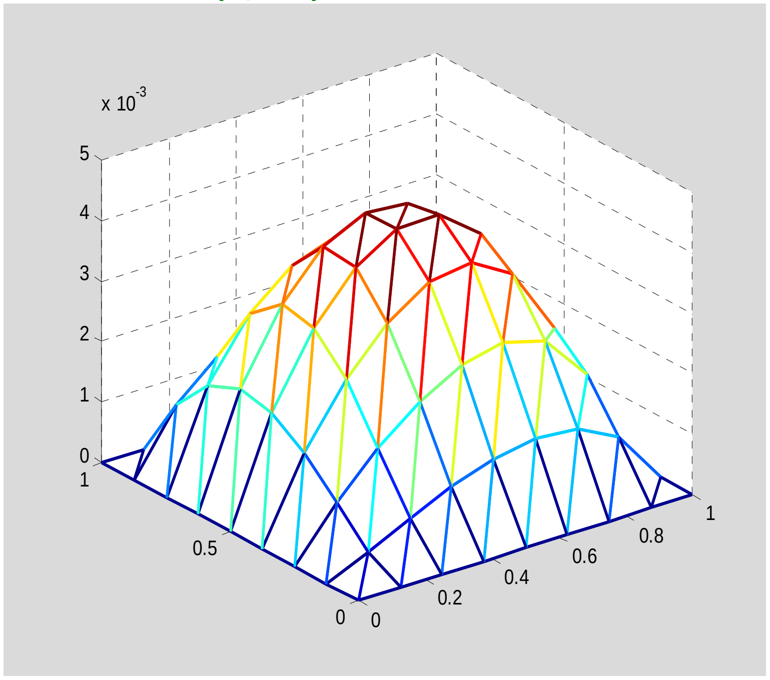
I now repeat with a sequence of finer meshes:

```
T=RectangleMeshD1(4);
K=Stiffness1(T,a);
F=Load1(T,f);
U=K\F;
V=NodalValues1(T,u);
ShowPWLinFcn1(T,V-U)
```

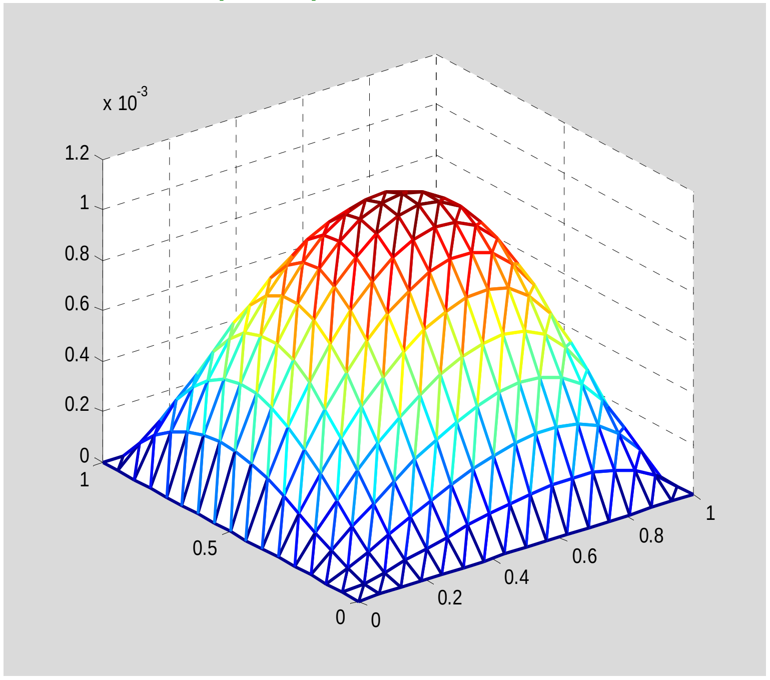


```
T=RectangleMeshD1(8);
```

```
K=Stiffness1(T,a);
F=Load1(T,f);
U=K\F;
V=NodalValues1(T,u);
ShowPWLinFcn1(T,V-U)
```



```
T=RectangleMeshD1(16);
K=Stiffness1(T,a);
F=Load1(T,f);
U=K\F;
V=NodalValues1(T,u);
ShowPWLinFcn1(T,V-U)
```



Using the code

My purpose for providing this code is so that you can see how finite element methods can be implemented in practice. To really benefit from the code, you should extend its capabilities. By writing some code yourself, you will learn how such programs are written. Here are some projects you might undertake, more or less in order of difficulty:

1. Write a command called **Mass1** that computes the mass matrix. The calling sequence should be simply **M=Mass1(T)**, where T is the triangulation.
2. Choose some other geometric shapes and/or combinations of boundary conditions, and write some mesh generation routines analogous to **RectangleMeshD1**.
3. Extend the code to handle inhomogeneous Dirichlet conditions. Recall that such boundary conditions change the load vector, so the routine **Load1** must be modified.
4. Extend the code to handle inhomogeneous Neumann conditions. Like inhomogeneous Dirichlet conditions, the load vector is affected
5. (**Hard**) Write a routine to refine a given mesh, according to the standard method suggested in Exercise 10.1.4 of the text.

As I mentioned above, the mesh data structure described in **Mesh1.m** includes the information needed to solve exercises 3, 4, and 5.