Mathematica Tutorial

To accompany

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

by

Mark S. Gockenbach

(SIAM, 2010)

Introduction

In this introduction, we explain the organization of this tutorial and give some basic information about *Mathematica* and *Mathematica* notebooks. We also give a preliminary introduction to the capabilities of *Mathematica*.

■ About this tutorial

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

Therefore, for example, in Section 2.1, *Heat flow in a bar; Fourier's Law,* there is no discussion of physics or modeling. (The physics and modeling are found in the text.) Instead, the *Mathematica* command for integration is presented, because Section 2.1 is the first place in the text where you are 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 *Mathematica* 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.

Our purpose in 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, you will not find a complete example of every problem type; otherwise, your "studying" could degenerate to simply looking for an example, copying it, and making a few changes. At crucial points, we do provide some complete examples, since we see no other way to illustrate the power of *Mathematica* than in context. However, there is still plenty for you to figure out for yourself.

Getting help about commands

Help on *Mathematica* commands is always available through the help browser. The browser provides several ways to access the information. We recommend choosing **Wolfram Documentation** from the **help** menu, after which you have two options:

1. Choose one of the listed options (e.g. “graphics and visualization”).

2. Enter a command name or query (e.g. “LinearSolve” or “solving linear systems”) in the search bar.

■ About *Mathematica*

Mathematica is the creation of Stephen Wolfram, a theoretical physicist who has made important contributions to mathematics and computer science. Wolfram describes *Mathematica* as "the world's only fully integrated environment for technical computing." At the heart of *Mathematica* is a computer algebra system, that is, a system for doing algebraic manipulations symbolically (and therefore exactly). However, *Mathematica* also incorporates floating point (or finite precision) computation, arbitrary precision arithmetic, graphics, and text processing. It is also a programming environment. We will touch on all of these capabilities in this tutorial.

■ *Mathematica* notebooks

This document you are reading is called a *notebook*. (Here we assume that you are reading this file in *Mathematica*, not as a printed document. If you are reading a printed copy, you will have to ignore a few comments about how *Mathematica* displays a notebook.) It consists of both text and *Mathematica* input and output, organized in *cells*. You are currently reading a text cell; the next section presents some input and output cells. The most important thing to understand about a notebook is that it is interactive: at any time you can execute a *Mathematica* command and see what it does. This makes a *Mathematica* notebook a tremendous learning environment: when you read an explanation of a *Mathematica* feature, you can immediately try it out.

■ Getting started with *Mathematica*

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

```
In[1]:= 2 + 2
Out[1]= 4
```

The input cell (labeled by In[1]:=) contains the expression 2+2, which *Mathematica* evaluates, returning the result (4) in the output cell (indicated by Out[1]=). The user types "2+2"; *Mathematica* automatically supplied the label "In[1]:=". Looking to the far right of this document, you will see the brackets that indicate the grouping of the material into the cells. (You will not see the brackets when the notebook is printed.) Moreover, the cells are nested. For example, the input and output cells are grouped together in an input/output cell, which is grouped together with the text cells and more input/output cells into this section of the document. Several sections are grouped together into this introductory chapter. Finally, all of the chapters are grouped in a single cell, the notebook. Below we discuss some elementary manipulations of cells, including the creation of text cells.

By default, when you type something in a *Mathematica* notebook, it is regarded as input. (You have to give a special command, as explained below, to create a text cell, a heading, or something else.) Here is how you create an input cell:

Start by using the mouse to move the cursor over these words you are reading (do it now!). The cursor will look like a vertical bar with little sticks at the top and bottom, almost like a capital I. Then move the cursor between this text cell and the next. You should see the cursor become horizontal. (If you go too far, into the next text cell, the cursor will become vertical again.) Clicking while the cursor is horizontal causes a horizontal line, extending across the entire document, to appear. You can now enter input. For example, type 3+3, followed by shift/return (that is, press return while holding down the shift key). If your keyboard has both a return key and an enter key, then the enter key by itself is equivalent to shift/return. (On the other hand, your computer may have only an enter key, in which case it is equivalent to a return key, and you must push shift/enter to tell *Mathematica* to execute a command.)

You should have noticed the following: when you pressed shift/return, *Mathematica* inserted "In[2]:=" before the "3+3" you typed, and then displayed "Out[2]=6" on the next line. Here is how it should appear:

```
In[2]:= 3 + 3
Out[2]= 6
```

(However, the indexing of the input and output cells might be different, depending on what you have done so far. For instance, when you typed "3+3" and pressed shift/return, *Mathematica* might have displayed "In[1]:=" instead of "In[2]:=".)

Now that you know how to enter commands and see the output, let's quickly go over some of the most basic capabilities of *Mathematica*. First of all, *Mathematica* can do arithmetic with integers and rational numbers exactly, regardless of the number of digits involved:

```
In[3]:= 123^45
Out[3]= 11 110 408 185 131 956 285 910 790 587 176 451 918 559 153 212 268 021 823 629 073 199 866 111 \
         001 242 743 283 966 127 048 043
```

```
In[4]:= 115 / 39 + 727 / 119
```

Out[4]= $\frac{42038}{4641}$

Mathematica knows the standard elementary functions, such as the trigonometric functions, logarithms and exponentials, the square root function, and so forth. It also knows the common mathematical constant π. Consider the following calculation:

```
In[5]:= Sin[Pi / 4]
```

Out[5]= $\frac{1}{\sqrt{2}}$

There are several important things to learn from this example. First of all, π is typed with the first letter capitalized, as is the built-in function Sin. In fact, every predefined symbol in *Mathematica* begins with a capital letter. One advantage of this is that you can define your own symbols beginning with lowercase letters, knowing that you will not conflict with any of *Mathematica*'s names.

Another thing to notice from the previous example is that *Mathematica* knows that the sine of $\pi/4$ is exactly $1/\sqrt{2}$; it does not return an estimate like 0.70710678, as a handheld calculator might. Now compare the previous example with this:

```
In[6]:= Sin[Pi / 4.0]
Out[6]= 0.707107
```

The only difference between the two examples is that the integer 4 in the first was replaced by the floating point number 4.0 in the second. This difference was enough to cause *Mathematica* to return a floating point (approximate) result in the

second example.

Here is an important rule: An expression involving only exact (symbolic) quantities will be evaluated exactly (symbolically), while an expression involving one or more floating point numbers will be evaluated approximately using floating point arithmetic. Here are two more examples to illustrate this rule:

```
In[7]:= Sqrt[2]
Out[7]= √2

In[8]:= Sqrt[2.0]
Out[8]= 1.41421
```

(This example also introduces the square root function, spelled "Sqrt".)

Here are a few more sample calculations.

```
In[9]:= (100 - 9) (100 + 9)
Out[9]= 9919

In[10]:= (-5 + Sqrt[5^2 - 4 * 1 * 4]) / 2
Out[10]= -1

In[11]:= (-1) ^2 + 5 (-1) + 4
Out[11]= 0
```

You should learn several things from these examples:

A power is formed using the "^" character.

Parentheses are used for algebraic grouping, as in the expression (100-9)(100+9).

Multiplication can be indicated either by the "*" character, or simply by juxtaposition (with a space between the symbols, if necessary).

The last point is illustrated in the following examples:

```
In[12]:= 2 * 2
Out[12]= 4

In[13]:= 2 × 2
Out[13]= 4
```

(Note that *Mathematica* automatically put in the times symbol; we just typed a pair of 2's, separated by a space.)

```
In[14]:= 2 * x - 2 x
Out[14]= 0

In[15]:= (-1) (-1)
Out[15]= 1
```

One will often wish to perform a calculation in several steps. *Mathematica* makes it possible to refer to previous results in a couple of ways. The "%" character always represents the last output:

```
In[16]:= 119 / 11 + 47 / 13
```

Out[16]= $\frac{2064}{143}$

```
In[17]:= % - 107 / 23
```

Out[17]= $\frac{32171}{3289}$

To refer to an earlier output (not just the most recent), use the "Out" keyword, followed by the index of the output. For example, Out[1] always refers to the first output of your *Mathematica* session:

```
In[18]:= Out[1]
Out[18]= 4
```

You can evaluate a symbolic expression in floating point using the N operator:

```
In[19]:= 129 / 9
```

Out[19]= $\frac{43}{3}$

```
In[20]:= N[%]
Out[20]= 14.3333
```

The N operator can perform calculations in any precision; you just follow the expression to be evaluated by the desired number of digits:

```
In[21]:= N[Pi, 100]
Out[21]= 3.141592653589793238462643383279502884197169399375105820974944592307816406286208 9
          98628034825342117068
```

■ A few more points about *Mathematica* notebooks

Text cells and headings

Eventually, you may wish to produce your own notebooks, perhaps when writing up solutions to homework. You can create a document like this one, with headings, subheadings, and text cells (in addition to input/output cells), using the **Style** option from the **Format** menu at the top. For example, to create a text cell (like this one), click between cells (or after the last cell in your document), so that *Mathematica* is prepared to receive input. Then (using the mouse) go to the **Format** menu, click on **Style**, and then select **Text** from the list of options. Whatever you then type in the cell will be in text format.

Headings are produced in exactly the same way. Go to **Format**, then **Style**, and select **Title**, **Subtitle**, **Subsubtitle**, **Section**, **Subsection**, or **Subsubsection**.

Initializing a notebook

When you first open a notebook that contains input/output cells, *Mathematica*'s kernel is not aware of those cells unless you execute them. For example, consider the following input/output cell:

```
In[22]:= a = 13
Out[22]= 13
```

What would happen if you moved your cursor after this cell, and tried to use the value of a (for example, you type a^2, expecting to get the value 169)? If you do not first initialize the notebook by evaluating the input cells, the kernel will *not* record that a has been assigned the value 13. One way to initialize a notebook is to simply move your cursor to each input cell in turn and press shift/return. The kernel will then execute the commands. For the purposes of this tutorial, this is a good way to proceed. Then the state of the *Mathematica* kernel is exactly as it appears as you read through the document. (You may wish to go back to the beginning and initialize the cells up to this point.) You may already have noticed that, when you issue a command in an uninitialized notebook, *Mathematica* prompts you as to whether you wish to initialize the entire notebook before proceeding (it may not do this on every computer system). If you wish to go through the tutorial sequentially, it is better to not initialize all of the cells at the beginning.

It may be preferable, in some cases, to initialize the entire notebook at the beginning. You can do this by going to the **Evaluation menu** and selecting **Evaluate notebook**. If this is how you want to initialize the kernel, then you should do it at the beginning of your session.

Saving a notebook

When you prepare a homework solution in *Mathematica*, or do some other work that you want to save for later reference, you must save the contents of the notebook. To save the document, go the **File** menu, select the **Save** option, and then enter a file name. By default, *Mathematica* will save the document in notebook format, that is, as a ".nb" file. Thus, if you enter the name "hw1", your file will be saved as "hw1.nb". (You can choose a different format (e.g. PDF) using the drop-down menu.) Thereafter, whenever you make changes to the notebook and save it, you will not need to enter the file name. As you work on your notebook, you should frequently save it, so that, if something goes wrong, you will never lose much work.

As you work your way through this tutorial, you will want to stop at times and come back to it later. At those times, you will have to decide if you wish to save the changes you have made or not. You may wish to save the tutorial with your modifications under a different name, so that you keep a clean copy of the original tutorial.

Manipulating cells

For the purposes of organization and appearance, the contents of cells can be hidden. For example, below is a section of text, grouped with a heading, that has been hidden. Only the heading is seen, and, if you look to the right, you will see that the cell delimiter has a little "flag" indicating that the contents of the cell are hidden. You can open the cell by moving your cursor to the cell delimiter with the flag and double clicking on it.

A sample closed cell

This is a text cell that was originally hidden.

The first time you open this notebook, the Introduction should be revealed, but the subsequent chapters, and the sections within the chapters, should be hidden. You will have to open them to read them. If you save the notebook after you read and possibly modify it, then the cells will be in the same state (open or closed) as you left them.

Chapter 1: Classification of Differential Equations

Mathematica allows us to define functions and compute their derivatives symbolically. Using these capabilities, it is usually straightforward to verify that a given function is a solution to a differential equation.

Example

Suppose you wish to verify that

$$u(t) = e^{at}$$

is a solution to the ordinary differential equation

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

First define the function:

```
In[23]:= u[t_] = E^(a*t)
Out[23]= e^13 t
```

Here we have intentionally illustrated a common mistake: the symbol "a" was previously assigned a value (13), but now we want to use it as an (indeterminate) parameter. The previous value must be cleared before the symbol is used again. It is a good idea to clear the values of any symbols that are about to be defined or are about to be used as variables. This is accomplished with the **ClearAll** command. There is no harm in clearing a symbol that has not been given a value yet, so it is a good idea to use **ClearAll** liberally. Doing so will eliminate many errors that may otherwise be difficult to find. (A related command is **Clear**, which clears the definition of a symbol but not necessarily all of the information associated with it. In almost every case, **Clear** is sufficient, but, on the other hand, in most cases, **ClearAll** is what you really want to do. We recommend the use of **ClearAll**.)

Here is the appropriate way to define $u(t)$:

```
In[24]:= ClearAll[u, t, a]
         u[t_] = E^(a*t)
Out[25]= e^a t
```

There are several important things to learn from this definition:

Function evaluation in Mathematica is indicated by square brackets. That is, while in mathematical notation, we write $f(x)$, in Mathematica the correct syntax is **f[x]**.

Ordinary parentheses are used exclusively for algebraic grouping. Thus we write **(a*t)** to indicate that the exponent is the product of a and t.

The value $e = 2.71828\ldots$ is a built-in constant, denoted by the capital **E**.

To indicate that a function is being defined by means of an expression involving a dummy variable (t in the above example), the variable name is followed by an underscore on the left hand side of the equation defining the func-

tion.

It is possible to put two or more *Mathematica* commands in the same input cell; each command begins on a new line. The return key (as opposed to shift/return) produces a new line in the same input cell. (You can also extend a single command over several lines if necessary by using the return key, but this is unnecessary; if you just keep typing beyond the end of the line, *Mathematica* will automatically begin a new line for you.)

Now that *Mathematica* knows the definition of u, we can evaluate it:

```
In[26]:= u[5]
Out[26]= e^(5 a)
```

We can also compute its derivative:

```
In[27]:= D[u[t], t]
Out[27]= a e^(a t)
```

The syntax of this command should be clear: differentiate (**D**) the first expression (**u[t]**) with respect to the variable given as the second expression (**t**).

We now know enough to check whether the given function satisfies the differential equation. Note that *Mathematica* automatically simplifies the given expression (although not necessarily as much as possible, as we shall see):

```
In[28]:= D[u[t], t] - a * u[t]
Out[28]= 0
```

This result shows that u is a solution.

Is $v(t) = a\, t$ another solution? We check by defining the function and then substituting it into the left hand side of the differential equation:

```
In[29]:= ClearAll[v, a, t]
         v[t_] = a * t
Out[30]= a t
```

```
In[31]:= D[v[t], t] - a * v[t]
Out[31]= a - a^2 t
```

Since the result is not the zero function, we see that $v(t)$ is not a solution.

It is no more difficult to check whether a function of several variables is a solution to 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, we check by defining the function and then substituting it into the left-hand side of the differential equation:

```
In[32]:= ClearAll[w, x, y]
         w[x_, y_] = Sin[Pi * x] + Sin[Pi * y]
Out[33]= Sin[π x] + Sin[π y]

In[34]:= D[w[x, y], x, x] + D[w[x, y], y, y]
Out[34]= -π^2 Sin[π x] - π^2 Sin[π y]
```

The answer is no, w is not a solution to the differential equation. To compute higher derivatives using the **D** command, as this example shows, the independent variable is listed as many times as the order of the derivative. Mixed partial derivatives are then computed in the obvious way; for example, to compute

$$\frac{\partial^2 w}{\partial x\, \partial y},$$

we type

```
In[35]:= D[w[x, y], x, y]
Out[35]= 0
```

There is an alternate way to indicate repeated differentiation with respect to a given variable. The following command computes

$$\frac{\partial^5 w}{\partial w^5}:$$

```
In[36]:= D[w[x, y], {x, 5}]
Out[36]= π^5 Cos[π x]
```

More about computing derivatives and defining functions:

Above, we showed how to compute derivatives, using the **D** command, by first defining the function to be differentiated. For example, the following commands compute the derivative of x^2 :

```
In[37]:= ClearAll[f, x]
         f[x_] = x^2
Out[38]= x^2

In[39]:= D[f[x], x]
Out[39]= 2 x
```

However, **D** computes the derivative of an *expression*, not of a function. Therefore, if it is convenient, we can skip the step of defining f:

```
In[40]:= D[x^2, x]
Out[40]= 2 x
```

On the other hand, if we have defined f, and we wish to compute its derivative, we must apply **D** to $f(x)$, not to f ($f(x)$ is an expression, while f is a function). Here is the wrong way to compute $f'(x)$, followed by the right way:

```
In[41]:= D[f, x]
Out[41]= 0
```

```
In[42]:= D[f[x], x]
Out[42]= 2 x
```

Now is a good time to try out the **Help Browser**, if you have not already done so. Go to the **Help** menu, select **Wolfram Documentation**, and type **D** into the search bar.

We close this section by explaining a little more about how *Mathematica* handles functions. You should recall that a function is simply a mapping from one set (the domain) into a second set (the codomain). In calculus (and therefore when dealing with differential equations), the domain of a function of one variable tends to be the set of real numbers, or an interval of real numbers. However, there are many other possibilities, and *Mathematica* is flexible enough to handle them. For example, we can define a function $g:\{1,2,3\}\longrightarrow \boldsymbol{R}$ (the domain is the set $\{1,2,3\}$ of three integers) by

$$g(1)=1,\ g(2)=4,\ g(3)=9$$

This is expressed in *Mathematica* as follows:

```
In[43]:= ClearAll[g]
         g[1] = 1
         g[2] = 4
         g[3] = 9
Out[44]= 1
Out[45]= 4
Out[46]= 9
```

Mathematica now knows the value of g for any of the three inputs 1, 2, or 3. For example:

```
In[47]:= g[2]
Out[47]= 4
```

However, any other input leads to an indeterminate result:

```
In[48]:= g[5]
Out[48]= g[5]
```

```
In[49]:= g[x]
Out[49]= g[x]
```

We could define the value of **g[x]** as follows:

```
In[50]:= g[x] = x^2
Out[50]= x^2
```

However, notice the lack of the underscore character in the above definition (we entered "**g[x]=x^2**", not "**g[x_]=x^2**"). This means that g is defined for the input x, but not for an arbitrary input:

```
In[51]:= g[5]
Out[51]= g[5]
```

The ? operator displays the definition of *g*:

```
In[52]:= ?g

Global`g

g[1] = 1

g[2] = 4

g[3] = 9

g[x] = x^2
```

Hopefully, the meaning of the underscore character is now clear: it is necessary to tell *Mathematica* when an input is regarded as a dummy variable. For example, recall the definition of *f*:

```
In[53]:= ?f

Global`f

f[x_] = x^2
```

The function f was defined for an arbitrary input *x*:

```
In[54]:= f[5]
Out[54]= 25
```

A common mistake is to neglect the underscore when you mean to define a function for an arbitrary input; be forewarned!

The ability to give several definitions for a function can be useful. Consider the function

```
In[55]:= ClearAll[h, x]
         h[x_] = Sin[x]/x
Out[56]= Sin[x]
         ------
           x
```

This function is formally undefined at x=0:

```
In[57]:= h[0]
         Power::infy : Infinite expression 1/0 encountered. >>
         Infinity::indet : Indeterminate expression 0 ComplexInfinity encountered. >>
Out[57]= Indeterminate
```

However, as you might remember from calculus, the limit of $h(x)$ as x approaches 0 exists and equals 1. Therefore, we might like to add a second definition:

```
In[58]:= h[0] = 1
Out[58]= 1
```

Now the function is completely defined:

```
In[59]:= h[0]
Out[59]= 1
```

```
In[60]:= h[x]
```

Out[60]= $\frac{\text{Sin}[x]}{x}$

```
In[61]:= ?h
```

Global`h

h[0] = 1

h[x_] = $\frac{\text{Sin}[x]}{x}$

Mathematica checks for any "special" definitions (like **h[0]=1**) before applying a general definition involving a dummy variable.

A note about using ClearAll

You may wonder why we recommend that you clear a symbol before assigning it a value. For example, if we assign a value to *u*, will not that value replace any previous value? The answer is "not always." Consider the following example:

```
In[62]:= ClearAll[u, x]
```

```
In[63]:= u = 4
Out[63]= 4
```

Now the symbol *u* has a value, and *Mathematica* will not allow another value to be assigned:

```
In[64]:= u[x_] = x^2
Set::write : Tag Integer in 4[x_] is Protected. >>
```

Out[64]= x^2

```
In[65]:= ?u
```

Global`u

u = 4

However, consider the following:

```
In[66]:= ClearAll[u, x]
```

In[67]:=
```
u[x_] = x^2
```
Out[67]= x^2

In[68]:=
```
u = 4
```
Out[68]= 4

In[69]:=
```
? u
```

Global`u

u = 4

u[x_] = x^2

Now the symbol *u* has two meanings, which is probably not what is intended.

In[70]:=
```
u[3]
```
Out[70]= 4[3]

The moral of the story is that it is a good idea to use **ClearAll** consistently, as we do in the rest of this tutorial. Undoubtedly, many of the instances of its use could be omitted without any harmful effects, but, in certain cases, omitting it leads to errors that are not easy to find.

A corollary of these remarks is that if you encounter unexpected behavior when using *Mathematica*, it is a good idea to verify that all symbols have been properly cleared and then assigned the desired value.

Chapter 2: Models in one dimension

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

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

In[71]:=
```
ClearAll[x]
```

In[72]:=
```
Integrate[Sin[x], x]
```
Out[72]= -Cos[x]

As this example shows, *Mathematica* does *not* add the "constant of integration." It simply returns one antiderivative (when possible). If the integrand is too complicated, the integral is returned unevaluated:

```
In[73]:= Integrate[E^Cos[x], x]
```

Out[73]= $\int e^{\mathrm{Cos}[x]}\,dx$

Computing a definite integral such as

$$\int_0^1 \sin(x)\,dx$$

requires that we specify the variable of integration together with the lower and upper limits of integration:

```
In[74]:= Integrate[Sin[x], {x, 0, 1}]
Out[74]= 1 - Cos[1]
```

Mathematica also has a command for computing a definite integral numerically (that is, approximately):

```
In[75]:= NIntegrate[E^Cos[x], {x, 0, 1}]
Out[75]= 2.34157
```

As this example shows, **NIntegrate** is useful for integrating functions for which no elementary antiderivative can be found.

As an example, let us use the commands for integration and differentiation to test Theorem 2.1 from the text. The theorem states that (under certain conditions)

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

Here is a specific instance of the theorem:

```
In[76]:= ClearAll[F, x, y]
         F[x_, y_] = x y^3 + x^2 y
```

Out[77]= $x^2\,y + x\,y^3$

```
In[78]:= D[Integrate[F[x, y], {y, c, d}], x]
```

Out[78]= $-\frac{c^4}{4} + \frac{d^4}{4} - c^2\,x + d^2\,x$

```
In[79]:= Integrate[D[F[x, y], x], {y, c, d}]
```

Out[79]= $-\frac{c^4}{4} + \frac{d^4}{4} - c^2\,x + d^2\,x$

The two results are equal, as expected.

Solving simple BVPs by integration

Consider the following BVP:

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

The solution can be found by two integrations (cf. Example 2.2 in the text). As mentioned above, *Mathematica* will not add a constant of integration, so we do it explicitly:

In[80]:=
```
ClearAll[u, x, C1, C2]
```

In[81]:=
```
Integrate[-(1 + x), x] + C1
```

Out[81]= $C1 - x - \frac{x^2}{2}$

In[82]:=
```
Integrate[%, x] + C2
```

Out[82]= $C2 + C1\,x - \frac{x^2}{2} - \frac{x^3}{6}$

The above result is our candidate for u:

In[83]:=
```
u[x_] = %
```

Out[83]= $C2 + C1\,x - \frac{x^2}{2} - \frac{x^3}{6}$

We now solve for the constants:

In[84]:=
```
Solve[{u[0] == 0, u[1] == 0}, {C1, C2}]
```

Out[84]= $\left\{\left\{C1 \to \frac{2}{3}, C2 \to 0\right\}\right\}$

As this example shows, *Mathematica* can solve algebraic equations. The **Solve** command requires two inputs, a list of equations and a list of variables for which to solve. Lists are always indicated by curly braces. When specifying an equation, the symbol for equality is the double equals sign == (a single equals sign means assignment, as we have already seen many times). *Mathematica* returns any solutions found as a list of *replacment rules,* from which we can read the solution(s). It is also possible to extract the solutions automatically from the replacement rules, but this requires a fair amount of explanation. It will be easier to understand how to manipulate replacement rules after we explain more about *Mathematica* in Chapter 3, so, for now, we just read off the solution and assign it to the constants **C1** and **C2**:

In[85]:=
```
C1 = 2 / 3
```

Out[85]= $\frac{2}{3}$

In[86]:=
```
C2 = 0
```

Out[86]= 0

We now have the solution to the BVP:

In[87]:=
```
u[x]
```

Out[87]= $\frac{2\,x}{3} - \frac{x^2}{2} - \frac{x^3}{6}$

Let us check that this really is the solution:

In[88]:=
```
-D[u[x], {x, 2}] - (1 + x)
```

Out[88]= 0

In[89]:=
```
u[0]
```
Out[89]= 0

In[90]:=
```
u[1]
```
Out[90]= 0

As another example, we solve a 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

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

Next we need to integrate the expression **C1/(1+x/2)**:

In[91]:=
```
Integrate[C1 / (1 + x / 2), x] + C2
```
Out[91]= $\frac{4}{3}\text{Log}[6+3x]$

Whoops! Here we made the mistake that we warned you about earlier. The symbols **C1** and **C2** already had values from the preceding example, so we need to clear them before using these symbols as undetermined constants:

In[92]:=
```
ClearAll[C1, C2]
```

In[93]:=
```
Integrate[C1 / (1 + x / 2), x] + C2
```
Out[93]= C2 + 2 C1 Log[2 + x]

From now on, we will be consistent about using the **ClearAll** command.

In[94]:=
```
ClearAll[u, x]
```

In[95]:=
```
u[x_] = %%
```
Out[95]= C2 + 2 C1 Log[2 + x]

(The symbol %% represents the last output but one.) Now we can solve for the constants of integration:

In[96]:=
```
Solve[{u[0] == 20, u[1] == 25}, {C1, C2}]
```
Out[96]= $\left\{\left\{C1\to-\frac{5}{2\,(\text{Log}[2]-\text{Log}[3])},\ C2\to\frac{5\,(5\,\text{Log}[2]-4\,\text{Log}[3])}{\text{Log}[2]-\text{Log}[3]}\right\}\right\}$

In[97]:=
```
C1 = -5 / (2 (Log[2] - Log[3]))
```
Out[97]= $-\frac{5}{2\,(\text{Log}[2]-\text{Log}[3])}$

In[98]:=
```
C2 = 5 (5 Log[2] - 4 Log[3]) / (Log[2] - Log[3])
```
Out[98]= $\frac{5\,(5\,\text{Log}[2]-4\,\text{Log}[3])}{\text{Log}[2]-\text{Log}[3]}$

Here is the final solution:

In[99]:=
```
u[x]
```
Out[99]= $\frac{5\,(5\,\text{Log}[2]-4\,\text{Log}[3])}{\text{Log}[2]-\text{Log}[3]}-\frac{5\,\text{Log}[2+x]}{\text{Log}[2]-\text{Log}[3]}$

Let's check it:

In[100]:=
```
-D[(1 + x / 2) D[u[x], x], x]
```
Out[100]= $-\frac{5\left(1+\frac{x}{2}\right)}{(2+x)^2\,(\text{Log}[2]-\text{Log}[3])}+\frac{5}{2\,(2+x)\,(\text{Log}[2]-\text{Log}[3])}$

The result is not zero; does this mean that we (or *Mathematica*!) made a mistake? The answer is no; before jumping to this conclusion, we should ask *Mathematica* to simplify the result. When evaluating an expression, *Mathematica* will apply some simplification transformations automatically, but it is necessary to give a command if we wish *Mathematica* to perform all of its known algebraic transformations:

In[101]:=
```
Simplify[%]
```
Out[101]= 0

We see that u is indeed a solution to the differential equation.

In[102]:=
```
u[0]
```
Out[102]= $-\frac{5\,\text{Log}[2]}{\text{Log}[2]-\text{Log}[3]}+\frac{5\,(5\,\text{Log}[2]-4\,\text{Log}[3])}{\text{Log}[2]-\text{Log}[3]}$

In[103]:=
```
Simplify[%]
```
Out[103]= 20

In[104]:=
```
u[1]
```
Out[104]= $\frac{5\,(5\,\text{Log}[2]-4\,\text{Log}[3])}{\text{Log}[2]-\text{Log}[3]}-\frac{5\,\text{Log}[3]}{\text{Log}[2]-\text{Log}[3]}$

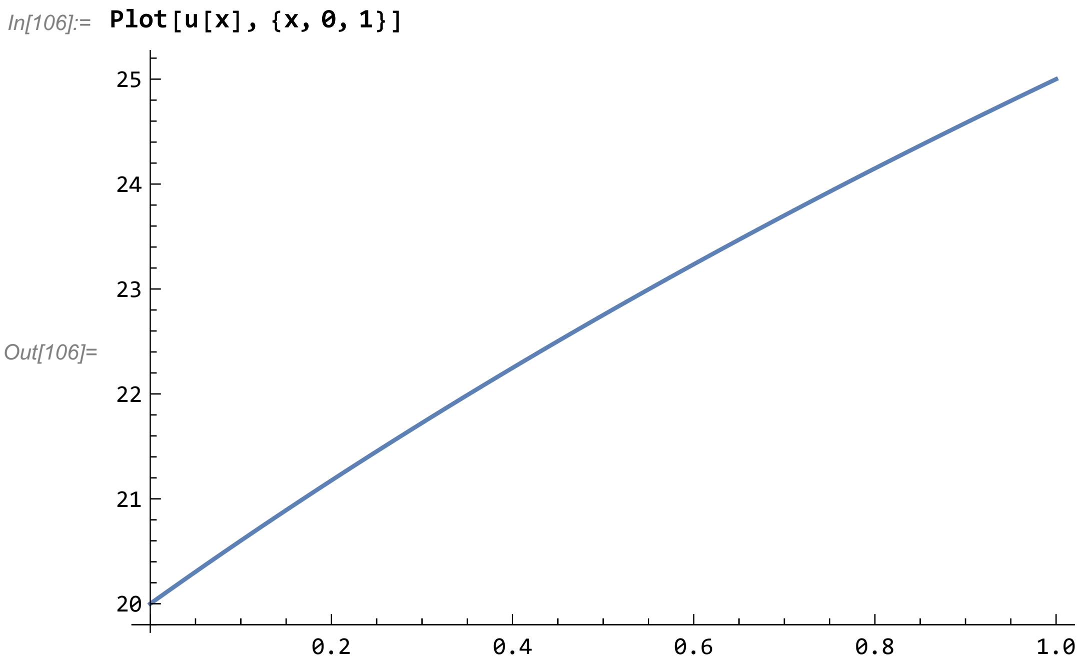
In[105]:=
```
Simplify[%]
```
Out[105]= 25

The boundary conditions are also satisfied.

Simple plots

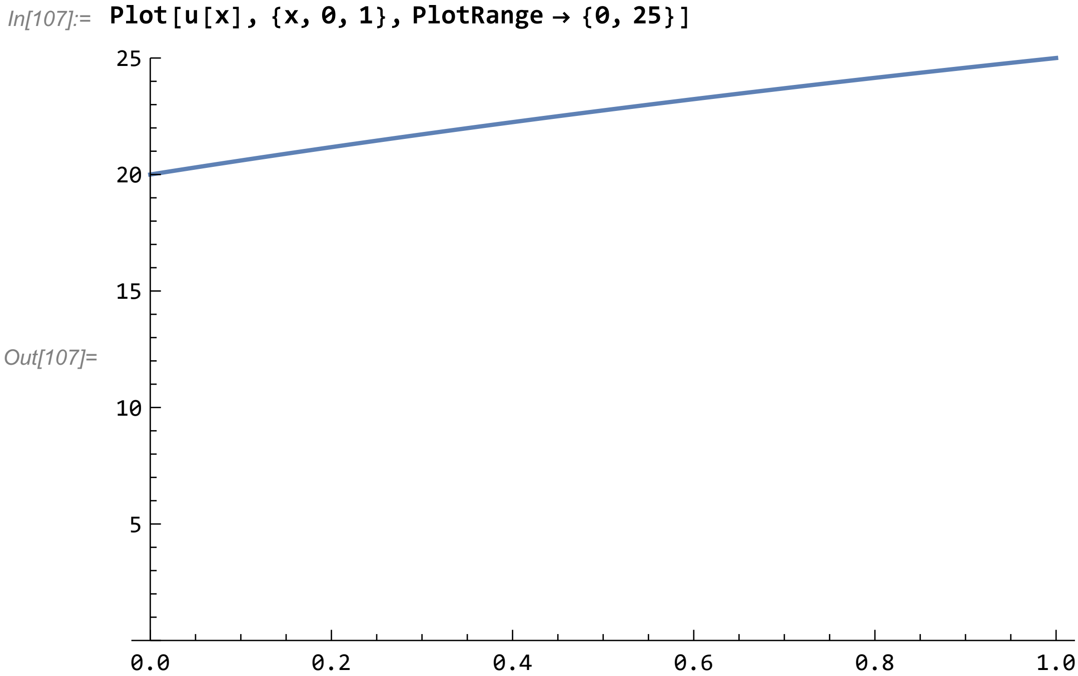
One of the most useful features of *Mathematica* is its ability to draw many kinds of graphs. Here we show how to produce the graph of a function of one variable. The command is called **Plot**, and we simply give it the expression to graph, the independent variable, and the interval. Here is a graph of the previous solution:

In[106]:= `Plot[u[x], {x, 0, 1}]`

Out[106]=


The output of the **Plot** command can be controlled by various options. For example, we may not like the fact that, in the above graph, the vertical range is the interval [20,25]. We can change this using the **PlotRange** option:

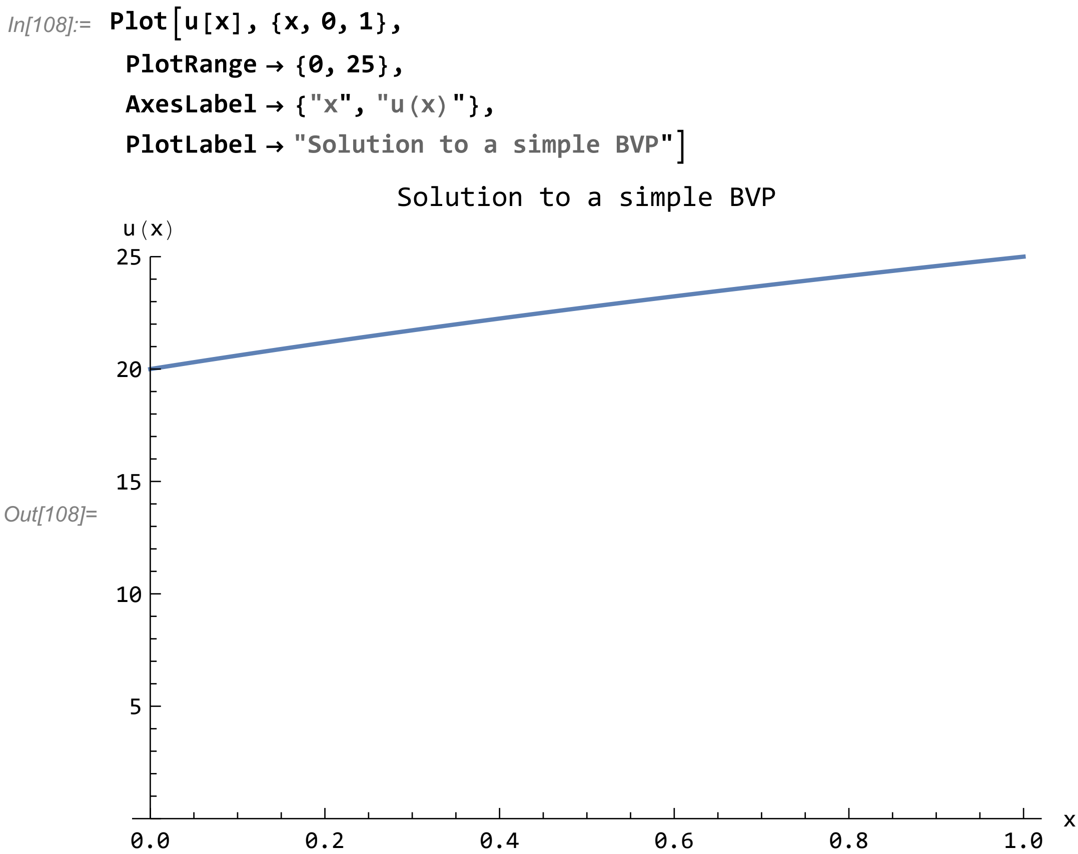
In[107]:= `Plot[u[x], {x, 0, 1}, PlotRange → {0, 25}]`

Out[107]=


We can label the graph using the **AxesLabel** and **PlotLabel** options (for readability, you might wish to put each option on a new line, as in the following example):

In[108]:=
```
Plot[u[x], {x, 0, 1},
 PlotRange → {0, 25},
 AxesLabel → {"x", "u(x)"},
 PlotLabel → "Solution to a simple BVP"]
```

Out[108]=



More information about options to the Plot command can be found in the "Options" section of the documention of the **Plot** command. One more option will be explained here: how to graph multiple functions in a single plot. For example, suppose we wish to compare the solution of the previous BVP to the solution of the related BVP with a constant coefficient in the differential equation:

$$-\frac{d^2 v}{dx^2} = 0, \quad 0 < x < 1,$$
$$v(0) = 20,$$
$$v(1) = 25,$$

which is

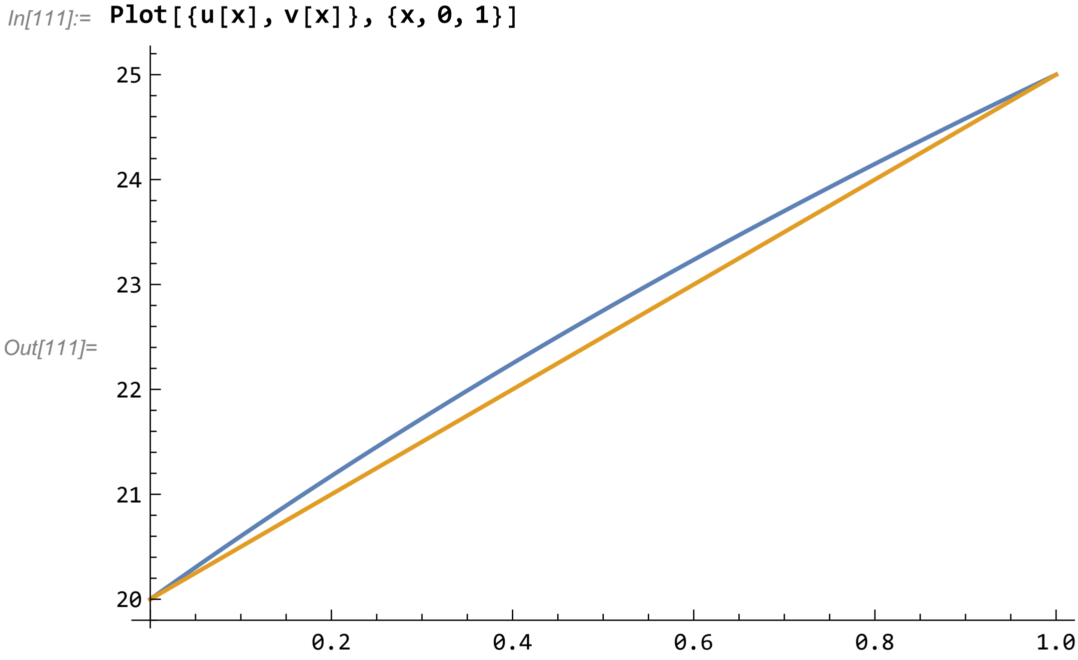
In[109]:=
```
ClearAll[v, x]
v[x_] = 20 + 5 x
```

Out[110]= 20 + 5 x

We simply give the **Plot** command a list of expressions to plot:

In[111]:=
```
Plot[{u[x], v[x]}, {x, 0, 1}]
```

Out[111]=



Chapter 3: Essential linear algebra

Section 3.1: Linear systems as linear operator equations

Mathematica will manipulate matrices and vectors, and perform the usual computations of linear algebra. Both symbolic and numeric (that is, floating point) computation are supported.

A vector is entered as a list of components:

In[112]:=
```
ClearAll[x]
x = {1, 2, 3}
```

Out[113]= {1, 2, 3}

By default, a vector is displayed as a list of numbers; however, the more common notation of a column matrix can be requested using the MatrixForm function:

In[114]:=
```
MatrixForm[x]
```

Out[114]//MatrixForm=

$$\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

An alternate way to apply the MatrixForm function (or any other function) is postfix syntax:

```
In[115]:= x // MatrixForm
Out[115]//MatrixForm=
```

$$\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

A matrix is entered as a list of row vectors:

```
In[116]:= ClearAll[A]
          A = {{1, 2, -1}, {4, 0, 1}, {-7, -2, 3}}
Out[117]= {{1, 2, -1}, {4, 0, 1}, {-7, -2, 3}}
```

```
In[118]:= A // MatrixForm
Out[118]//MatrixForm=
```

$$\begin{pmatrix} 1 & 2 & -1 \\ 4 & 0 & 1 \\ -7 & -2 & 3 \end{pmatrix}$$

The transpose of a matrix is available:

```
In[119]:= Transpose[A] // MatrixForm
Out[119]//MatrixForm=
```

$$\begin{pmatrix} 1 & 4 & -7 \\ 2 & 0 & -2 \\ -1 & 1 & 3 \end{pmatrix}$$

Multiplication of a matrix times a vector or a matrix times a matrix is indicated by the Dot (.) operator:

```
In[120]:= A.x
Out[120]= {2, 7, -2}
```

```
In[121]:= A.A
Out[121]= {{16, 4, -2}, {-3, 6, -1}, {-36, -20, 14}}
```

```
In[122]:= ClearAll[B]
```

```
In[123]:= B = {{2, 1}, {-1, 0}, {3, -2}, {1, 4}}
Out[123]= {{2, 1}, {-1, 0}, {3, -2}, {1, 4}}
```

```
In[124]:= MatrixForm[B]
Out[124]//MatrixForm=
```

$$\begin{pmatrix} 2 & 1 \\ -1 & 0 \\ 3 & -2 \\ 1 & 4 \end{pmatrix}$$

```
In[125]:= A.B
```

Dot::dotsh : Tensors {{1, 2, −1}, {4, 0, 1}, {−7, −2, 3}}
and {{2, 1}, {−1, 0}, {3, −2}, {1, 4}} have incompatible shapes. ≫

```
Out[125]= {{1, 2, -1}, {4, 0, 1}, {-7, -2, 3}}.{{2, 1}, {-1, 0}, {3, -2}, {1, 4}}
```

As the last computation shows, *Mathematica* is aware of the rules of matrix algebra. The computation failed because the number of columns in A is not the same as the number of rows in B.

It is frequently useful to refer to the components of a vector, the entries of a matrix, or the rows of a matrix. For these purposes, *Mathematica* has an indexing operator. Since parentheses are reserved for algebraic grouping, square brackets for function evaluation, and curly brackets for lists, *Mathematica* uses double square brackets for the indexing operator. Thus the third component of x is obtained as follows:

```
In[126]:= x[[3]]
Out[126]= 3
```

The (2,3) entry of the matrix A is

```
In[127]:= A[[2, 3]]
Out[127]= 1
```

Since, in *Mathematica*, a matrix is a list of rows, we can extract a row using a single index. Here is the second row of A:

```
In[128]:= A[[2]]
Out[128]= {4, 0, 1}
```

A column of a matrix can be extracted using the **All** keyword:

```
In[129]:= A[[All, 2]]
Out[129]= {2, 0, -2}
```

You should understand the above notation as denoted all of the entries in the second column (all rows and column 2). The following is an alternative for extracting the second row:

```
In[130]:= A[[2, All]]
Out[130]= {4, 0, 1}
```

■ Section 3.2: Existence and Uniqueness of solutions to Ax=b

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

```
In[131]:= ClearAll[B]
          B = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}
Out[132]= {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}
```

In[133]:=
```
MatrixForm[B]
```

Out[133]//MatrixForm=

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

In[134]:=
```
NullSpace[B]
```

Out[134]= {{1, -2, 1}}

The **NullSpace** command returns a list of vectors, which forms a basis for the nullspace of the given matrix. In the above example, the nullspace is one-dimensional, so the result is a list containing a single vector. To refer to that vector, we extract it from the list using the index operator:

In[135]:=
```
ClearAll[y]
y = %%[[1]]
```

Out[136]= {1, -2, 1}

We can test whether the result is correct:

In[137]:=
```
B.y // MatrixForm
```

Out[137]//MatrixForm=

$$\begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

If a matrix is nonsingular, then its null space is trivial (that is, it contains only the zero vector). Since the trivial subspace does not have a basis, the **NullSpace** command returns an empty list:

In[138]:=
```
NullSpace[A]
```

Out[138]= {}

(Recall that *A* is a nonsingular matrix that I defined above.)

Here is another example (see Example 3.16 in the text):

In[139]:=
```
ClearAll[A1]
A1 = {{1, 3, -1, 2}, {0, 1, 4, 2}, {2, 7, 2, 6}, {1, 4, 3, 4}}
```

Out[140]= {{1, 3, -1, 2}, {0, 1, 4, 2}, {2, 7, 2, 6}, {1, 4, 3, 4}}

In[141]:=
```
MatrixForm[A1]
```

Out[141]//MatrixForm=

$$\begin{pmatrix} 1 & 3 & -1 & 2 \\ 0 & 1 & 4 & 2 \\ 2 & 7 & 2 & 6 \\ 1 & 4 & 3 & 4 \end{pmatrix}$$

In[142]:=
```
NullSpace[A1]
```

Out[142]= {{4, -2, 0, 1}, {13, -4, 1, 0}}

The null space of this matrix has dimension two.

Mathematica can compute the inverse of a matrix:

In[143]:=
```
ClearAll[Ainv]
```

In[144]:=
```
Ainv = Inverse[A]
```

Out[144]= $\left\{\left\{-\frac{1}{14}, \frac{1}{7}, -\frac{1}{14}\right\}, \left\{\frac{19}{28}, \frac{1}{7}, \frac{5}{28}\right\}, \left\{\frac{2}{7}, \frac{3}{7}, \frac{2}{7}\right\}\right\}$

In[145]:=
```
MatrixForm[Ainv]
```

Out[145]//MatrixForm=

$$\begin{pmatrix} -\frac{1}{14} & \frac{1}{7} & -\frac{1}{14} \\ \frac{19}{28} & \frac{1}{7} & \frac{5}{28} \\ \frac{2}{7} & \frac{3}{7} & \frac{2}{7} \end{pmatrix}$$

Check:

In[146]:=
```
Ainv.A // MatrixForm
```

Out[146]//MatrixForm=

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

The command **IdentityMatrix** creates an identity matrix of a given size. Here is the 4 by 4 identity:

In[147]:=
```
ClearAll[I4]
I4 = IdentityMatrix[4]
```

Out[148]= {{1, 0, 0, 0}, {0, 1, 0, 0}, {0, 0, 1, 0}, {0, 0, 0, 1}}

In[149]:=
```
MatrixForm[I4]
```

Out[149]//MatrixForm=

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

(The symbol **I** is reserved; it means $\sqrt{-1}$. Therefore, you cannot call your identity matrix **I**.)

Using the inverse matrix, you can solve a linear system:

In[150]:=
```
ClearAll[x, b]
```

In[151]:=
```
b = {1, 1, 1}
```

Out[151]= {1, 1, 1}

In[152]:=
```
x = Ainv.b
```

Out[152]= {0, 1, 1}

However, it is more efficient to solve the system directly using the **LinearSolve** command:

In[153]:=
```
LinearSolve[A, b]
```
Out[153]=
```
{0, 1, 1}
```

(LinearSolve does not compute A^{-1}.)

Section 3.3: Basis and dimension

In this section, we demonstrate some more capabilities of *Mathematica* 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:

In[154]:=
```
ClearAll[v1, v2, v3]
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]}
```
Out[155]= $\left\{\frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}\right\}$

Out[156]= $\left\{\frac{1}{\sqrt{2}}, 0, -\frac{1}{\sqrt{2}}\right\}$

Out[157]= $\left\{\frac{1}{\sqrt{6}}, -\sqrt{\frac{2}{3}}, \frac{1}{\sqrt{6}}\right\}$

The following calculations verify that these vectors are orthogonal:

In[158]:=
```
v1.v2
```
Out[158]=
```
0
```

In[159]:=
```
v1.v3
```
Out[159]=
```
0
```

In[160]:=
```
v2.v3
```
Out[160]=
```
0
```

Example 3.27

In this example, we verify that the following three quadratic polynomials form a basis for P_2:

```
In[161]:= ClearAll[p1, p2, p3, x]
          p1[x_] = 1
          p2[x_] = x - 1 / 2
          p3[x_] = x^2 - x + 1 / 6
```

Out[162]= 1

Out[163]= $-\frac{1}{2}+x$

Out[164]= $\frac{1}{6}-x+x^2$

Suppose that $q(x)$ is an arbitrary quadratic polynomial.

```
In[165]:= ClearAll[a, b, c, q, x]
          q[x_] = a x^2 + b x + c
```

Out[166]= $c+b\,x+a\,x^2$

The following calculations write q in terms of **p1**, **p2**, and **p3**:

```
In[167]:= ClearAll[c1, c2, c3]
```

```
In[168]:= q[x] - (c1 p1[x] + c2 p2[x] + c3 p3[x])
```

Out[168]= $c-c1-c2\left(-\frac{1}{2}+x\right)+b\,x+a\,x^2-c3\left(\frac{1}{6}-x+x^2\right)$

We need to set each coefficient (that is, the constant term, the coefficient of x, and the coefficient of x^2) to zero and solve for **c1**, **c2**, and **c3**. To see the coefficients, we use the **Collect** command:

```
In[169]:= Collect[%, x]
```

Out[169]= $c-c1+\frac{c2}{2}-\frac{c3}{6}+(b-c2+c3)\,x+(a-c3)\,x^2$

Now we can extract the coefficients of the above polynomial, set them equal to zero, and solve for **c1**, **c2**, **c3**:

```
In[170]:= Solve[{Coefficient[%, x, 0] == 0,
            Coefficient[%, x, 1] == 0, Coefficient[%, x, 2] == 0}, {c1, c2, c3}]
```

Out[170]= $\left\{\left\{c1\to\frac{1}{6}(2\,a+3\,b+6\,c),\ c2\to a+b,\ c3\to a\right\}\right\}$

There is a unique solution **c1**, **c2**, **c3** for any a, b, c, that is, for any quadratic polynomial $q(x)$. This shows that {**p1**,**p2**,**p3**} is a basis for P_2.

Digression: Manipulating replacement rules

Here is a good place to explain how to manipulate the transformation rules that commands like **Solve** return. First, we must explain the *replacement operator* "/.". Suppose we have an expression like x^2+3x+1, and we want to know its value for $x=3$. One way to do this is to assign the value 3 to x, and then type in the expression:

```
In[171]:= ClearAll[x]
          x = 3
Out[172]= 3

In[173]:= x^2 + 3 x + 1
Out[173]= 19
```

The disadvantage of this approach is that x now has the value 3 and therefore x will be replaced by 3 whenever it is used (unless its value is cleared).

As an alternative, we can type the expression followed by the transformation rule $x \to 3$. In between goes the replacement operator "/.", which tells *Mathematica* to apply the following transformation rule. (In the following example, we first clear x to undo the above assignment .)

```
In[174]:= ClearAll[x]

In[175]:= x^2 + 3 x + 1 /. x → 3
Out[175]= 19
```

The above transformation rule is applied only to the foregoing expression; it has no permanent effect. In particular, x has not been assigned the value of 3:

```
In[176]:= x
Out[176]= x
```

We can also give a list of transformation rules:

```
In[177]:= ClearAll[x, y]

In[178]:= x^2 y /. {x → 2, y → -4}
Out[178]= -16
```

Of course, a list can contain only one entry:

```
In[179]:= x^2 /. {x → -2}
Out[179]= 4
```

This last point is important in understanding how to manipulate the results returned by **Solve**. Here is the behavior of **Solve**: it returns a list of solutions, each of which is a list of transformation rules for the unknown variables. Thus **Solve** always returns a list of lists!

The following examples will hopefully make this clear. First, the simplest example: a single unknown, with a unique solution:

```
In[180]:= ClearAll[x]

In[181]:= Solve[2 x + 1 == 0, x]
```

Out[181]= $\left\{\left\{x \to -\frac{1}{2}\right\}\right\}$

The result is a list of solutions; since the equation has a unique solution, this list contains a single entry:

In[182]:=
```
%[[1]]
```
Out[182]= $\left\{x \to -\frac{1}{2}\right\}$

This entry is a list of transformation rules, and since the equation contains a single unknown, this list also contains a single entry. To extract the value of the unknown (without retyping it), I can use the replacement operator:

In[183]:=
```
x /. %
```
Out[183]= $-\frac{1}{2}$

Usually the previous two steps would be combined into one:

In[184]:=
```
Solve[2 x + 1 == 0, x]
```
Out[184]= $\left\{\left\{x \to -\frac{1}{2}\right\}\right\}$

In[185]:=
```
x /. %[[1]]
```
Out[185]= $-\frac{1}{2}$

If we do not extract the first entry in the list, we get a list of solutions (in this case, a list with only one entry):

In[186]:=
```
x /. %%
```
Out[186]= $\left\{-\frac{1}{2}\right\}$

Next, here is an example with a single unknown, but two solutions:

In[187]:=
```
Solve[x^2 - 3 x + 3 == 0, x]
```
Out[187]= $\left\{\left\{x \to \frac{1}{2}\left(3 - i\sqrt{3}\right)\right\}, \left\{x \to \frac{1}{2}\left(3 + i\sqrt{3}\right)\right\}\right\}$

Let us extract the second solution:

In[188]:=
```
x /. %[[2]]
```
Out[188]= $\frac{1}{2}\left(3 + i\sqrt{3}\right)$

Here are the list of all solutions :

In[189]:=
```
x /. %%
```
Out[189]= $\left\{\frac{1}{2}\left(3 - i\sqrt{3}\right), \frac{1}{2}\left(3 + i\sqrt{3}\right)\right\}$

Finally, here is an example with two variables and two solutions. Notice that it is convenient to assign a name to the output of **Solve**, for easy reference later.

```
In[190]:= ClearAll[x, y, sols]
          sols = Solve[{x^2 + y^2 == 1, y == x}, {x, y}]
```

Out[191]= $\left\{\left\{x \to -\frac{1}{\sqrt{2}}, y \to -\frac{1}{\sqrt{2}}\right\}, \left\{x \to \frac{1}{\sqrt{2}}, y \to \frac{1}{\sqrt{2}}\right\}\right\}$

Here is the first solution:

```
In[192]:= x /. sols[[1]]
```

Out[192]= $-\frac{1}{\sqrt{2}}$

```
In[193]:= y /. sols[[1]]
```

Out[193]= $-\frac{1}{\sqrt{2}}$

When the solution is complicated, it is important to be able to extract it from the transformation rule without retyping it, which is tedious and difficult to do correctly.

Example

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

```
In[194]:= ClearAll[u1, u2, u3]
          u1 = {1, 0, 2}
          u2 = {0, 1, 1}
          u3 = {1, 2, -1}
Out[195]= {1, 0, 2}
Out[196]= {0, 1, 1}
Out[197]= {1, 2, -1}
```

We will verify that {**u1**,**u2**,**u3**} is a basis for R^3 and express the following vector x in terms of the basis.

```
In[198]:= ClearAll[x]
          x = {8, 2, -4}
Out[199]= {8, 2, -4}
```

As discussed in the text, {**u1**,**u2**,**u3**} is linearly independent if and only if the matrix A whose columns are **u1**, **u2**, **u3** is nonsingular.

```
In[200]:= ClearAll[A]
          A = Transpose[{u1, u2, u3}]
Out[201]= {{1, 0, 1}, {0, 1, 2}, {2, 1, -1}}
```

```
In[202]:= MatrixForm[A]
Out[202]//MatrixForm=
```

$$\begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 2 \\ 2 & 1 & -1 \end{pmatrix}$$

(Note that, since *Mathematica* represents a matrix as a list of row vectors, the desired matrix is the transpose of the matrix **{u1,u2,u3}.)** We can verify that A is nonsingular by computing its determinant:

```
In[203]:= Det[A]
Out[203]= -5
```

Since det(A) is nonzero, A is nonsingular. In general, computing the determinant is not a good way to check that the matrix is nonsingular. If the matrix is large and the entries are floating point numbers, the determinant is likely to give a misleading answer due to round-off error. A better command is **MatrixRank**. Here is an example:

```
In[204]:= B = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}
Out[204]= {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}

In[205]:= MatrixRank[B]
Out[205]= 2
```

Since the rank of the 3 by 3 matrix is only 2, the matrix is singular.

Back to the example: We can express x as a linear combination of **u1**, **u2**, **u3** by solving $Ac=x$:

```
In[206]:= ClearAll[c]
          c = LinearSolve[A, x]
```

Out[207]= $\left\{\frac{18}{5}, -\frac{34}{5}, \frac{22}{5}\right\}$

Check:

```
In[208]:= c[[1]] u1 + c[[2]] u2 + c[[3]] u3
Out[208]= {8, 2, -4}

In[209]:= x
Out[209]= {8, 2, -4}
```

■ Section 3.4: Orthogonal bases and projection

We have already explained how to compute dot products and verify orthogonality. For example:

In[210]:=
```
ClearAll[v1, v2, v3]
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]}
```

Out[211]= $\left\{\frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}\right\}$

Out[212]= $\left\{\frac{1}{\sqrt{2}}, 0, -\frac{1}{\sqrt{2}}\right\}$

Out[213]= $\left\{\frac{1}{\sqrt{6}}, -\sqrt{\frac{2}{3}}, \frac{1}{\sqrt{6}}\right\}$

These vectors are orthogonal:

In[214]:=
```
v1.v2
v1.v3
v2.v3
```

Out[214]= 0

Out[215]= 0

Out[216]= 0

They are also normalized:

In[217]:=
```
v1.v1
v2.v2
v3.v3
```

Out[217]= 1

Out[218]= 1

Out[219]= 1

Therefore, we can easily express any vector as a linear combination of these three vectors, which form an orthonormal basis:

In[220]:=
```
ClearAll[a, b, c, x]
x = {a, b, c}
```

Out[221]= {a, b, c}

In[222]:=
```
(x.v1) v1 + (x.v2) v2 + (x.v3) v3
```

Out[222]=
$$\left\{\frac{\frac{a}{\sqrt{2}}-\frac{c}{\sqrt{2}}}{\sqrt{2}}+\frac{\frac{a}{\sqrt{3}}+\frac{b}{\sqrt{3}}+\frac{c}{\sqrt{3}}}{\sqrt{3}}+\frac{\frac{a}{\sqrt{6}}-\sqrt{\frac{2}{3}}\,b+\frac{c}{\sqrt{6}}}{\sqrt{6}},\ \frac{\frac{a}{\sqrt{3}}+\frac{b}{\sqrt{3}}+\frac{c}{\sqrt{3}}}{\sqrt{3}}-\sqrt{\frac{2}{3}}\left(\frac{a}{\sqrt{6}}-\sqrt{\frac{2}{3}}\,b+\frac{c}{\sqrt{6}}\right),\ -\frac{\frac{a}{\sqrt{2}}-\frac{c}{\sqrt{2}}}{\sqrt{2}}+\frac{\frac{a}{\sqrt{3}}+\frac{b}{\sqrt{3}}+\frac{c}{\sqrt{3}}}{\sqrt{3}}+\frac{\frac{a}{\sqrt{6}}-\sqrt{\frac{2}{3}}\,b+\frac{c}{\sqrt{6}}}{\sqrt{6}}\right\}$$

In[223]:=
```
Simplify[%]
```

Out[223]= {a, b, c}

Working with the L^2inner product

The L^2inner product is not provided in *Mathematica*, as is the dot product, so it is convenient to define a function implementing it. Suppose we are working on the interval [0,1]. We call the desired function **l2ip**:

In[224]:=
```
ClearAll[l2ip]
l2ip[f_, g_] := Integrate[f[x] × g[x], {x, 0, 1}]
```

This is the first time we have used ":=" for the assignment operator. The ":=" operator is the *delayed-evaluation* assignment operator; it tells *Mathematica* to save the formula on the right-hand side until the function is called with specific inputs. On the other hand, the "=" operator is the *immediate-evaluation* assignment operator. Here is an example:

In[226]:=
```
ClearAll[f, g, x]
```

In[227]:=
```
f[x_] = x x
```

Out[227]= x^2

Notice how *Mathematica* evaluated the right-hand side (x times x) and decided to simplify it as x^2. With the delayed-evaluation operator, this does not happen:

In[228]:=
```
g[x_] := x x
```

In[229]:=
```
? g
```

Global`g

```
g[x_] := x x
```

In[230]:=
```
? f
```

Global`f

```
f[x_] = x^2
```

For many purposes, immediate evaluation is fine. However, when defining functions in terms of integrals, sums, and so forth, delayed evaluation may be essential. In the case of the function **l2ip** defined above, we do not want *Mathematica* to try to integrate $f(x)g(x)$ until particular functions f and g are specified.

For convenience, we also define a function implementing the L^2 norm:

In[231]:=
```
ClearAll[l2norm]
l2norm[f_] := Sqrt[l2ip[f, f]]
```

Example 3.35

Now consider the following two functions:

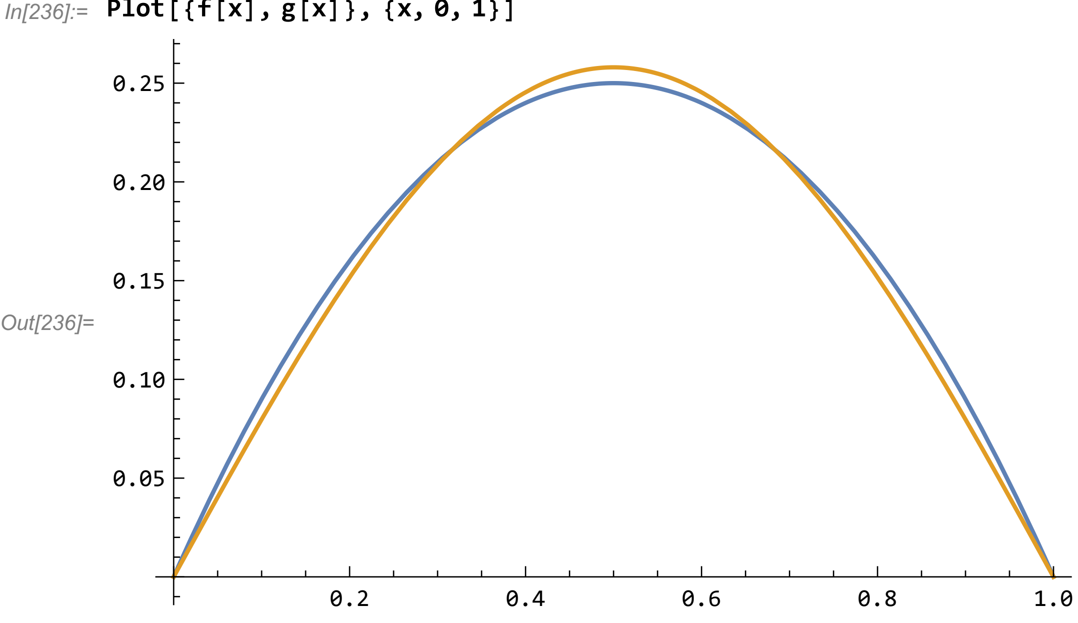
In[233]:=
```
ClearAll[f, g, x]
f[x_] = x (1 - x)
g[x_] = 8 / Pi^3 Sin[Pi x]
```

Out[234]= $(1 - x)\, x$

Out[235]= $\frac{8 \operatorname{Sin}[\pi x]}{\pi^3}$

The following graph shows that the two functions are quite similar:

In[236]:=
```
Plot[{f[x], g[x]}, {x, 0, 1}]
```

Out[236]=


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:

```
In[237]:= ClearAll[h, x]
          h[x_] = f[x] - g[x]
```

Out[238]= $(1-x)\,x-\frac{8\,\mathrm{Sin}[\pi\,x]}{\pi^3}$

```
In[239]:= N[l2norm[h] / l2norm[f]]
Out[239]= 0.038013
```

The difference is less than 4%. (The previous command computed ||*f*-*g*||/||*f*||, where the norm is the L^2 norm.)

To further illustrate the capabilities of *Mathematica*, we work through two more examples from Section 3.4.

Example 3.37

The data given in this example can be stored in two vectors:

```
In[240]:= ClearAll[x, y]
          x = {0.1, 0.3, 0.4, 0.75, 0.9}
          y = {1.7805, 2.2285, 2.3941, 3.2226, 3.5697}
Out[241]= {0.1, 0.3, 0.4, 0.75, 0.9}
Out[242]= {1.7805, 2.2285, 2.3941, 3.2226, 3.5697}
```

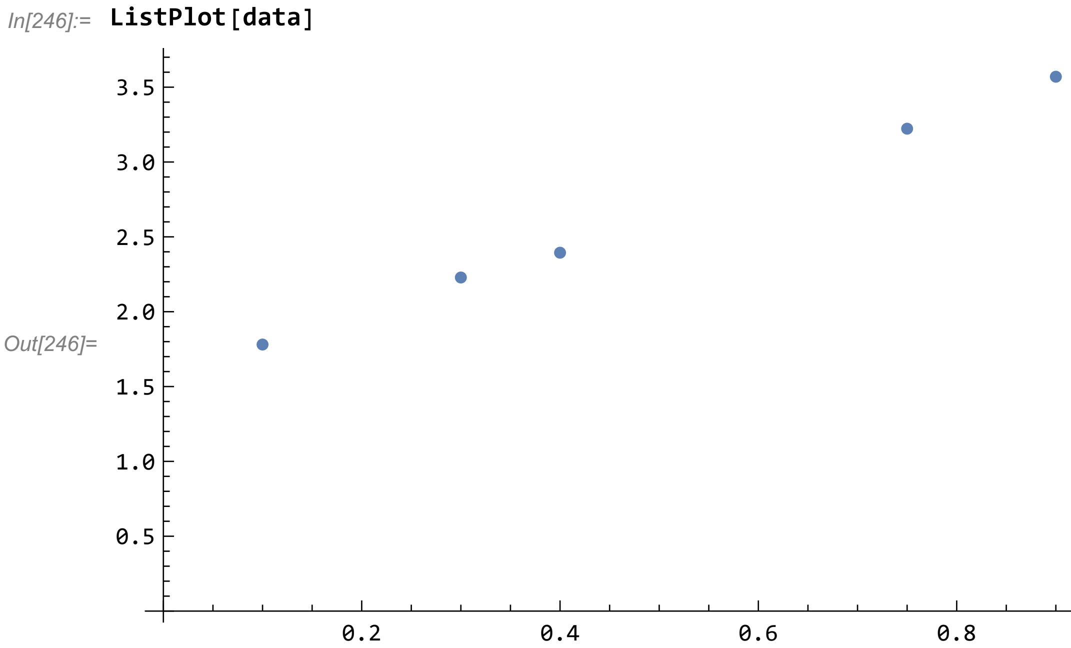
When working with discrete data like this, a useful command is **ListPlot**, which plots a collection of points in the plane. In order to use this command, the data must be stored as (x,y) pairs in a table or a matrix with two columns. Here is one way to do this:

```
In[243]:= ClearAll[data]
          data = Transpose[{x, y}]
          MatrixForm[data]
Out[244]= {{0.1, 1.7805}, {0.3, 2.2285}, {0.4, 2.3941}, {0.75, 3.2226}, {0.9, 3.5697}}
```

Out[245]//MatrixForm=

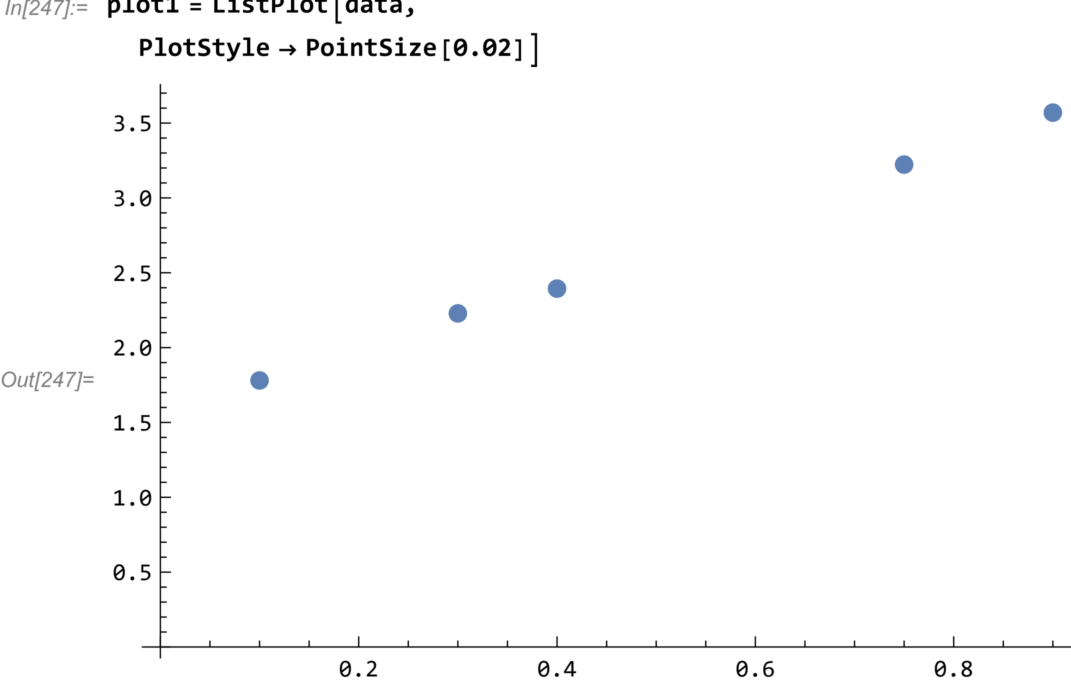
$$\begin{pmatrix} 0.1 & 1.7805 \\ 0.3 & 2.2285 \\ 0.4 & 2.3941 \\ 0.75 & 3.2226 \\ 0.9 & 3.5697 \end{pmatrix}$$

In[246]:=
```
ListPlot[data]
```

Out[246]=


If the dots are not easily visible, they can be given a greater weight:

In[247]:=
```
plot1 = ListPlot[data,
  PlotStyle → PointSize[0.02]]
```

Out[247]=


Now we compute the first-degree function $f(x)=mx+b$ that best fits this data. The Gram matrix and the right-hand side of the normal equations are computed as follows:

```
In[248]:= ClearAll[e, G, z]
          e = {1, 1, 1, 1, 1}
          G = {{x.x, x.e}, {e.x, e.e}}
          z = {x.y, e.y}
Out[249]= {1, 1, 1, 1, 1}
Out[250]= {{1.6325, 2.45}, {2.45, 5}}
Out[251]= {7.43392, 13.1954}
```

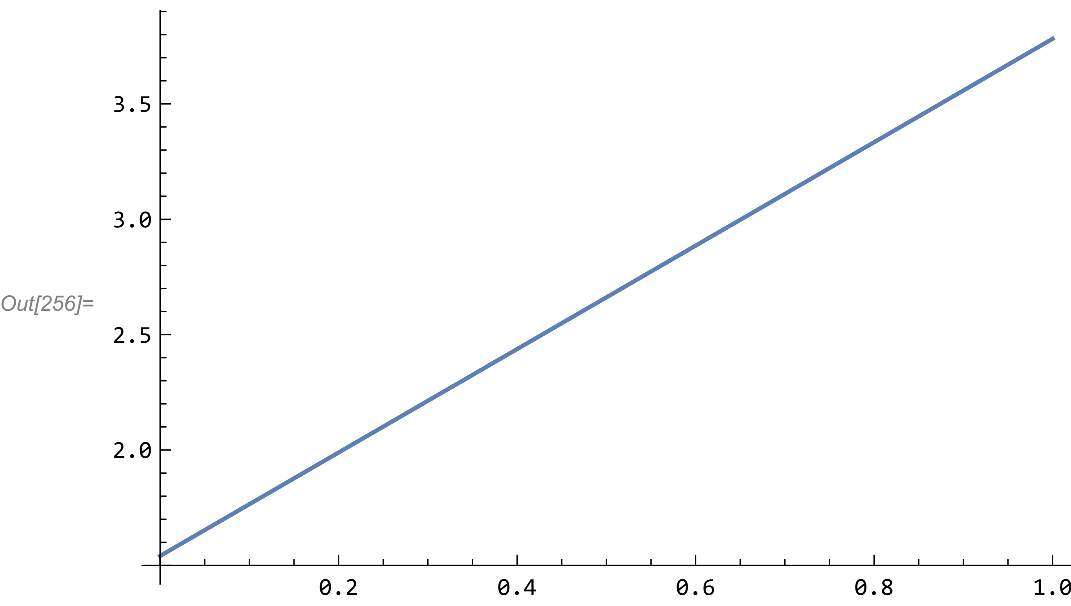
Now we can solve for the coefficients in the best approximation:

```
In[252]:= ClearAll[c]
          c = LinearSolve[G, z]
Out[253]= {2.24114, 1.54092}
```

The solution is shown below:

```
In[254]:= ClearAll[l, s]
          l[s_] = c[[1]] s + c[[2]]
          plot2 = Plot[l[s], {s, 0, 1}]
Out[255]= 1.54092 + 2.24114 s
```

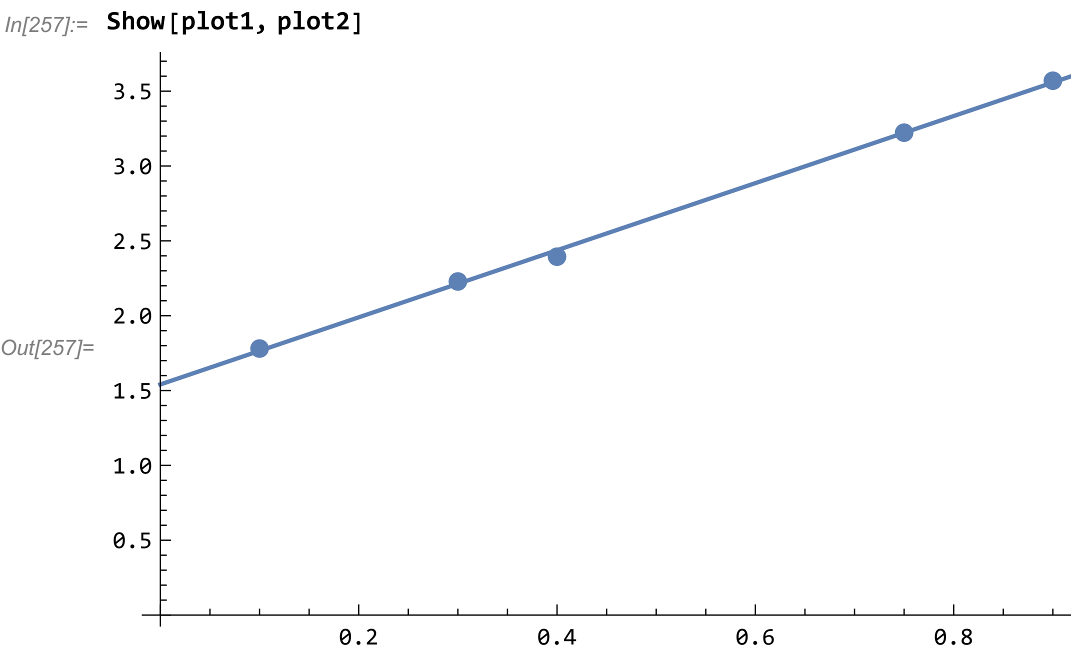
Out[256]=



The **Show** command allows us to display two (or more) graphs together. In this case, we would like to see the data and the approximating line together. Above, the two graphs were assigned variable names (**plot1**, **plot2).**

In[257]:= `Show[plot1, plot2]`

Out[257]=



The fit is not bad.

Example 3.38

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

In[258]:=
```
ClearAll[p1, p2, p3, x]
p1[x_] = 1
p2[x_] = x
p3[x_] = x^2
```

Out[259]= 1

Out[260]= x

Out[261]= x^2

We now compute the Gram matrix and the right-hand side of the normal equations. Notice the function e^x is named **Exp**.

In[262]:=
```
ClearAll[G]
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]}}
```

Out[263]= $\left\{\left\{1, \frac{1}{2}, \frac{1}{3}\right\}, \left\{\frac{1}{2}, \frac{1}{3}, \frac{1}{4}\right\}, \left\{\frac{1}{3}, \frac{1}{4}, \frac{1}{5}\right\}\right\}$

In[264]:=
```
ClearAll[b]
b = {l2ip[p1, Exp], l2ip[p2, Exp], l2ip[p3, Exp]}
```

Out[265]= $\{-1 + e, 1, -2 + e\}$

Now we solve the normal equations and find the best quadratic approximation:

In[266]:=
```
ClearAll[c]
c = LinearSolve[G, b]
```

Out[267]= {3 (-35 + 13 ⅇ), -12 (-49 + 18 ⅇ), 30 (-19 + 7 ⅇ)}

In[268]:=
```
ClearAll[q, x]
q[x_] = c[[1]] × p1[x] + c[[2]] × p2[x] + c[[3]] × p3[x]
```

Out[269]= $3(-35+13e) - 12(-49+18e)x + 30(-19+7e)x^2$

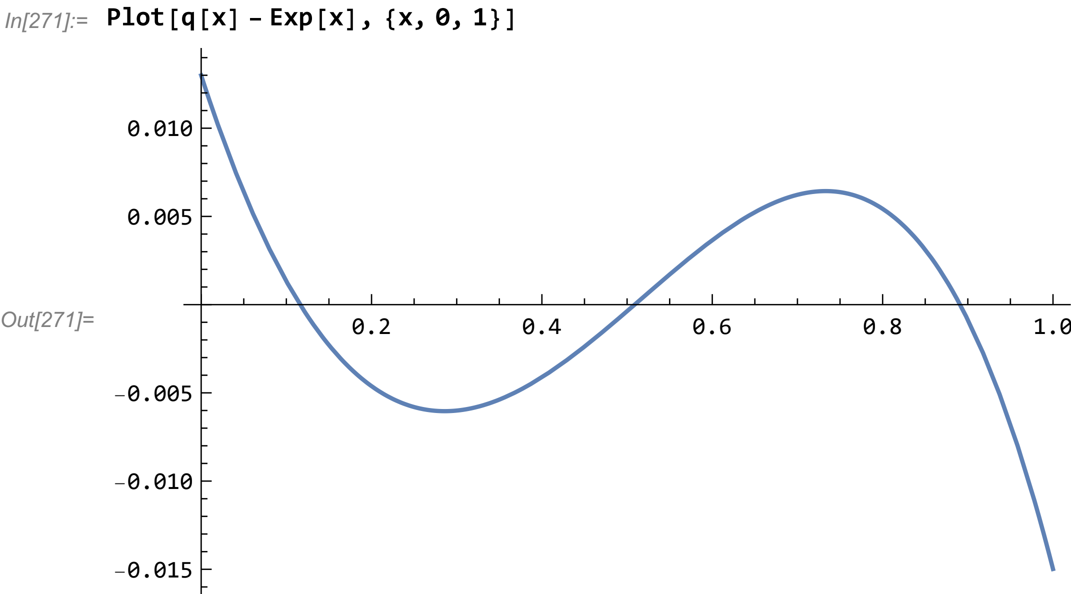
One way to judge the goodness of fit is with a plot:

In[270]:=
```
Plot[{Exp[x], q[x]}, {x, 0, 1}]
```

Out[270]=

The fit is quite good, so it is more informative to graph the error:

In[271]:=
```
Plot[q[x] - Exp[x], {x, 0, 1}]
```

Out[271]=


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

In[272]:=
```
ClearAll[h, x]
```

In[273]:=
```
h[x_] = Exp[x] - q[x]
```

Out[273]= $e^x - 3\,(-35 + 13\,e) + 12\,(-49 + 18\,e)\,x - 30\,(-19 + 7\,e)\,x^2$

In[274]:=
```
N[l2norm[h] / l2norm[Exp]]
```

Out[274]= 0.00295186

The error is less than 0.3%.

Particularly if you wish to use a subspace with a larger dimension, typing in the formula for the Gram matrix and the right-hand side of the normal equations can be quite monotonous. One can avoid the repetitive typing, but only by defining the basis functions differently. Here is an example:

In[275]:=
```
ClearAll[p, x]
p[1][x_] = 1
p[2][x_] = x
p[3][x_] = x^2
```

Out[276]= 1

Out[277]= x

Out[278]= x^2

Now, for each i=1,2,3, **p[i]** is one of the basis functions. Here is the third, for example:

In[279]:=
```
p[3][x]
```

Out[279]= x^2

We can now define the Gram matrix using a **Table** command:

```
In[280]:= ClearAll[G]
          G = Table[l2ip[p[i], p[j]], {i, 1, 3}, {j, 1, 3}]
```

Out[281]= $\left\{\left\{1, \frac{1}{2}, \frac{1}{3}\right\}, \left\{\frac{1}{2}, \frac{1}{3}, \frac{1}{4}\right\}, \left\{\frac{1}{3}, \frac{1}{4}, \frac{1}{5}\right\}\right\}$

A similar command computes the right-hand side:

```
In[282]:= ClearAll[b]
          b = Table[l2ip[p[i], Exp], {i, 1, 3}]
```

Out[283]= $\{-1 + e, 1, -2 + e\}$

Now we solve as before:

```
In[284]:= ClearAll[c]
          c = LinearSolve[G, b]
```

Out[285]= $\{3\,(-35 + 13\,e), -12\,(-49 + 18\,e), 30\,(-19 + 7\,e)\}$

Further examples of the **Table** command will be given below. Remember that you can also consult the help browser for more information.

■ Section 3.5: Eigenvalues and eigenvectors of a symmetric matrix

Mathematica can compute the eigenvalues and eigenvectors of a square matrix:

```
In[286]:= ClearAll[A]
          A = {{1, 2, -1}, {4, 0, 1}, {-7, -2, 3}}
          MatrixForm[A]
```

Out[287]= {{1, 2, -1}, {4, 0, 1}, {-7, -2, 3}}

Out[288]//MatrixForm=

$$\begin{pmatrix} 1 & 2 & -1 \\ 4 & 0 & 1 \\ -7 & -2 & 3 \end{pmatrix}$$

```
In[289]:= ClearAll[evals, evecs]
          evals = Eigenvalues[A]
          evecs = Eigenvectors[A]
```

Out[290]= $\left\{1 + \sqrt{15}, 1 - \sqrt{15}, 2\right\}$

Out[291]= $\left\{\left\{-\frac{-1 + \sqrt{15}}{7 + \sqrt{15}}, -\frac{4 - \sqrt{15}}{7 + \sqrt{15}}, 1\right\}, \left\{-\frac{1 + \sqrt{15}}{-7 + \sqrt{15}}, -\frac{-4 - \sqrt{15}}{-7 + \sqrt{15}}, 1\right\}, \{0, 1, 2\}\right\}$

Note that the eigenvalues are returned as a list of numbers, while the eigenvectors are returned as a list of vectors. As far as we know, there is no guarantee that the order in these two lists is the same, that is, that the first eigenvector listed corresponds to the first eigenvalue, for example. However, the command **Eigensystem** combines the two previous commands, and the orders of the eigenvalues and eigenvectors are guaranteed to correspond:

In[292]:=
```
ClearAll[evals, evecs]
{evals, evecs} = Eigensystem[A]
```

Out[293]= $\left\{\left\{1+\sqrt{15},\ 1-\sqrt{15},\ 2\right\},\right.$
$\left.\left\{\left\{-\frac{-1+\sqrt{15}}{7+\sqrt{15}},\ -\frac{4-\sqrt{15}}{7+\sqrt{15}},\ 1\right\},\ \left\{-\frac{1+\sqrt{15}}{-7+\sqrt{15}},\ -\frac{-4-\sqrt{15}}{-7+\sqrt{15}},\ 1\right\},\ \{0,1,2\}\right\}\right\}$

(Thus, if you want both the eigenvalues and the corresponding eigenvectors, you should use the **Eigensystem** command, not **Eigenvalues** or **Eigenvectors**.)

We now have a list of eigenvalues and a list of eigenvectors. The individual eigenvalues or eigenvectors can be accessed using the index operator. For instance, here is the first eigenvalue/eigenvector pair:

In[294]:=
```
evals[[1]]
evecs[[1]]
```

Out[294]= $1+\sqrt{15}$

Out[295]= $\left\{-\frac{-1+\sqrt{15}}{7+\sqrt{15}},\ -\frac{4-\sqrt{15}}{7+\sqrt{15}},\ 1\right\}$

We can check to see if *Mathematica*'s calculation was correct. If λ, x form an eigenvalue/eigenvector pair, then $Ax - \lambda x = 0$ must hold:

In[296]:=
```
A.evecs[[1]] - evals[[1]] * evecs[[1]]
```

Out[296]= $\left\{-1-\frac{2\left(4-\sqrt{15}\right)}{7+\sqrt{15}}-\frac{-1+\sqrt{15}}{7+\sqrt{15}}+\frac{\left(-1+\sqrt{15}\right)\left(1+\sqrt{15}\right)}{7+\sqrt{15}},\right.$
$\left.1-\frac{4\left(-1+\sqrt{15}\right)}{7+\sqrt{15}}+\frac{\left(4-\sqrt{15}\right)\left(1+\sqrt{15}\right)}{7+\sqrt{15}},\ 2-\sqrt{15}+\frac{2\left(4-\sqrt{15}\right)}{7+\sqrt{15}}+\frac{7\left(-1+\sqrt{15}\right)}{7+\sqrt{15}}\right\}$

Does the above result mean that *Mathematica* made a mistake? Before jumping to this conclusion, we should make sure that the result is simplified as much as possible:

In[297]:=
```
Simplify[%]
```

Out[297]= {0, 0, 0}

The problem of computing the eigenvalues of an *n* by *n* matrix is equivalent to the problem of finding the roots of an *n*th-degree polynomial. One of the most famous theoretical results of the 19th century 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. Therefore, asking *Mathematica* for the eigenvalues of large matrix is most likely an exercise in futility:

In[298]:=
```
ClearAll[B]
B = Table[1/(i+j+1), {i, 1, 5}, {j, 1, 5}];
MatrixForm[B]
```

Out[300]//MatrixForm=

$$\begin{pmatrix} \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} \\ \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} \\ \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} \\ \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} & \frac{1}{10} \\ \frac{1}{7} & \frac{1}{8} & \frac{1}{9} & \frac{1}{10} & \frac{1}{11} \end{pmatrix}$$

In[301]:=
```
ClearAll[evals1]
evals1 = Eigenvalues[B]
```

Out[302]=
```
{Root[-1 + 5 603 255 #1 - 246 181 488 000 #1^2 +
    191 979 048 240 000 #1^3 - 4 545 067 645 440 000 #1^4 + 5 175 372 787 200 000 #1^5 &, 5],
 Root[-1 + 5 603 255 #1 - 246 181 488 000 #1^2 + 191 979 048 240 000 #1^3 -
    4 545 067 645 440 000 #1^4 + 5 175 372 787 200 000 #1^5 &, 4],
 Root[-1 + 5 603 255 #1 - 246 181 488 000 #1^2 + 191 979 048 240 000 #1^3 -
    4 545 067 645 440 000 #1^4 + 5 175 372 787 200 000 #1^5 &, 3],
 Root[-1 + 5 603 255 #1 - 246 181 488 000 #1^2 + 191 979 048 240 000 #1^3 -
    4 545 067 645 440 000 #1^4 + 5 175 372 787 200 000 #1^5 &, 2],
 Root[-1 + 5 603 255 #1 - 246 181 488 000 #1^2 + 191 979 048 240 000 #1^3 -
    4 545 067 645 440 000 #1^4 + 5 175 372 787 200 000 #1^5 &, 1]}
```

Mathematica cannot find the roots of the characteristic polynomial of **B**; the output simply indicates that the desired eigenvalues are the roots of the fifth-degree polynomial shown above.

In a case like this, it is possible to have *Mathematica* estimate the roots numerically. In fact, *Mathematica* will perform a numerical computation instead of a symbolic computation whenever the input matrix has floating point entries instead of symbolic entries.

In[303]:=
```
N[B] // MatrixForm
```

Out[303]//MatrixForm=

$$\begin{pmatrix} 0.333333 & 0.25 & 0.2 & 0.166667 & 0.142857 \\ 0.25 & 0.2 & 0.166667 & 0.142857 & 0.125 \\ 0.2 & 0.166667 & 0.142857 & 0.125 & 0.111111 \\ 0.166667 & 0.142857 & 0.125 & 0.111111 & 0.1 \\ 0.142857 & 0.125 & 0.111111 & 0.1 & 0.0909091 \end{pmatrix}$$

In[304]:=
```
ClearAll[evals2, evecs2]
{evals2, evecs2} = Eigensystem[N[B]]
```

Out[305]= $\{\{0.83379, 0.0430979, 0.00129982, 0.0000229962, 1.79889\times 10^{-7}\},$
$\{\{-0.612635, -0.489105, -0.408486, -0.351316, -0.308497\},$
$\{0.684809, 0.0159186, -0.287687, -0.435542, -0.508254\},$
$\{0.374365, -0.632042, -0.347579, 0.13307, 0.567323\},$
$\{0.122651, -0.564324, 0.514257, 0.403397, -0.489189\},$
$\{0.0228469, -0.206359, 0.604238, -0.711648, 0.292141\}\}\}$

Let us check the result for the first eigenvalue/eigenvector pair:

In[306]:=
```
B.evecs2[[1]] - evals2[[1]] * evecs2[[1]]
```

Out[306]= $\{-1.11022\times 10^{-16}, 1.11022\times 10^{-16}, -1.11022\times 10^{-16}, -5.55112\times 10^{-17}, 0.\}$

As should be expected for a computation performed in finite precision arithmetic, the expected relationship fails to hold, but only by an amount attributable to roundoff error.

Example 3.49

We now use the spectral method to solve Ax = b for the following A and b:

In[307]:=
```
ClearAll[A, b]
A = {{11, -4, -1}, {-4, 14, -4}, {-1, -4, 11}}
b = {1, 2, 1}
```

Out[308]= {{11, -4, -1}, {-4, 14, -4}, {-1, -4, 11}}

Out[309]= {1, 2, 1}

We need the eigenpairs of A:

In[310]:=
```
ClearAll[evals, evecs]
{evals, evecs} = Eigensystem[A]
```

Out[311]= {{18, 12, 6}, {{1, -2, 1}, {-1, 0, 1}, {1, 1, 1}}}

Here is an important point: *Mathematica* does not necessarily normalize the eigenvectors it returns. These eigenvectors are orthogonal since A is symmetric and the eigenvalues are distinct, but we must normalize the eigenvectors manually. We call the normalized eigenvectors **u1**, **u2**, **u3**:

```
In[312]:= Clear[u1, u2, u3]
          u1 = evecs[[1]] / Norm[evecs[[1]]]
          u2 = evecs[[2]] / Norm[evecs[[2]]]
          u3 = evecs[[3]] / Norm[evecs[[3]]]
```

Out[313]= $\left\{\frac{1}{\sqrt{6}}, -\sqrt{\frac{2}{3}}, \frac{1}{\sqrt{6}}\right\}$

Out[314]= $\left\{-\frac{1}{\sqrt{2}}, 0, \frac{1}{\sqrt{2}}\right\}$

Out[315]= $\left\{\frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}\right\}$

The solution of $Ax=b$ is computed as follows:

```
In[316]:= ClearAll[x]
          x = (u1.b / 18) u1 + (u2.b / 12) u2 + (u3.b / 6) u3
```

Out[317]= $\left\{\frac{11}{54}, \frac{7}{27}, \frac{11}{54}\right\}$

Check:

```
In[318]:= A.x - b
Out[318]= {0, 0, 0}
```

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:

$$\frac{d^2u}{dt^2}+4\frac{du}{dt}-3u=0,$$

$$u(0)=1,$$

$$\frac{du}{dt}(0)=0.$$

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

In[319]:=
```
ClearAll[r]
roots = Solve[r^2 + 4 r - 3 == 0, r]
```

Out[320]= $\{\{r \to -2 - \sqrt{7}\}, \{r \to -2 + \sqrt{7}\}\}$

The general solution is given below:

In[321]:=
```
r1 = (r /. roots[[1]])
r2 = (r /. roots[[2]])
ClearAll[u, t, c1, c2]
u[t_] = c1 Exp[r1 t] + c2 Exp[r2 t]
```

Out[321]= $-2 - \sqrt{7}$

Out[322]= $-2 + \sqrt{7}$

Out[324]= $c1\, e^{(-2-\sqrt{7})\, t} + c2\, e^{(-2+\sqrt{7})\, t}$

We can now solve for the unknowns **c1**, **c2**:

In[325]:=
```
ClearAll[sols]
sols = Solve[{u[0] == 1, (D[u[t], t] /. t → 0) == 0}, {c1, c2}]
c1 = (c1 /. sols[[1]])
c2 = (c2 /. sols[[1]])
```

Out[326]= $\left\{\left\{c1 \to \frac{1}{14}\left(7 - 2\sqrt{7}\right), c2 \to \frac{1}{14}\left(7 + 2\sqrt{7}\right)\right\}\right\}$

Out[327]= $\frac{1}{14}\left(7 - 2\sqrt{7}\right)$

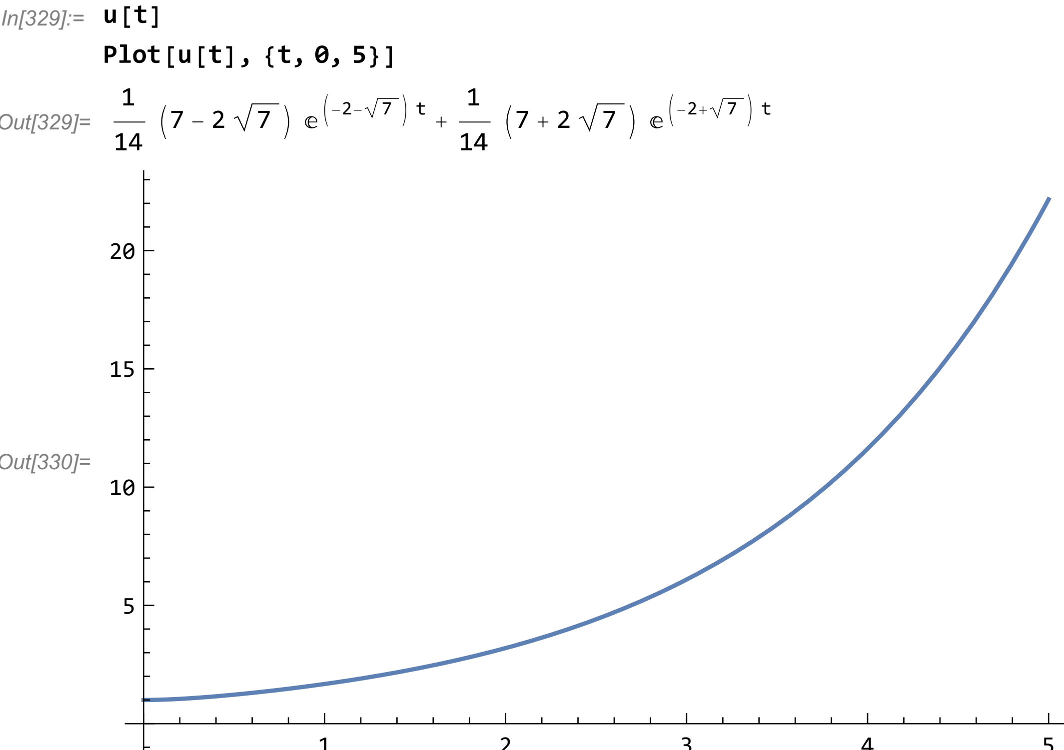
Out[328]= $\frac{1}{14}\left(7 + 2\sqrt{7}\right)$

Here is the solution of the IVP:

In[329]:=
```
u[t]
Plot[u[t], {t, 0, 5}]
```

Out[329]= $\frac{1}{14}\left(7-2\sqrt{7}\right)e^{\left(-2-\sqrt{7}\right)t}+\frac{1}{14}\left(7+2\sqrt{7}\right)e^{\left(-2+\sqrt{7}\right)t}$

Out[330]=



A special inhomogeneous second-order linear ODE

Consider the IVP

$$\frac{d^2u}{dt^2}+4u=\sin(\pi t),$$

$$u(0)=0,$$

$$\frac{du}{dt}(0)=0.$$

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

In[331]:=
```
ClearAll[u, t, s]
u[t_] = Simplify[(1 / 2) Integrate[Sin[2 (t - s)] Sin[Pi s], {s, 0, t}]]
```

Out[332]= $\frac{\pi \text{Cos}[t]\,\text{Sin}[t]-\text{Sin}[\pi t]}{-4+\pi^2}$

Let us check this solution:

In[333]:=
```
D[u[t], {t, 2}] + 4 u[t]
```

Out[333]= $\frac{4\,(\pi \text{Cos}[t]\,\text{Sin}[t]-\text{Sin}[\pi t])}{-4+\pi^2}+\frac{-4\pi \text{Cos}[t]\,\text{Sin}[t]+\pi^2\,\text{Sin}[\pi t]}{-4+\pi^2}$

In[334]:=
```
Simplify[%]
```

Out[334]= Sin[π t]

In[335]:=
```
u[0]
```
Out[335]= 0

In[336]:=
```
u'[0]
```
Out[336]= 0

First-order linear ODEs

Now consider the following IVP:

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

Section 4.2.4 contains an explicit formula for the solution:

In[337]:=
```
ClearAll[u, t, s]
u[t_] = Simplify[Exp[1 / 2 t] + Integrate[Exp[1 / 2 (t - s)] (-s), {s, 0, t}]]
```
Out[338]= $4 - 3 e^{t/2} + 2 t$

Here is a graph:

In[339]:=
```
Plot[u[t], {t, 0, 3}]
```
Out[339]=

Just out of curiosity, let us determine the value of t at which u[t] is zero:

```
In[340]:= Solve[u[t] == 0, t]
```

Solve::ifun :
Inverse functions are being used by Solve, so some solutions may not be found; use Reduce for complete solution information. ≫

$$\text{Out[340]=} \left\{\left\{t \to -2\left(1 + \text{ProductLog}\left[-\frac{3}{4\,e}\right]\right)\right\}, \left\{t \to -2\left(1 + \text{ProductLog}\left[-1, -\frac{3}{4\,e}\right]\right)\right\}\right\}$$

This type of result is sometimes seen when using *Mathematica*; the program is aware of many special mathematical functions, and results are sometimes expressed using functions that are unfamiliar. In this case, we can obtain a more meaningful answer using numerical evaluation:

```
In[341]:= N[%]
Out[341]= {{t → -1.16026}, {t → 1.92256}}
```

The second solution is the one we are seeking.

Mathematica has another command for solving equations, one which uses numerical rather than symbolic methods. This command, called FindRoot, needs a starting point (an estimate of the desired solution):

```
In[342]:= FindRoot[u[t] == 0, {t, 1.9}]
Out[342]= {t → 1.92256}
```

■ Section 4.3: Linear systems with constant coefficients

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

$$\frac{dx}{dt} = Ax,$$
$$x(0) = x0,$$

where **A** and **x0** are the following matrix and vector:

```
In[343]:= ClearAll[A, x0]
          A = {{1, 2}, {3, 4}}
          x0 = {4, 1}
Out[344]= {{1, 2}, {3, 4}}
Out[345]= {4, 1}
```

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

```
In[346]:= ClearAll[l, V]
          {l, V} = Eigensystem[A]
```

$$\text{Out[347]=} \left\{\left\{\frac{1}{2}\left(5 + \sqrt{33}\right), \frac{1}{2}\left(5 - \sqrt{33}\right)\right\}, \left\{\left\{\frac{1}{6}\left(-3 + \sqrt{33}\right), 1\right\}, \left\{\frac{1}{6}\left(-3 - \sqrt{33}\right), 1\right\}\right\}\right\}$$

Next, the initial vector must be expressed in terms of the eigenvectors of *A*:

In[348]:=
```
ClearAll[c]
c = LinearSolve[Transpose[V], x0]
```

Out[349]= $\left\{\frac{1}{22}\left(11+9\sqrt{33}\right), \frac{1}{22}\left(11-9\sqrt{33}\right)\right\}$

Here is the solution:

In[350]:=
```
ClearAll[x]
x[t_] = c[[1]] Exp[l[[1]] t] V[[1]] + c[[2]] Exp[l[[2]] t] V[[2]]
```

Out[351]= $\left\{\frac{1}{132}\left(11-9\sqrt{33}\right)\left(-3-\sqrt{33}\right)e^{\frac{1}{2}\left(5-\sqrt{33}\right)t}+\frac{1}{132}\left(-3+\sqrt{33}\right)\left(11+9\sqrt{33}\right)e^{\frac{1}{2}\left(5+\sqrt{33}\right)t}, \frac{1}{22}\left(11-9\sqrt{33}\right)e^{\frac{1}{2}\left(5-\sqrt{33}\right)t}+\frac{1}{22}\left(11+9\sqrt{33}\right)e^{\frac{1}{2}\left(5+\sqrt{33}\right)t}\right\}$

Notice that $x(t)$ is a vector (that is, x is a vector-valued function). For example, the first component is

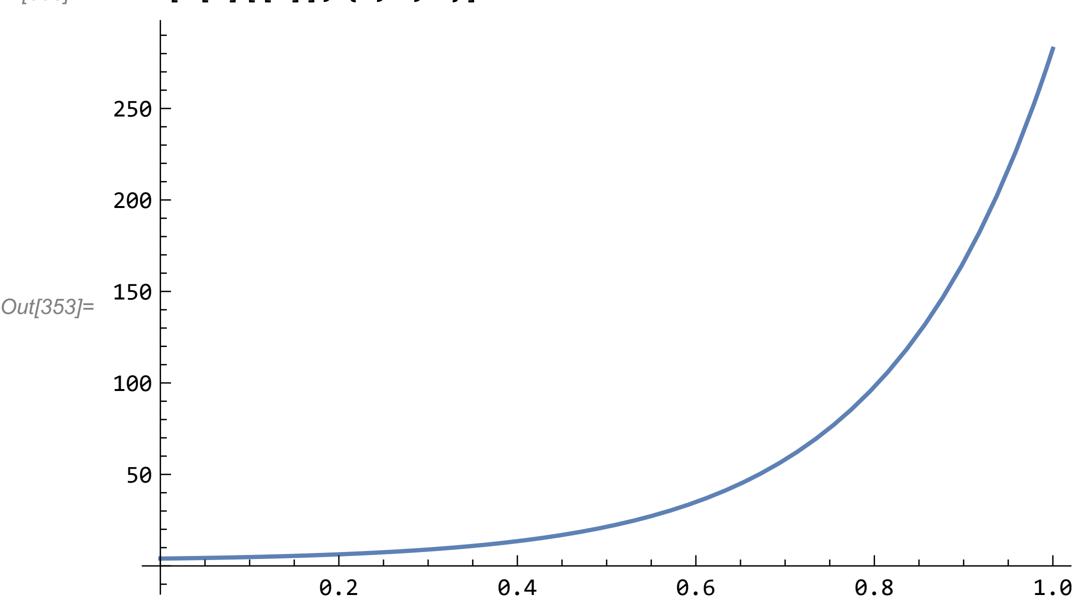
In[352]:=
```
x[t][[1]]
```

Out[352]= $\frac{1}{132}\left(11-9\sqrt{33}\right)\left(-3-\sqrt{33}\right)e^{\frac{1}{2}\left(5-\sqrt{33}\right)t}+\frac{1}{132}\left(-3+\sqrt{33}\right)\left(11+9\sqrt{33}\right)e^{\frac{1}{2}\left(5+\sqrt{33}\right)t}$

Here is its graph:

In[353]:=
```
Plot[x[t][[1]], {t, 0, 1}]
```

Out[353]=



(Notice that the solution is dominated by a rapidly growing exponential.)

Inhomogeneous systems and variation of parameters

Next, we use *Mathematica* to solve inhomogeneous systems of the form

$$\frac{dx}{dt}=Ax+f(t),$$

$x(0) = x_0.$

Consider the following matrix, (vector-valued) function, and initial vector:

```
In[354]:= ClearAll[A, f, t]
A = {{1, 2}, {2, 1}}
f[t_] = {Sin[t], 0}
x0 = {0, 1}
Out[355]= {{1, 2}, {2, 1}}
Out[356]= {Sin[t], 0}
Out[357]= {0, 1}
```

First, we find the eigenvalues and eigenvectors of A:

```
In[358]:= ClearAll[l, U]
{l, U} = Eigensystem[A]
Out[359]= {{3, -1}, {{1, 1}, {-1, 1}}}
```

Next, we normalize the eigenvectors (which are orthogonal, as they must be):

```
In[360]:= ClearAll[u1, u2]
u1 = U[[1]] / Sqrt[U[[1]].U[[1]]]
u2 = U[[2]] / Sqrt[U[[2]].U[[2]]]
```

Out[361]= $\left\{\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}\right\}$

Out[362]= $\left\{-\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}\right\}$

We have $f(t) = c_1(t)\, u_1 + c_2(t)\, u_2$ and $x_0 = b_1\, u_1 + b_2\, u_2$, where c_1, c_2, b_1, b_2 are calculated as follows;

```
In[363]:= ClearAll[c1, c2, b1, b2, t]
c1[t_] = u1.f[t]
c2[t_] = u2.f[t]
b1 = u1.x0
b2 = u2.x0
```

Out[364]= $\frac{\text{Sin}[t]}{\sqrt{2}}$

Out[365]= $-\frac{\text{Sin}[t]}{\sqrt{2}}$

Out[366]= $\frac{1}{\sqrt{2}}$

Out[367]= $\frac{1}{\sqrt{2}}$

We then solve the two decoupled IVPs

$$\frac{da_1}{dt} = \lambda_1 a_1 + c_1(t),\ a_1(0) = b_1,$$

$$\frac{da_2}{dt} = \lambda_2 a_2 + c_2(t),\ a_2(0) = b_2.$$

The solutions, using the techniques of Section 4.2, are computed as follows:

```
In[368]:= ClearAll[a1, a2, t]
          a1[t_] = b1 Exp[l[[1]] t] + Integrate[Exp[l[[1]] (t - s)] c1[s], {s, 0, t}]
          a2[t_] = b2 Exp[l[[2]] t] + Integrate[Exp[l[[2]] (t - s)] c2[s], {s, 0, t}]
```

Out[369]= $\frac{e^{3t}}{\sqrt{2}} - \frac{-e^{3t} + \text{Cos}[t] + 3\,\text{Sin}[t]}{10\sqrt{2}}$

Out[370]= $\frac{e^{-t}}{\sqrt{2}} - \frac{e^{-t} - \text{Cos}[t] + \text{Sin}[t]}{2\sqrt{2}}$

The solution to the original system is then

```
In[371]:= ClearAll[x, t]
          x[t_] = Simplify[a1[t] u1 + a2[t] u2]
```

Out[372]= $\left\{\frac{1}{20}\left(-5\,e^{-t} + 11\,e^{3t} - 6\,\text{Cos}[t] + 2\,\text{Sin}[t]\right),\ \frac{1}{20}\left(5\,e^{-t} + 11\,e^{3t} + 4\,\text{Cos}[t] - 8\,\text{Sin}[t]\right)\right\}$

Check:

```
In[373]:= D[x[t], t] - A.x[t] - f[t]
```

Out[373]= $\left\{\frac{1}{20}\left(5\,e^{-t} - 11\,e^{3t} + 6\,\text{Cos}[t] - 2\,\text{Sin}[t]\right) - \text{Sin}[t] + \frac{1}{20}\left(5\,e^{-t} + 33\,e^{3t} + 2\,\text{Cos}[t] + 6\,\text{Sin}[t]\right) + \frac{1}{10}\left(-5\,e^{-t} - 11\,e^{3t} - 4\,\text{Cos}[t] + 8\,\text{Sin}[t]\right),\ \frac{1}{20}\left(-5\,e^{-t} + 33\,e^{3t} - 8\,\text{Cos}[t] - 4\,\text{Sin}[t]\right) + \frac{1}{10}\left(5\,e^{-t} - 11\,e^{3t} + 6\,\text{Cos}[t] - 2\,\text{Sin}[t]\right) + \frac{1}{20}\left(-5\,e^{-t} - 11\,e^{3t} - 4\,\text{Cos}[t] + 8\,\text{Sin}[t]\right)\right\}$

```
In[374]:= Simplify[%]
Out[374]= {0, 0}

In[375]:= x[0] - x0
Out[375]= {0, 0}
```

■ Section 4.4: Numerical methods for initial value problems

When we turn to numerical methods for initial value problems in ODEs, we naturally wish to write programs to implement the methods. Since time-stepping methods involve repeated steps of the same form, it would be quite tedious to apply the methods manually. Here is another strength of *Mathematica*: not only does it integrate symbolic and numeric computation

with graphics, but it also provides a programming environment. In this section, we explain the basics of programming in *Mathematica*.

Interactive commands

Mathematica supports the usual programming constructs, which can be used both interactively and in programs. For example, suppose we wish to apply Euler's method to estimate the solution of

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

on the interval [0,1]. The exact solution, which we will use below to test our results, is the following function:

```
In[376]:= ClearAll[v, t]
          v[t_] = Exp[ArcTan[t]]
Out[377]= ⅇ^ArcTan[t]
```

We now apply Euler's method with a step length of Δt=0.1. The new *Mathematica* command that we need is the **Do** command, which implements an indexed loop. We also need to know how to store the computed results; since Euler's method (and other time-stepping methods for IVPs) produces estimates of the solution on a grid, we will store the results as a list of points of the form (t_i,u_i), $i=1,2,\ldots,n$. The results will then be of the right form to pass to **ListPlot** and other useful commands.

```
In[378]:= ClearAll[dt, n]
          dt = 0.1
          n = 10
Out[379]= 0.1
Out[380]= 10
```

The following command creates an empty list of points (the **Table** command is useful for creating lists):

```
In[381]:= ClearAll[U]
          U = Table[{Null, Null}, {i, 0, n}]
Out[382]= {{Null, Null}, {Null, Null}, {Null, Null}, {Null, Null}, {Null, Null}, {Null, Null},
           {Null, Null}, {Null, Null}, {Null, Null}, {Null, Null}, {Null, Null}}
```

It is not very instructive to see the output of the last command, and this is often the case. *Mathematica* will not display the output of commands that end in a semicolon, as follows:

```
In[383]:= U = Table[{Null, Null}, {i, 0, n}];
```

We now assign the initial values of t and u:

```
In[384]:= U[[1, 1]] = 0.
          U[[1, 2]] = 1.
Out[384]= 0.
Out[385]= 1.
```

Here is an important but subtle point. When implementing a numerical method, we want to do floating point, not symbolic, computations. This is why the initial data is entered as "0." and "1.", rather than "0" and "1". When the following loop

executes, the operations will be done in floating point. If we used symbolic data for the initial data, the computations would be done symbolically, which is comparatively slow and undesirable in most cases.

Euler's method is now performed by a single command:

In[386]:=
```
Do[
 U[[i + 1, 1]] = U[[i, 1]] + dt;
 U[[i + 1, 2]] = U[[i, 2]] + dt * U[[i, 2]] / (1 + U[[i, 1]]^2),
 {i, 1, n}]
```

We can now look at the computed results:

In[387]:=
```
TableForm[U]
```

Out[387]//TableForm=
```
0.     1.
0.1    1.1
0.2    1.20891
0.3    1.32515
0.4    1.44673
0.5    1.57144
0.6    1.69716
0.7    1.82195
0.8    1.94423
0.9    2.06278
1.     2.17675
```

How well did Euler's method work? We can compare with the exact solution at the final time:

In[388]:=
```
N[v[1] - U[[n + 1, 2]]]
```

Out[388]= `0.016535`

Several comments about the above computations are in order.

The **Table** command produced a list of points that is indexed from 1 to n+1. Thus *Mathematica*, like Fortran or MATLAB, but unlike C, uses indices beginning with 1.

The **Null** object is used to as a place holder, indicating the absence of an expression.

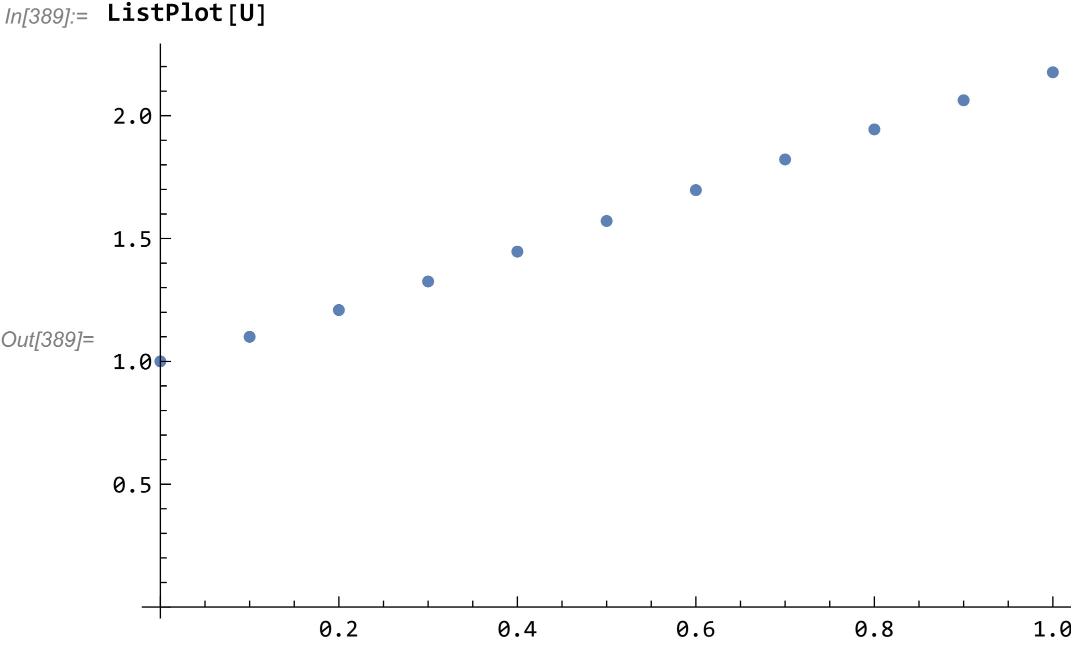
The **Do** command has the form **Do[expr,{i,imin,imax}]**. This command causes the expression *expr* to be evaluated (**imax**-**imin**+1) times, the first time with i=**imin**, the second time with i=**imin**+1, and so forth until i=**imax**.

If several commands are to be performed during each iteration of the Do loop, then *expr* can be replaced by *expr1;expr2;...;exprk*. That is, expr can be a sequence of expressions, separated by semicolons.

The loop index i can be incremented by a value other than 1; the iterator would be **{i,imin,imax,step}**, giving i=**imin**, i=**imin**+**step**, i=**imin**+2***step**,....

Mathematica gives us two ways to visualize the computed solution. First of all, we can graph the points (t_i,u_i) as points in the (t,u) plane using the **ListPlot** command:

In[389]:=
```
ListPlot[U]
```

Out[389]=


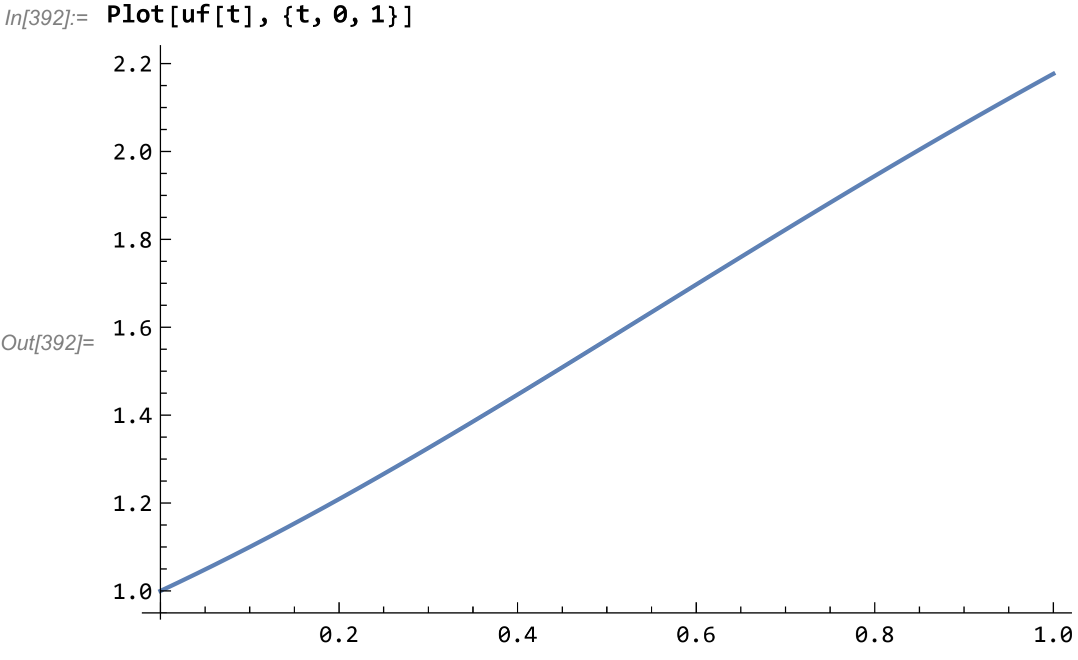
Secondly, we can create a function from the computed data; specifically, an interpolating function:

In[390]:=
```
ClearAll[uf]
uf = Interpolation[U]
```

Out[391]=
```
InterpolatingFunction[ Domain: {{0., 1.}} Output: scalar ]
```

We can then evaluate the interpolating function at any point in the interval [0,10], or plot it (or differentiate it, etc.):

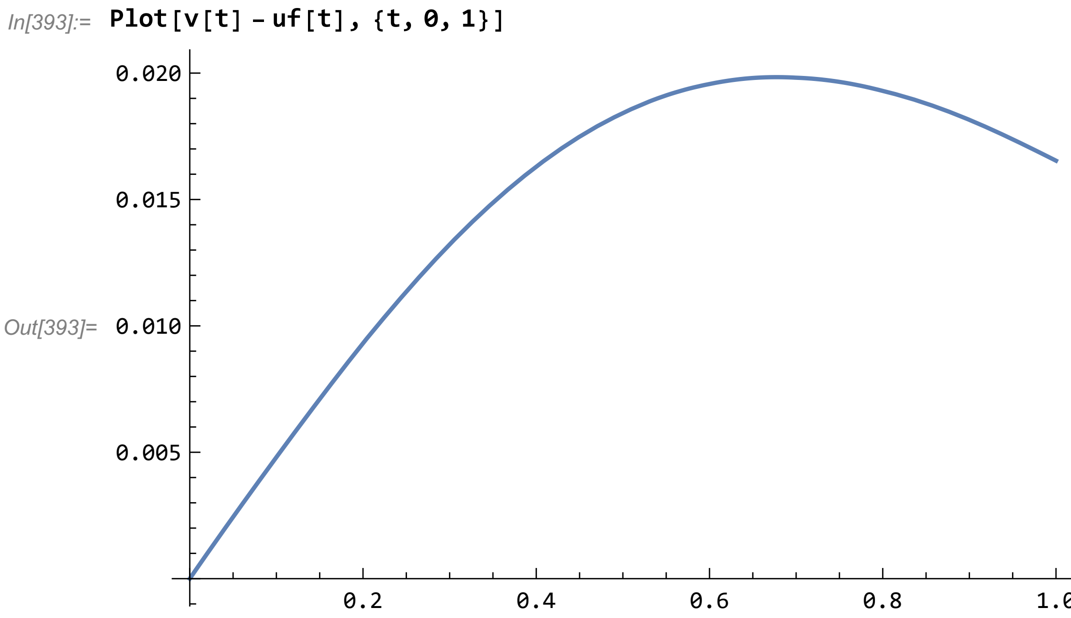
In[392]:=
```
Plot[uf[t], {t, 0, 1}]
```

Out[392]=


Here is the error in the computed solution.

```
In[393]:= Plot[v[t] - uf[t], {t, 0, 1}]
```

Out[393]=



Creating new *Mathematica* commands

For convenience, we can write a program to implement Euler's method. In *Mathematica*, writing a program really implies creating a new *Mathematica* command. A reasonable command for applying Euler's method to the IVP

$$\frac{du}{dt} = f(t, u)$$

$$u(t_0) = u_0$$

would take as inputs f, t_0, u_0, as well as T (the final time) and n (the number of steps), and would produce a list of the computed estimates (like the output U above). A simple function consists of a list of *Mathematica* commands (expressions), enclosed in parentheses; the return value of the function is simply the value of the last expression:

```
In[394]:= ClearAll[f, t0, T, u0, n, U, euler]
euler[f_, t0_, T_, u0_, n_] :=
 (U = Table[{Null, Null}, {i, 0, n}];
  U[[1, 1]] = t0;
  U[[1, 2]] = u0;
  dt = (T - t0) / n;
  Do[
   U[[i + 1, 1]] = U[[i, 1]] + dt;
   U[[i + 1, 2]] = U[[i, 2]] + dt * f[U[[i, 1]], U[[i, 2]]],
   {i, 1, n}];
  U)
```

(Notice that the use of the delayed-evaluation assignment operator ":=" is important in defining new commands.) Now let us see how the command **euler** works.

In[396]:=
```
ClearAll[f, t, u, r]
f[t_, u_] = u / (1 + t^2)
r = euler[f, 0., 1., 1.0, 10]
```

Out[397]= $\frac{u}{1 + t^2}$

Out[398]=
```
{{0., 1.}, {0.1, 1.1}, {0.2, 1.20891},
 {0.3, 1.32515}, {0.4, 1.44673}, {0.5, 1.57144}, {0.6, 1.69716},
 {0.7, 1.82195}, {0.8, 1.94423}, {0.9, 2.06278}, {1., 2.17675}}
```

In[399]:=
```
ListPlot[r]
```

Out[399]=

In *Mathematica*, one need not "declare" variables in the same way one does in a compiled language. For example, in the **euler** function, there is no instruction to *Mathematica* to expect a scalar input for u_0, or to expect f to be a scalar-valued function. Therefore, **euler** will work just as well for a system, as the following examples shows.

Consider the system

In[144]:=

$$\frac{dx_1}{dt} = -x_2,\ x_1(0) = 1$$

$$\frac{dx_2}{dt} = x_1,\ x_2(0) = 0$$

whose solution is the following function:

In[400]:=
```
ClearAll[x, t]
x[t_] = {Cos[t], Sin[t]}
```

Out[401]=
```
{Cos[t], Sin[t]}
```

Define the vector-valued function f as follows:

```
In[402]:= ClearAll[f, t, u]
          f[t_, u_] := {-u[[2]], u[[1]]}
```

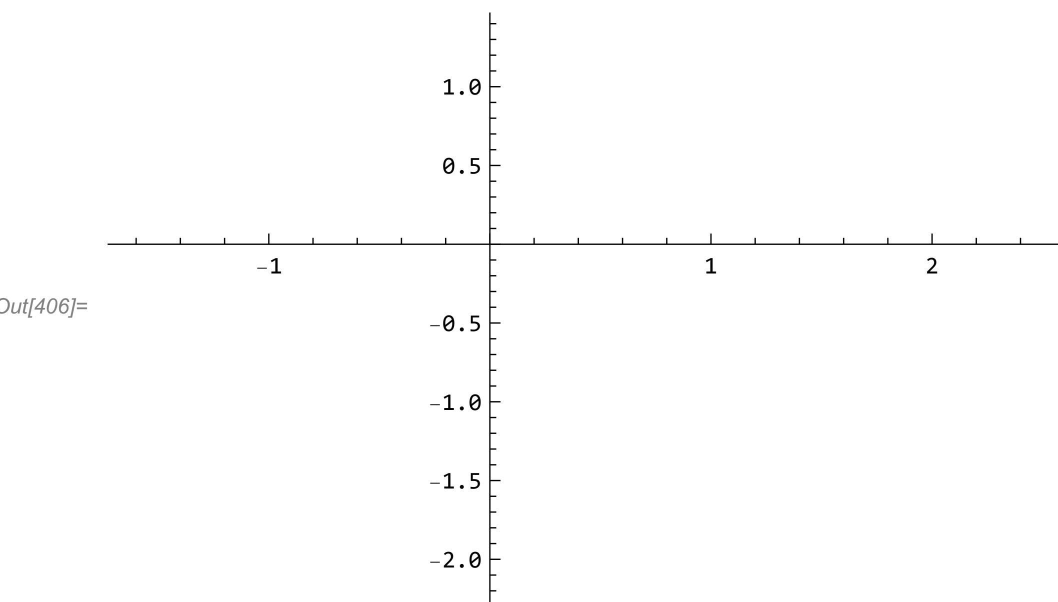
Now we can apply Euler's method:

```
In[404]:= r1 = euler[f, 0., 2. * Pi, {1, 0}, 20];
          TableForm[r1]
```

Out[405]//TableForm=

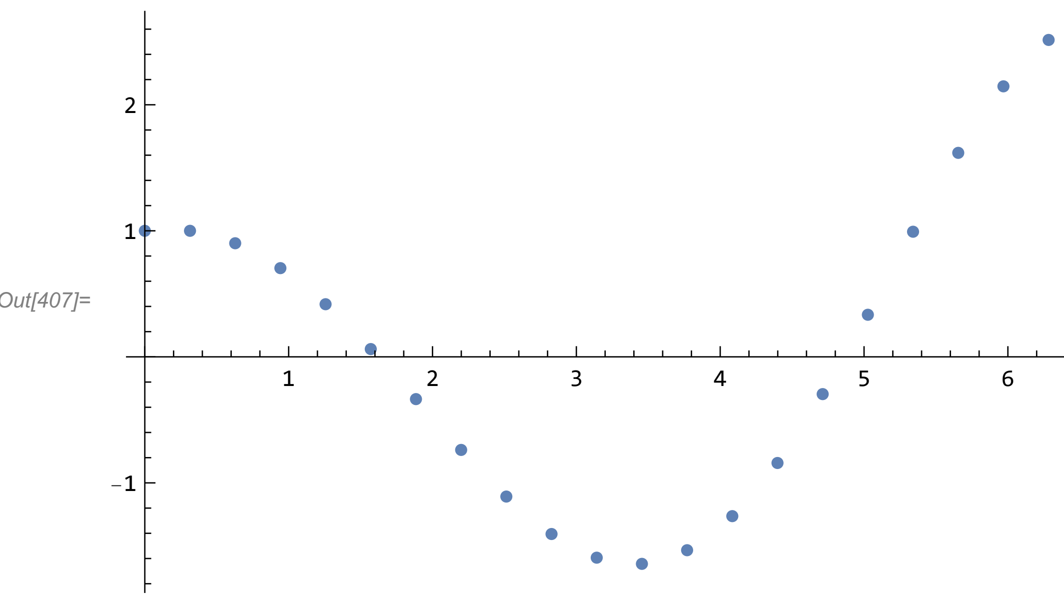
```
0.          1
            0
0.314159    1.
            0.314159
0.628319    0.901304
            0.628319
0.942478    0.703912
            0.911472
1.25664     0.417565
            1.13261
1.5708      0.0617441
            1.26379
1.88496     -0.335288
            1.28319
2.19911     -0.738415
            1.17786
2.51327     -1.10845
            0.945877
2.82743     -1.40561
            0.597648
3.14159     -1.59336
            0.156064
3.45575     -1.64239
            -0.344506
3.76991     -1.53416
            -0.860478
4.08407     -1.26383
            -1.34245
4.39823     -0.842091
            -1.73949
4.71239     -0.295613
            -2.00405
5.02655     0.333976
            -2.09691
5.34071     0.992742
            -1.99199
5.65487     1.61854
            -1.68011
5.96903     2.14637
            -1.17163
6.28319     2.51445
            -0.497332
```

Working with the output is a little more difficult, since **ListPlot**, for example, will not handle data in which the second coordinate is a vector:



To plot the first component of the solution, for example, we have to create a list of data points of the form $(t_i,(x_1)_i)$:

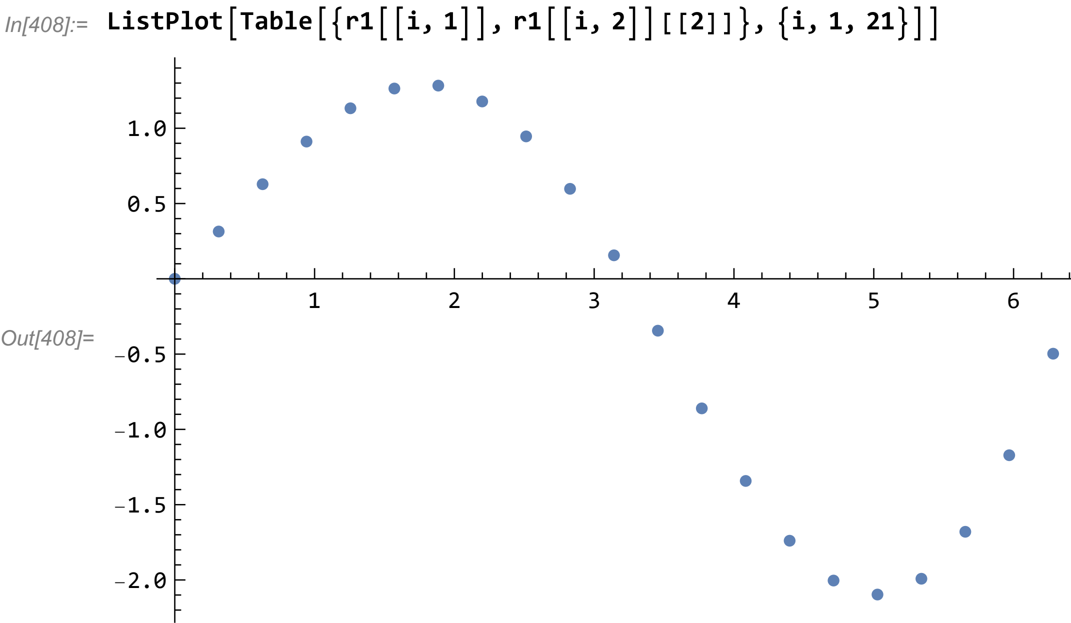
```
In[407]:= ListPlot[Table[{r1[[i, 1]], r1[[i, 2]][[1]]}, {i, 1, 21}]]
```



A similar command will graph the second component:

In[408]:=
```
ListPlot[Table[{r1[[i, 1]], r1[[i, 2]][[2]]}, {i, 1, 21}]]
```

Out[408]=



There is a serious shortcoming to the program we have written: the symbols used by the **euler** command are not "local". For example, **dt** now has the value it was assigned during the last execution of **euler**:

In[409]:=
```
dt
```

Out[409]=
```
0.314159
```

Obviously this is undesirable, since the use of a function like **euler** can change the value of symbols that are in use outside the scope of the function. *Mathematica* provides a way to avoid this type of behavior. The **Module** command creates a block of code with local variables. The command **Module[{vars},expr]** evaluates the expression *expr*, while regarding the variables *vars* as local to the scope of the **Module** command. Here is the Euler function rewritten to use local variables:

In[410]:=
```
ClearAll[euler]
euler[f_, t0_, T_, u0_, n_] :=
 Module[{U, i, dt}, U = Table[{Null, Null}, {i, 0, n}];
  U[[1, 1]] = t0;
  U[[1, 2]] = u0;
  dt = (T - t0) / n;
  Do[
   U[[i + 1, 1]] = U[[i, 1]] + dt;
   U[[i + 1, 2]] = U[[i, 2]] + dt * f[U[[i, 1]], U[[i, 2]]],
   {i, 1, n}];
  U]
```

Now we have the desired behavior:

```
In[412]:= ClearAll[dt]
          r1 = euler[f, 0., 2. * Pi, {1, 0}, 10];
          dt
Out[414]= dt
```

This shows that **dt** has no value after the execution of **euler**.

Having decided that a certain algorithm is useful enough to be made into a *Mathematica* command, you will probably wish to save it in a file, so that it can be loaded (rather than typed anew) whenever it is needed in a new *Mathematica* session. Doing this is simple enough; the *Mathematica* commands defining the function are simply typed into a plain text file and read into *Mathematica* using the << operator.

To illustrate this, we wrote a file called euler, containing the definition of **euler** (just as it was previously typed it into *Mathematica*). There is an important detail about using an external file like this: *Mathematica* must be able to find it. The simplest way to ensure that *Mathematica* can find a file is to make sure the file is in the current working directory. The **Directory** command tells the current working directory:

```
In[415]:= Directory[]
Out[415]= /Users/msgocken
```

If this is not the desired directory, **SetDirectory** allows us to switch to a different working directory:

```
In[416]:= SetDirectory["/Users/msgocken/books/pdebook2/tutorial/mathematica"]
Out[416]= /Users/msgocken/books/pdebook2/tutorial/mathematica
```

The **FileNames** command lists the files in the current directory:

```
In[417]:= FileNames[]
Out[417]= {euler, fempack, fempack~, mathtut2.nb, mathtut2.pdf, mathtut2.tar.gz}
```

The file euler is there, so I can load it now. (I will first clear **euler** just to show that the << operator really does retrieve the definition from the file):

```
In[418]:= ClearAll[euler]
          ? euler

          Global`euler

In[420]:= << euler
```

In[421]:=
```
? euler
```

Global`euler

```
euler[f_, t0_, T_, u0_, n_] := Module[{U, i, h}, U = Table[{Null, Null}, {i, 0, n}];
  U[[1, 1]] = t0;
  U[[1, 2]] = u0;
  h = (T-t0)/n;
  Do[U[[i + 1, 1]] = U[[i, 1]] + h;
   U[[i + 1, 2]] = U[[i, 2]] + h f[U[[i, 1]], U[[i, 2]]], {i, 1, n}];
  U]
```

The **FilePrint** command lists the contents of a text file. This operator may be useful, for example, to check that one really wants to load a given file.

In[422]:=
```
FilePrint["euler"]
```

```
euler[f_, t0_, T_, u0_, n_] :=

  Module[{U, i, h}, U = Table[{Null, Null}, {i, 0, n}];
    U[[1, 1]] = t0;
    U[[1, 2]] = u0;
    h = (T - t0)/n;
    Do[
      U[[i + 1, 1]] = U[[i, 1]] + h;
      U[[i + 1, 2]] =
        U[[i, 2]] + h*f[U[[i, 1]], U[[i, 2]]],
      {i, 1, n}];
    U]
```

Having loaded the definition of **euler**, we can now use it as before.

If euler had not been found in the current working directory, we could proceed in three ways. One option would be to move the file euler into the current directory (using an operating system command). The second would be to change the working directory using the **SetDirectory** command, as illustrated above. The final option is to give a full path name so that *Mathematica* can find the file.

In[423]:=
```
ClearAll[euler]
? euler
```

Global`euler

In[425]:=
```
<< /Users/msgocken/books/pdebook2/tutorial/mathematica/euler
```

In[426]:=
```
? euler
```

Global`euler

```
euler[f_, t0_, T_, u0_, n_] := Module[{U, i, h}, U = Table[{Null, Null}, {i, 0, n}];
  U⟦1, 1⟧ = t0;
  U⟦1, 2⟧ = u0;
  h = (T-t0)/n;
  Do[U⟦i + 1, 1⟧ = U⟦i, 1⟧ + h;
   U⟦i + 1, 2⟧ = U⟦i, 2⟧ + h f[U⟦i, 1⟧, U⟦i, 2⟧], {i, 1, n}];
  U]
```

Chapter 5: Boundary value problems in statics

■ Section 5.2: Introduction to the spectral method; eigenfunctions

We begin this section by verifying that the eigenfunctions of the negative second derivative operator (under Dirichlet conditions), $\sin(n\pi x/l)$, l=1,2,3,..., are mutually orthogonal:

In[427]:=
```
ClearAll[m, n, l, x]
Integrate[Sin[n Pi x / l] Sin[m Pi x / l], {x, 0, l}]
```

Out[428]= $\frac{l\, n \cos[n\pi] \sin[m\pi] - l\, m \cos[m\pi] \sin[n\pi]}{m^2 \pi - n^2 \pi}$

In[429]:=
```
Simplify[%]
```

Out[429]= $\frac{l\, n \cos[n\pi] \sin[m\pi] - l\, m \cos[m\pi] \sin[n\pi]}{m^2 \pi - n^2 \pi}$

At first glance, this result is surprising: Why did *Mathematica* 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 positive integers, and *Mathematica* has no way of knowing that the symbols m and n are intended to represent integers. Fortunately, there is a way to tell *Mathematica*:

In[430]:=
```
Simplify[%, Element[{m, n}, Integers]]
```

Out[430]= 0

When performing Fourier series calculations, the above command is very useful. However, it would be tedious to repeatedly specify the assumption that m and n are integers. We can specify such an assumption once and for all using the global variable **$Assumptions:**

In[431]:=
```
$Assumptions = Element[{m, n}, Integers]
```
Out[431]=
```
(m | n) ∈ Integers
```

Now *m* and *n* will be treated as integers, whereas another variable, such as *k*, will not:

In[432]:=
```
Simplify[Sin[m Pi]]
Simplify[Sin[n Pi]]
Simplify[Sin[k Pi]]
```
Out[432]= 0

Out[433]= 0

Out[434]= Sin[k π]

Example 5.5

Let

In[435]:=
```
ClearAll[f, x]
f[x_] = x (1 - x)
```
Out[436]= (1 - x) x

We can easily compute the Fourier sine coefficients of *f* on the interval [0,1].   For convenience, we define a function (of *n*) representing the coefficient:

In[437]:=
```
ClearAll[a, n]
a[n_] = 2 Integrate[f[x] Sin[n Pi x], {x, 0, 1}]
```
Out[438]= $-\frac{4\,(-1+(-1)^n)}{n^3\,\pi^3}$

Mathematica can represent finite sums using the **Sum** command. The following trick is useful: since we often wish to experiment with the number of terms used in a partial Fourier series, we define the number of terms to be an input variable:

In[439]:=
```
ClearAll[s, x, M]
s[x_, M_] := Sum[a[n] Sin[n Pi x], {n, 1, M}]
```

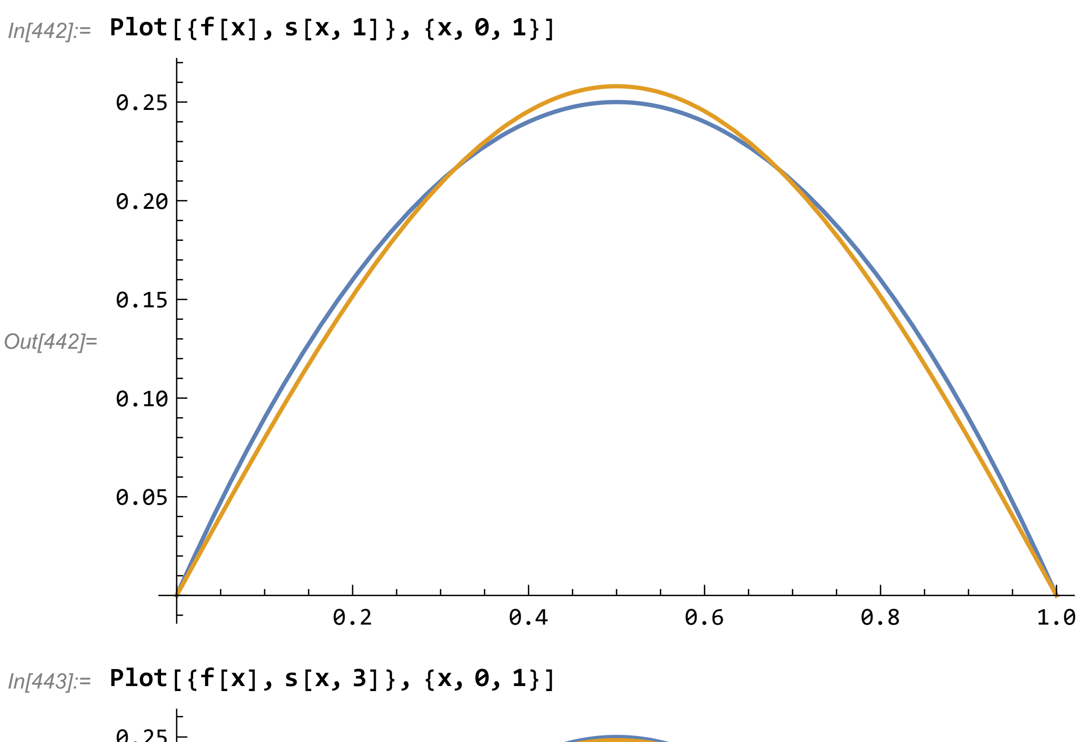
(Notice the use of the delayed-evaluation assignment operator ":=".) The function becomes a function of *x* alone when an integer is entered for *M*:

In[441]:=
```
s[x, 5]
```
Out[441]= $\frac{8\,\mathrm{Sin}[\pi\,x]}{\pi^3}+\frac{8\,\mathrm{Sin}[3\,\pi\,x]}{27\,\pi^3}+\frac{8\,\mathrm{Sin}[5\,\pi\,x]}{125\,\pi^3}$

We can now see how well the partial Fourier series approximates *f* by looking at some graphs:

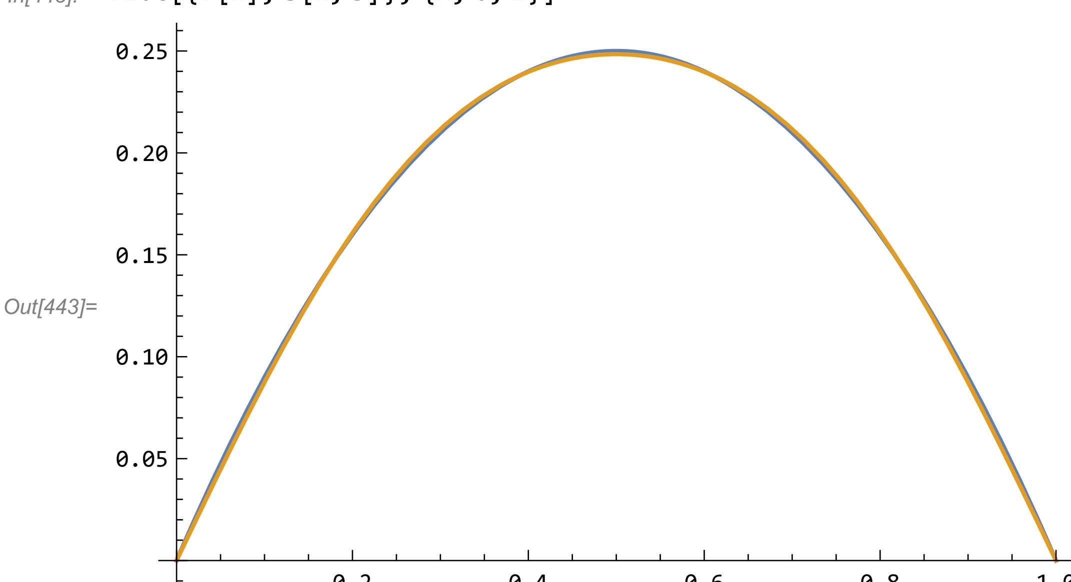
In[442]:= `Plot[{f[x], s[x, 1]}, {x, 0, 1}]`

Out[442]=



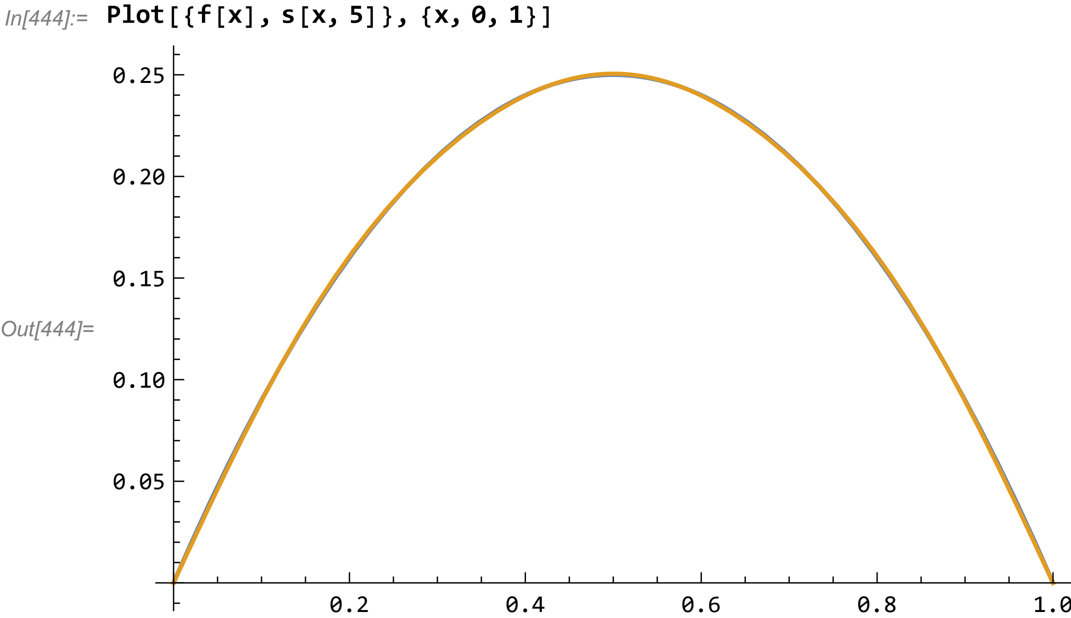
In[443]:= `Plot[{f[x], s[x, 3]}, {x, 0, 1}]`

Out[443]=



In[444]:=
```
Plot[{f[x], s[x, 5]}, {x, 0, 1}]
```

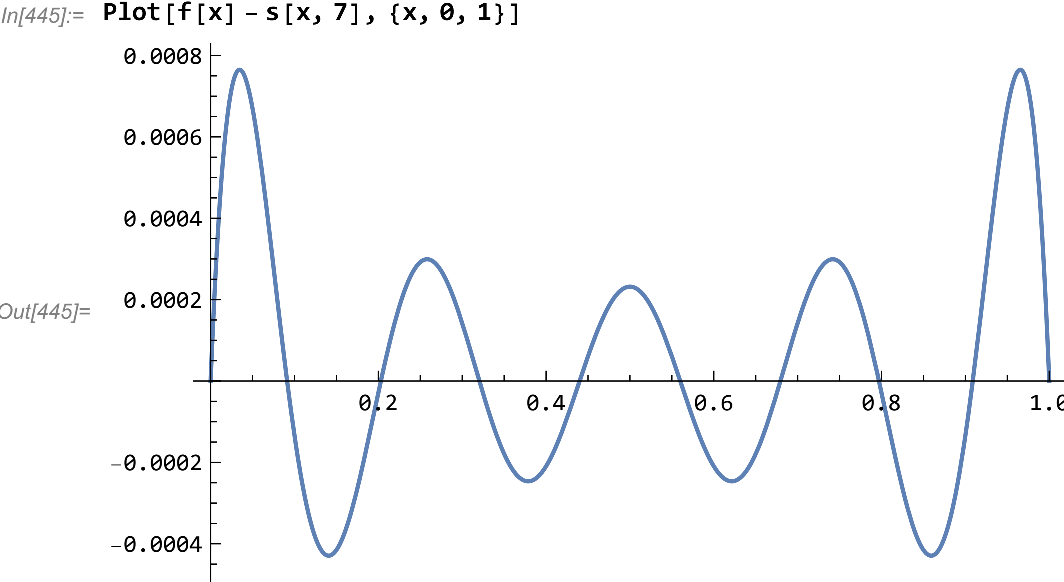
Out[444]=



Once the approximation becomes very accurate, it is more informative to plot the error:

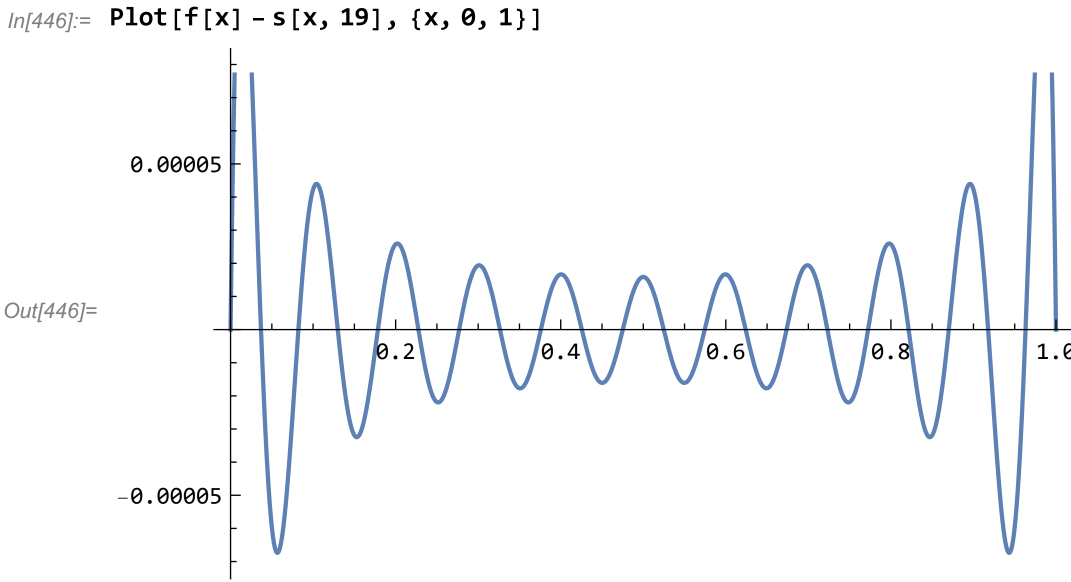
In[445]:=
```
Plot[f[x] - s[x, 7], {x, 0, 1}]
```

Out[445]=



In[446]:=
```
Plot[f[x] - s[x, 19], {x, 0, 1}]
```

Out[446]=



■ Section 5.5: The Galerkin method

We now show how to apply the Galerkin method with a polynomial basis. Suppose we wish to approximate the solution to the BVP

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

using the subspace of $C_D^2[0, 1]$ spanned by the following four polynomials:

In[447]:=
```
ClearAll[p, x]
p[1][x_] = x (1 - x)
p[2][x_] = x (1 / 2 - x) (1 - x)
p[3][x_] = x (1 / 3 - x) (2 / 3 - x) (1 - x)
p[4][x_] = x (1 / 4 - x) (1 / 2 - x) (3 / 4 - x) (1 - x)
```

Out[448]= $(1-x)\,x$

Out[449]= $\left(\frac{1}{2}-x\right)(1-x)\,x$

Out[450]= $\left(\frac{1}{3}-x\right)\left(\frac{2}{3}-x\right)(1-x)\,x$

Out[451]= $\left(\frac{1}{4}-x\right)\left(\frac{1}{2}-x\right)\left(\frac{3}{4}-x\right)(1-x)\,x$

The energy inner product is defined as follows:

In[452]:=
```
ClearAll[a, u, v]
a[u_, v_] := Integrate[(1 + x) D[u[x], x] × D[v[x], x], {x, 0, 1}]
```

The L^2 inner product is defined as

In[454]:=
```
ClearAll[L, u, v]
L[u_, v_] := Integrate[u[x] × v[x], {x, 0, 1}]
```

Now the calculation is simple in principle (but it would be very tedious to carry out by hand): We just need to compute the stiffness matrix and the load vector, and solve the linear system. The stiffness matrix is

In[456]:=
```
ClearAll[K]
K = Table[a[p[i], p[j]], {i, 1, 4}, {j, 1, 4}]
```

Out[457]= $\left\{\left\{\frac{1}{2}, -\frac{1}{30}, \frac{1}{90}, -\frac{1}{672}\right\}, \left\{-\frac{1}{30}, \frac{3}{40}, -\frac{19}{3780}, \frac{3}{896}\right\}, \left\{\frac{1}{90}, -\frac{19}{3780}, \frac{5}{567}, -\frac{41}{60480}\right\}, \left\{-\frac{1}{672}, \frac{3}{896}, -\frac{41}{60480}, \frac{43}{43008}\right\}\right\}$

Let the right-hand side be $g(x) = x^2$:

In[458]:=
```
ClearAll[g, x]
g[x_] = x^2
```

Out[459]= x^2

The load vector is

In[460]:=
```
ClearAll[f]
f = Table[L[p[i], g], {i, 1, 4}]
```

Out[461]= $\left\{\frac{1}{20}, -\frac{1}{120}, \frac{1}{630}, -\frac{1}{2688}\right\}$

Then the coefficients defining the (approximate) solution are

In[462]:=
```
ClearAll[c]
c = LinearSolve[K, f]
```

Out[463]= $\left\{\frac{3325}{34997}, -\frac{9507}{139988}, \frac{1575}{69994}, \frac{420}{34997}\right\}$

and the approximate solution is

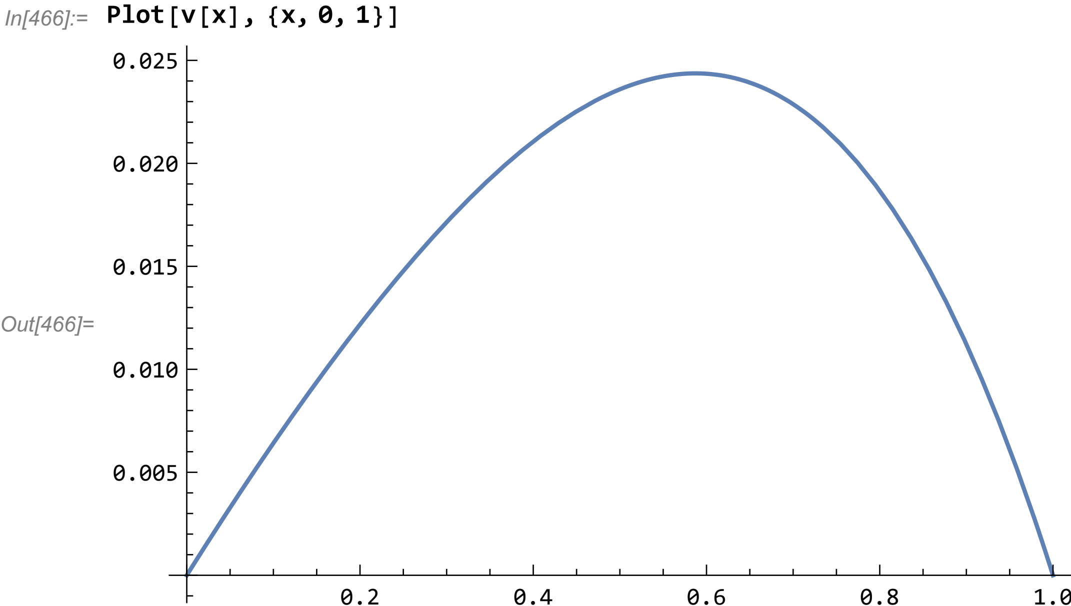
In[464]:=
```
ClearAll[v, x]
v[x_] = Sum[c[[j]] × p[j][x], {j, 1, 4}]
```

Out[465]= $$\frac{3325\,(1-x)\,x}{34997} - \frac{9507\left(\frac{1}{2}-x\right)(1-x)\,x}{139988} + \frac{1575\left(\frac{1}{3}-x\right)\left(\frac{2}{3}-x\right)(1-x)\,x}{69994} + \frac{420\left(\frac{1}{4}-x\right)\left(\frac{1}{2}-x\right)\left(\frac{3}{4}-x\right)(1-x)\,x}{34997}$$

Here is a graph:

In[466]:=
```
Plot[v[x], {x, 0, 1}]
```

Out[466]=



The exact solution can be found by integration:

In[467]:=
```
ClearAll[c1, s, x]
Integrate[-s^2, {s, 0, x}] + c1
```

Out[468]= $c1 - \frac{x^3}{3}$

In[469]:=
```
u[x_] = Simplify[Integrate[(c1 - s^3 / 3) / (1 + s), {s, 0, x}]]
```

Out[469]= ConditionalExpression$\left[\frac{1}{18}\,(x\,(-6+(3-2\,x)\,x)+6\,(1+3\,c1)\,\mathrm{Log}[1+x]),\ \mathrm{Im}[x]\neq 0\,||\,\mathrm{Re}[x]>-1\right]$

The last result is unnecessarily complicated (because *Mathematica* does not know that x is a real number); let us repeat the calculation with the assumption that $x > 0$:

In[470]:=
```
u[x_] = Simplify[Integrate[(c1 - s^3 / 3) / (1 + s), {s, 0, x}], x > 0]
```

Out[470]= $\frac{1}{18}\,(x\,(-6+(3-2\,x)\,x)+6\,(1+3\,c1)\,\mathrm{Log}[1+x])$

In[471]:=
```
ClearAll[sols]
sols = Solve[u[1] == 0, c1]
```

Out[472]= $\left\{\left\{c1 \to \frac{5-6\,\mathrm{Log}[2]}{18\,\mathrm{Log}[2]}\right\}\right\}$

In[473]:=
```
c1 = (c1 /. sols[[1]])
```

Out[473]= $\frac{5-6\,\mathrm{Log}[2]}{18\,\mathrm{Log}[2]}$

Check:

In[474]:= `-D[(1 + x) D[u[x], x], x]`

Out[474]= $-\frac{1}{18}(1+x)\left(6-12x-\frac{6\left(1+\frac{5-6\,\text{Log}[2]}{6\,\text{Log}[2]}\right)}{(1+x)^2}\right)+\frac{1}{18}\left(6-(3-4x)\,x-(3-2x)\,x-\frac{6\left(1+\frac{5-6\,\text{Log}[2]}{6\,\text{Log}[2]}\right)}{1+x}\right)$

In[475]:= `Simplify[%]`

Out[475]= x^2

In[476]:= `u[0]`

Out[476]= 0

In[477]:= `u[1]`

Out[477]= $\frac{1}{18}\left(-5+6\left(1+\frac{5-6\,\text{Log}[2]}{6\,\text{Log}[2]}\right)\text{Log}[2]\right)$

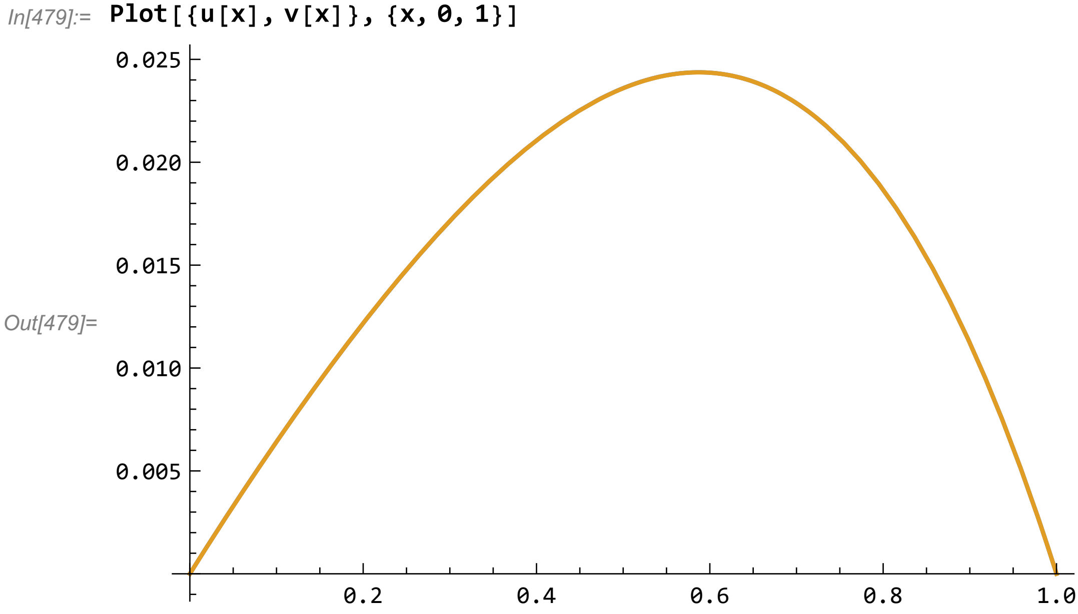
In[478]:= `Simplify[u[1]]`

Out[478]= 0

Now we can compare the exact and approximate solutions:

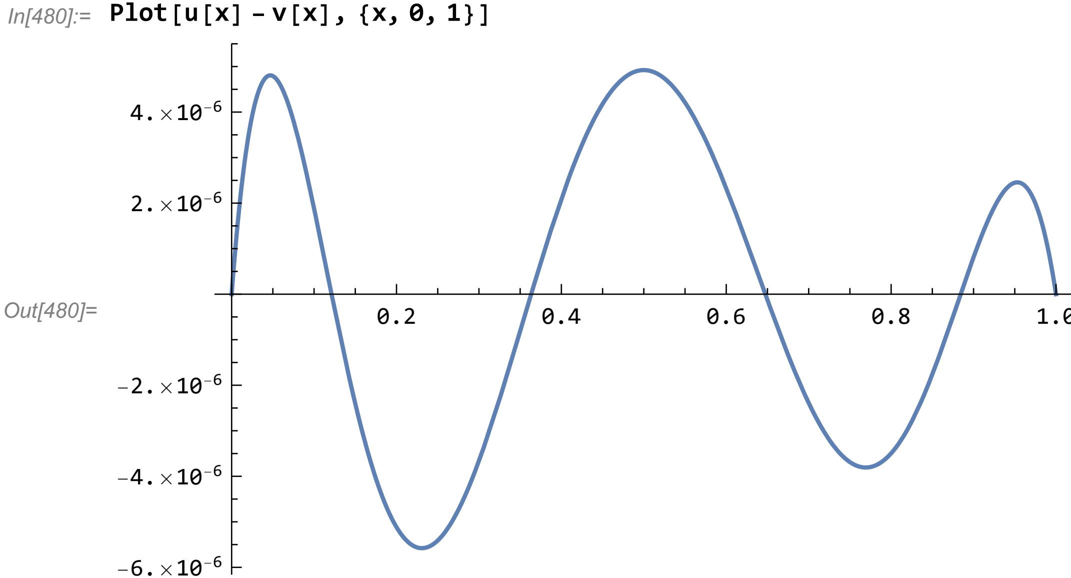
In[479]:= `Plot[{u[x], v[x]}, {x, 0, 1}]`

Out[479]=



In[480]:=
```
Plot[u[x] - v[x], {x, 0, 1}]
```

Out[480]=



The computed solution is highly accurate!

Section 5.6: Piecewise polynomials and the finite element method

Representing piecewise linear functions

A useful feature of *Mathematica* for working with piecewise linear functions is the **Interpolation** command. Since a continuous piecewise linear function has values defined by linear interpolation, it is possible to define a function from the nodes and nodal values. Interpolation was introduced before, in Section 4.4, but we will now explain it in more detail.

For example, consider the following data (which define a piecewise linear interpolant of $\sin(\pi x)$ on the interval [0,1]):

In[481]:=
```
ClearAll[v, i]
v = Table[{i/5, N[Sin[Pi i/5]]}, {i, 0, 5}];
```

In[483]:=
```
TableForm[v]
```

Out[483]//TableForm=

0	0.
$\frac{1}{5}$	0.587785
$\frac{2}{5}$	0.951057
$\frac{3}{5}$	0.951057
$\frac{4}{5}$	0.587785
1	0.

The following command creates the continuous piecewise linear function with these nodal values. (Notice the use of the **InterpolationOrder** option. This is necessary to cause *Mathematica* to use *linear* interpolation.)

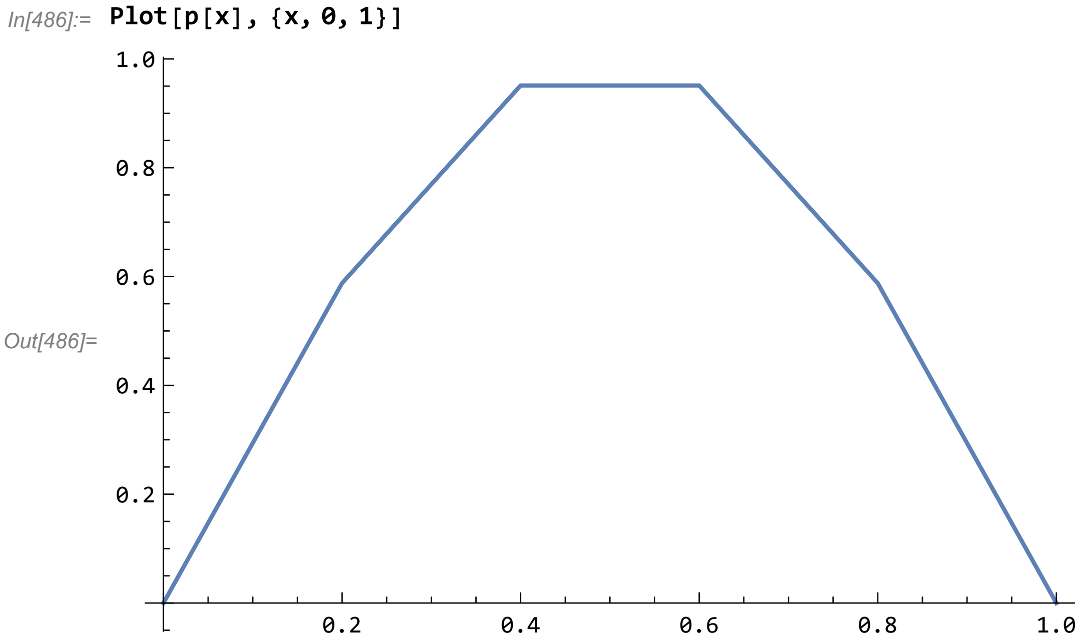
In[484]:=
```
ClearAll[p]
p = Interpolation[v, InterpolationOrder → 1]
```

Out[485]= InterpolatingFunction[Domain: {{0., 1.}} Output: scalar]

We can now plot the piecewise linear function:

In[486]:=
```
Plot[p[x], {x, 0, 1}]
```

Out[486]=



We can also compare it to the function it interpolates:

In[487]:=
```
Plot[{Sin[Pi x], p[x]}, {x, 0, 1}]
```

Out[487]=

1.0
0.8
0.6
0.4
0.2
0.2 0.4 0.6 0.8 1.0

We can use interpolating functions to represent the standard piecewise linear basis functions:

In[488]:=
```
ClearAll[phi, i, n]
phi[i_, n_] :=
 Interpolation[Table[{j/n, If[i == j, 1, 0]}, {j, 0, n}], InterpolationOrder → 1]
```

In[490]:=
```
Plot[phi[3, 10][x], {x, 0, 1}, PlotRange → All]
```

Out[490]=

1.0
0.8
0.6
0.4
0.2
0.2 0.4 0.6 0.8 1.0

However, although **InterpolatingFunctions** are useful for some purposes, such as graphing piecewise linear functions, *Mathematica* is not able to perform some needed operations on them. For example, suppose we want to apply the finite element method, with piecewise linear functions, to the following BVP:

$$-\frac{d}{dx}[k(x)\frac{du}{dx}] = x, \ \ 0 < x < 1,$$
$$u(0) = u(1) = 0.$$

Using a regular mesh with 10 subintervals, we would need to compute the following integrals:

$$\int_0^1 \phi_i(x)\, x\, dx, \ \ i = 1, 2, \ \ldots, 10$$

Mathematica cannot do this directly:

In[491]:=
```
Integrate[phi[i, 10][x] * x, {x, 0, 1}]
```

Out[491]=
$$\int_0^1 x \, \text{InterpolatingFunction}\left[\boxed{\text{Domain: } \{\{0, 1\}\} \ \ \text{Output: scalar}}\right][x] \, dx$$

In[492]:=
```
Integrate[phi[1, 10][x] * x, {x, 0, 1}]
```

Out[492]=
$$\int_0^1 x\,\text{InterpolatingFunction}\left[\begin{array}{l}\text{Domain: }\{\{0, 1\}\}\\ \text{Output: scalar}\end{array}\right][x]\,dx$$

(The definition of the interpolating function is just too complicated for *Mathematica* to integrate automatically.)

The best way to proceed is to recognize that each ϕ_i is represented by simple formulas on the two subintervals that form its support. We define these two formulas in general:

In[493]:=
```
ClearAll[phi1, phi2, x, i, n]
phi1[x_, i_, n_] = (x - (i - 1) / n) / (1 / n)
phi2[x_, i_, n_] = - (x - (i + 1) / n) / (1 / n)
```

Out[494]= $n\left(-\frac{-1+i}{n}+x\right)$

Out[495]= $n\left(\frac{1+i}{n}-x\right)$

The function **phi1** represents ϕ_i on the subinterval $[x_{i-1}, x_i]$, while **phi2** represents ϕ_i on the subinterval $[x_i, x_{i+1}]$.

we can now compute the load vector for the finite element method by recognizing that

$$\int_0^1 \phi_i(x)\,x\,dx = \int_{x_{i-1}}^{x_i} \phi_i(x)\,x\,dx + \int_{x_i}^{x_{i+1}} \phi_i(x)\,x\,dx.$$

Here are two different ways to create the load vector. First, create a vector of all zeros:

In[496]:=
```
ClearAll[n, h, f]
n = 10
h = 1 / n
f = Table[0, {i, 1, n - 1}]
```

Out[497]= 10

Out[498]= $\frac{1}{10}$

Out[499]= {0, 0, 0, 0, 0, 0, 0, 0, 0}

Then compute the components using a loop:

In[500]:=
```
ClearAll[i]
Do[f[[i]] = Integrate[phi1[x, i, n] x, {x, i * h - h, i * h}] +
   Integrate[phi2[x, i, n] x, {x, i * h, i * h + h}], {i, 1, n - 1}]
```

In[502]:=
```
f
```

Out[502]= $\left\{\frac{1}{100}, \frac{1}{50}, \frac{3}{100}, \frac{1}{25}, \frac{1}{20}, \frac{3}{50}, \frac{7}{100}, \frac{2}{25}, \frac{9}{100}\right\}$

We can accomplish the same thing in a single **Table** command:

In[503]:=
```
ClearAll[f, i]
f = Table[Integrate[phi1[x, i, n] x, {x, i*h - h, i*h}] +
   Integrate[phi2[x, i, n] x, {x, i*h, i*h + h}], {i, 1, n - 1}]
```

Out[504]= $\left\{\frac{1}{100}, \frac{1}{50}, \frac{3}{100}, \frac{1}{25}, \frac{1}{20}, \frac{3}{50}, \frac{7}{100}, \frac{2}{25}, \frac{9}{100}\right\}$

We can use similar commands to create the stiffness matrix K. For the first example, we choose the constant coefficient $k(x)$=1, because we already know what the result should be (see Section 5.6.1 of the text).

In[505]:=
```
ClearAll[k, x]
k[x_] = 1
```

Out[506]= 1

Since most of the entries of K are zero, we define a zero matrix, and then fill in the nonzero entries:

In[507]:=
```
ClearAll[K, i, j]
K = Table[0.0, {i, 1, n - 1}, {j, 1, n - 1}]
```

Out[508]=
```
{{0., 0., 0., 0., 0., 0., 0., 0., 0.},
 {0., 0., 0., 0., 0., 0., 0., 0., 0.}, {0., 0., 0., 0., 0., 0., 0., 0., 0.},
 {0., 0., 0., 0., 0., 0., 0., 0., 0.}, {0., 0., 0., 0., 0., 0., 0., 0., 0.},
 {0., 0., 0., 0., 0., 0., 0., 0., 0.}, {0., 0., 0., 0., 0., 0., 0., 0., 0.},
 {0., 0., 0., 0., 0., 0., 0., 0., 0.}, {0., 0., 0., 0., 0., 0., 0., 0., 0.}}
```

The following loop computes the diagonal entries:

In[509]:=
```
Do[K[[i, i]] = Integrate[k[x] × D[phi1[x, i, n], x]^2, {x, i*h - h, i*h}] +
   Integrate[k[x] × D[phi2[x, i, n], x]^2, {x, i*h, i*h + h}], {i, 1, n - 1}]
```

In[510]:=
```
MatrixForm[K]
```

Out[510]//MatrixForm=

$$\begin{pmatrix} 20 & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. \\ 0. & 20 & 0. & 0. & 0. & 0. & 0. & 0. & 0. \\ 0. & 0. & 20 & 0. & 0. & 0. & 0. & 0. & 0. \\ 0. & 0. & 0. & 20 & 0. & 0. & 0. & 0. & 0. \\ 0. & 0. & 0. & 0. & 20 & 0. & 0. & 0. & 0. \\ 0. & 0. & 0. & 0. & 0. & 20 & 0. & 0. & 0. \\ 0. & 0. & 0. & 0. & 0. & 0. & 20 & 0. & 0. \\ 0. & 0. & 0. & 0. & 0. & 0. & 0. & 20 & 0. \\ 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. & 20 \end{pmatrix}$$

(Note that 20=2/h, which is the expected result.) Now for the off-diagonal entries:

In[511]:=
```
Do[K[[i, i + 1]] =
   Integrate[k[x] × D[phi2[x, i, n], x] × D[phi1[x, i + 1, n], x], {x, i*h, i*h + h}];
  K[[i + 1, i]] = K[[i, i + 1]],
  {i, 1, n - 2}]
```

You should notice several things:

The matrix K is an (n-1) by (n-1) matrix, with (n-2) entries along the first subdiagonal and superdiagonal. This explains why the loop goes from i=1 to i=n-2.

The matrix K is symmetric, so we just compute **K[[i,i+1]]** and assigned the value to **K[[i+1,i]]**.

Over the interval $[x_i, x_{i+1}]$, which is the common support of ϕ_i and ϕ_{i+1}, we use the function **phi2** to represent ϕ_i and **phi1** to represent ϕ_{i+1}. This is because the interval $[x_i, x_{i+1}]$ supports the right half of (the nonzero part of) ϕ_i and the left half of (the nonzero part of) ϕ_{i+1}.

Here is the result; notice that the off-diagonal entries are

$$-\frac{1}{h} = -10,$$

as expected.

```
In[512]:= MatrixForm[K]
```

Out[512]//MatrixForm=

$$\begin{pmatrix} 20 & -10 & 0. & 0. & 0. & 0. & 0. & 0. & 0. \\ -10 & 20 & -10 & 0. & 0. & 0. & 0. & 0. & 0. \\ 0. & -10 & 20 & -10 & 0. & 0. & 0. & 0. & 0. \\ 0. & 0. & -10 & 20 & -10 & 0. & 0. & 0. & 0. \\ 0. & 0. & 0. & -10 & 20 & -10 & 0. & 0. & 0. \\ 0. & 0. & 0. & 0. & -10 & 20 & -10 & 0. & 0. \\ 0. & 0. & 0. & 0. & 0. & -10 & 20 & -10 & 0. \\ 0. & 0. & 0. & 0. & 0. & 0. & -10 & 20 & -10 \\ 0. & 0. & 0. & 0. & 0. & 0. & 0. & -10 & 20 \end{pmatrix}$$

We complete the solution process by solving the equation $Ku=f$:

```
In[513]:= ClearAll[u]
          u = LinearSolve[K, f]
Out[514]= {0.0165, 0.032, 0.0455, 0.056, 0.0625, 0.064, 0.0595, 0.048, 0.0285}
```

Whether we wish to graph the nodal values or create the piecewise linear function defined by them (using the **Interpolation** command), we need to create a table containing the data pairs (x_i, u_i). First, we append the boundary value 0 to each end of the vector u. This requires the **Flatten** command, which "flattens" a nested list:

```
In[515]:= u = Flatten[{0, u, 0}]
Out[515]= {0, 0.0165, 0.032, 0.0455, 0.056, 0.0625, 0.064, 0.0595, 0.048, 0.0285, 0}
```

Next, we create the x-values defining the mesh:

```
In[516]:= ClearAll[X, i]
          X = Table[i * h, {i, 0, n}]
```

Out[517]= $\left\{0, \frac{1}{10}, \frac{1}{5}, \frac{3}{10}, \frac{2}{5}, \frac{1}{2}, \frac{3}{5}, \frac{7}{10}, \frac{4}{5}, \frac{9}{10}, 1\right\}$

Finally, we make the table of data points:

```
In[518]:= ClearAll[U]
          U = Transpose[{X, u}];
```

In[520]:=
```
MatrixForm[U]
```

Out[520]//MatrixForm=

$$\begin{pmatrix} 0 & 0 \\ \frac{1}{10} & 0.0165 \\ \frac{1}{5} & 0.032 \\ \frac{3}{10} & 0.0455 \\ \frac{2}{5} & 0.056 \\ \frac{1}{2} & 0.0625 \\ \frac{3}{5} & 0.064 \\ \frac{7}{10} & 0.0595 \\ \frac{4}{5} & 0.048 \\ \frac{9}{10} & 0.0285 \\ 1 & 0 \end{pmatrix}$$

Now we can use the **ListPlot** command:

In[521]:=
```
ListPlot[U,
 PlotStyle → PointSize[0.02]]
```

Out[521]=

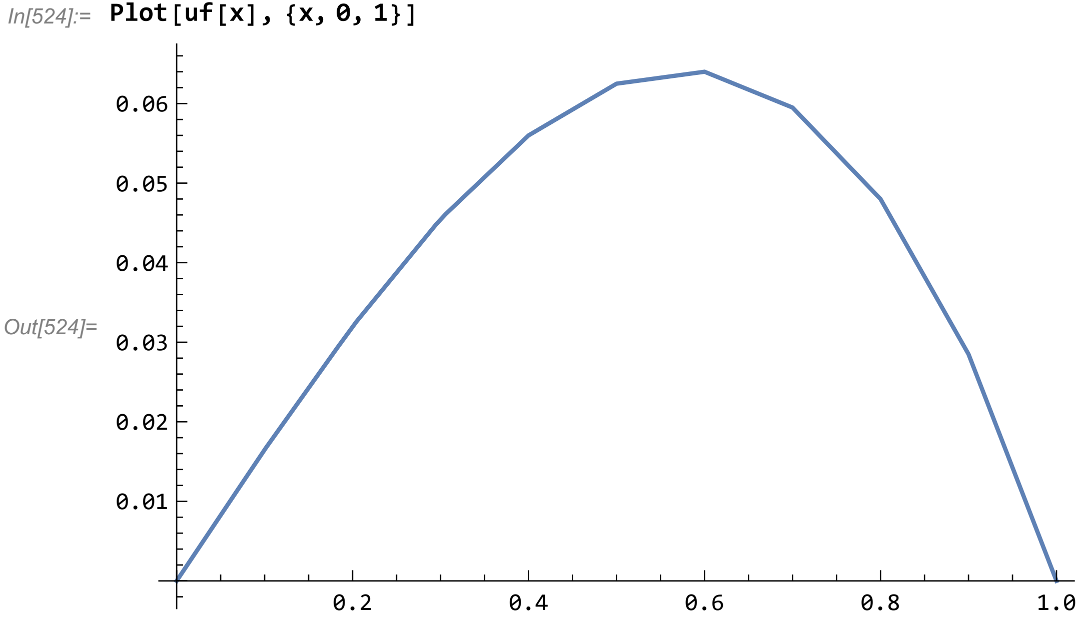
We can also create an interpolating function and use the **Plot** command. We want to use linear interpolation, so we give the option **InterpolationOrder->1**:

In[522]:=
```
ClearAll[uf]
uf = Interpolation[U, InterpolationOrder → 1]
```

Out[523]= InterpolatingFunction[Domain: {{0., 1.}} Output: scalar]

In[524]:= `Plot[uf[x], {x, 0, 1}]`

Out[524]=



An advantage of creating an interpolating function is that we can compare to the exact function, when we know it. In the above example, the exact solution is $v(x) = (x - x^3)/6$:

In[525]:= `ClearAll[v, x]`
`v[x_] = (x - x^3) / 6`

Out[526]= $\frac{1}{6}\left(x - x^3\right)$

Here is the error in the finite element solution:

In[527]:= `Plot[uf[x] - v[x], {x, 0, 1}]`

Out[527]=

0.2 0.4 0.6 0.8 1.0
-0.0002
-0.0004
-0.0006
-0.0008
-0.0010
-0.0012

A nonconstant coefficient example

Now suppose we wish to solve the BVP

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

where

```
In[528]:= ClearAll[k, x]
          k[x_] = 1 + x
Out[529]= 1 + x
```

We use the same mesh; then the load vector is exactly the same as before and need not be recomputed. The stiffness matrix can be computed just as before:

```
In[530]:= ClearAll[K, i, j]
          K = Table[0.0, {i, 1, n - 1}, {j, 1, n - 1}];
```

The following loop computes the diagonal entries:

```
In[532]:= Do[K[[i, i]] = Integrate[k[x] × D[phi1[x, i, n], x]^2, {x, i*h - h, i*h}] +
            Integrate[k[x] × D[phi2[x, i, n], x]^2, {x, i*h, i*h + h}], {i, 1, n - 1}]
```

Now for the off-diagonal entries:

```
In[533]:= Do[K[[i, i + 1]] =
           Integrate[k[x] × D[phi2[x, i, n], x] × D[phi1[x, i + 1, n], x], {x, i*h, i*h + h}];
          K[[i + 1, i]] = K[[i, i + 1]], {i, 1, n - 2}]
```

Here is the result:

```
In[534]:= MatrixForm[K]
```

Out[534]//MatrixForm=

$$\begin{pmatrix}
22 & -\frac{23}{2} & 0. & 0. & 0. & 0. & 0. & 0. & 0. \\
-\frac{23}{2} & 24 & -\frac{25}{2} & 0. & 0. & 0. & 0. & 0. & 0. \\
0. & -\frac{25}{2} & 26 & -\frac{27}{2} & 0. & 0. & 0. & 0. & 0. \\
0. & 0. & -\frac{27}{2} & 28 & -\frac{29}{2} & 0. & 0. & 0. & 0. \\
0. & 0. & 0. & -\frac{29}{2} & 30 & -\frac{31}{2} & 0. & 0. & 0. \\
0. & 0. & 0. & 0. & -\frac{31}{2} & 32 & -\frac{33}{2} & 0. & 0. \\
0. & 0. & 0. & 0. & 0. & -\frac{33}{2} & 34 & -\frac{35}{2} & 0. \\
0. & 0. & 0. & 0. & 0. & 0. & -\frac{35}{2} & 36 & -\frac{37}{2} \\
0. & 0. & 0. & 0. & 0. & 0. & 0. & -\frac{37}{2} & 38
\end{pmatrix}$$

Now we solve for the nodal values:

In[535]:=
```
ClearAll[u]
u = LinearSolve[K, f]
```

Out[536]=
```
{0.0131347, 0.0242576, 0.0328907, 0.0386621,
 0.0412769, 0.0404971, 0.0361283, 0.0280091, 0.0160044}
```

Finally, we assemble the data points and plot the result:

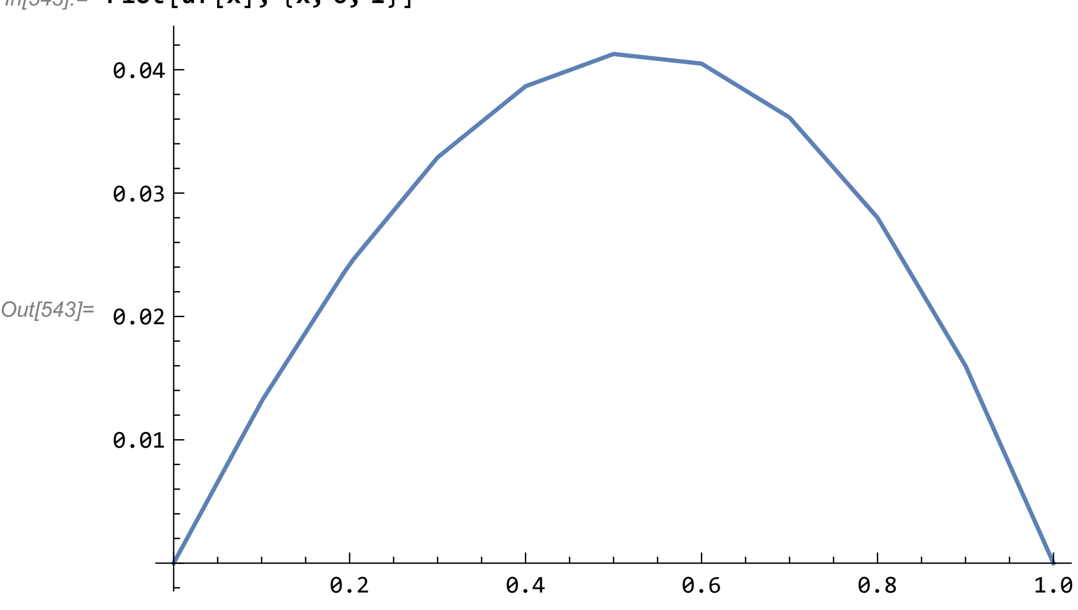
In[537]:=
```
u = Flatten[{0, u, 0}];
ClearAll[X, U]
X = Table[i * h, {i, 0, n}];
U = Transpose[{X, u}]
```

Out[540]= $\left\{\{0, 0\}, \left\{\frac{1}{10}, 0.0131347\right\}, \left\{\frac{1}{5}, 0.0242576\right\}, \left\{\frac{3}{10}, 0.0328907\right\}, \left\{\frac{2}{5}, 0.0386621\right\}, \left\{\frac{1}{2}, 0.0412769\right\}, \left\{\frac{3}{5}, 0.0404971\right\}, \left\{\frac{7}{10}, 0.0361283\right\}, \left\{\frac{4}{5}, 0.0280091\right\}, \left\{\frac{9}{10}, 0.0160044\right\}, \{1, 0\}\right\}$

In[541]:=
```
ClearAll[uf]
uf = Interpolation[U, InterpolationOrder → 1]
```

Out[542]= InterpolatingFunction[Domain: {{0., 1.}} Output: scalar]

In[543]:=
```
Plot[uf[x], {x, 0, 1}]
```

Out[543]=


Chapter 6: Heat flow and diffusion

■ Section 6.1: Fourier series methods for the heat equation

Example 6.2: An inhomogeneous example

Consider the following IBVP:

$$\frac{\partial u}{\partial t} - A\frac{\partial^2 u}{\partial x^2} = 10^{-7},\ 0 < x < 100,\ t > 0,$$
$$u(x, 0) = 0,\ 0 < x < 100,$$
$$u(0, t) = 0,\ t > 0,$$
$$u(100, t) = 0,\ t > 0.$$

The constant A has value 0.208 cm^2/s.

```
In[544]:= ClearAll[A]
          A = 0.208
Out[545]= 0.208
```

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}:

```
In[546]:= $Assumptions = Element[{m, n}, Integers]
          ClearAll[c, n, x]
          2 / 100 Integrate[10^(-7) Sin[n Pi x / 100], {x, 0, 100}]
Out[546]= m ∈ Integers
```

$$\text{Out[548]=}\ \frac{\mathrm{Sin}\left[\frac{n\pi}{2}\right]^2}{2\,500\,000\, n\,\pi}$$

In[549]:=
```
c[n_] = %
```
Out[549]=
$$\frac{\text{Sin}\left[\frac{n\pi}{2}\right]^2}{2\,500\,000\,n\,\pi}$$

We compute $a_n(t)$ by the formula $a_n(t) = \int_0^t e^{-\mathrm{A}n^2\,\pi^2(t-s)/100^2}\,c_n\,ds$.

In[550]:=
```
ClearAll[a, n, x, t]
Integrate[Exp[-A n^2 Pi^2 (t - s) / 100^2] c[n], {s, 0, t}]
```
Out[551]=
$$\frac{e^{-0.000205288\,n^2\,t}\left(-0.000620222 + 0.000620222\,e^{0.000205288\,n^2\,t}\right)\text{Sin}\left[\frac{n\pi}{2}\right]^2}{n^3}$$

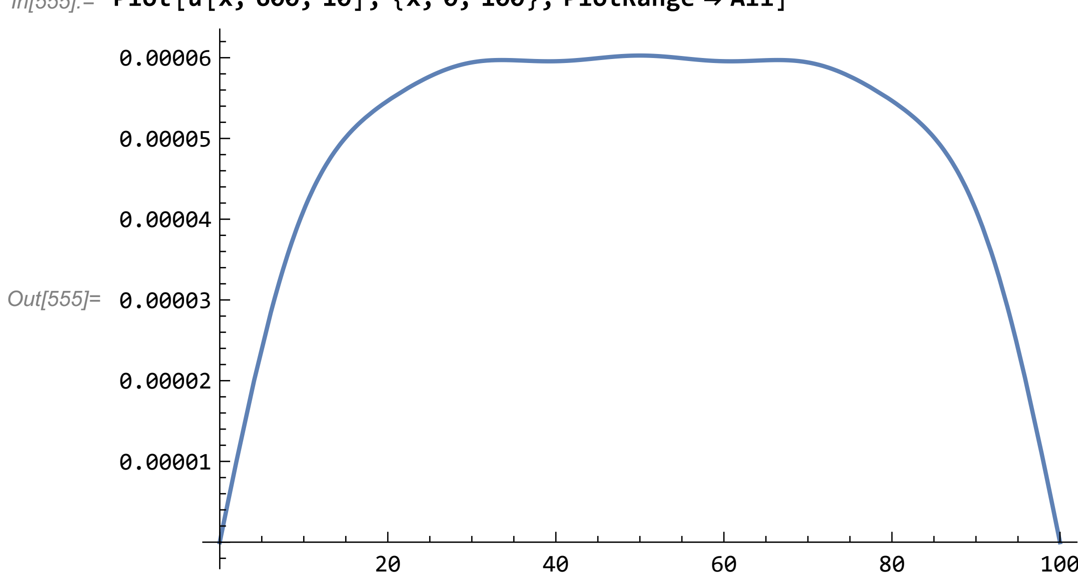
In[552]:=
```
a[t_, n_] = %
```
Out[552]=
$$\frac{e^{-0.000205288\,n^2\,t}\left(-0.000620222 + 0.000620222\,e^{0.000205288\,n^2\,t}\right)\text{Sin}\left[\frac{n\pi}{2}\right]^2}{n^3}$$

Now we define the Fourier series solution:

In[553]:=
```
ClearAll[u, x, t, M]
u[x_, t_, M_] := Sum[a[t, n] Sin[n Pi x / 100], {n, 1, M}]
```

We can easily look at some "snapshots" of the solution. For example, we 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 plot. (As discussed in the text, this number decreases as t increases, due to the smoothing of the solution.)

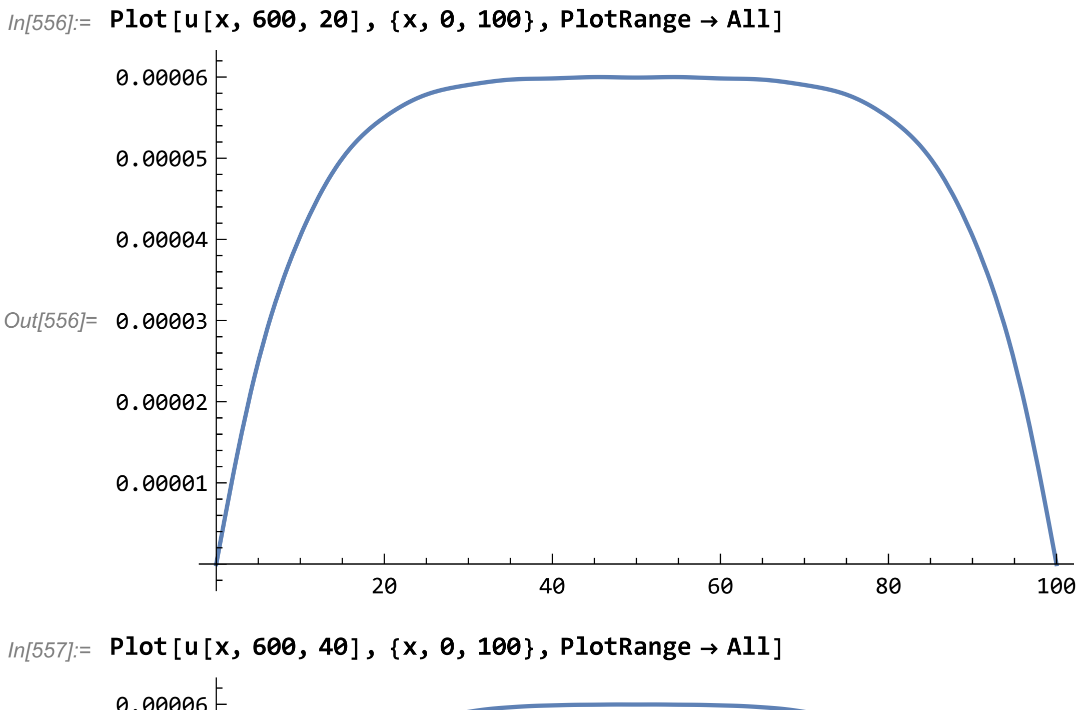
In[555]:=
```
Plot[u[x, 600, 10], {x, 0, 100}, PlotRange → All]
```
Out[555]=



The wiggles at the top of the arch suggest that we did not use enough terms in the Fourier series, so we try again with more terms. (Of course, perhaps those wiggles are really part of the solution. In that case, they will persist when we draw the graph with more terms.)

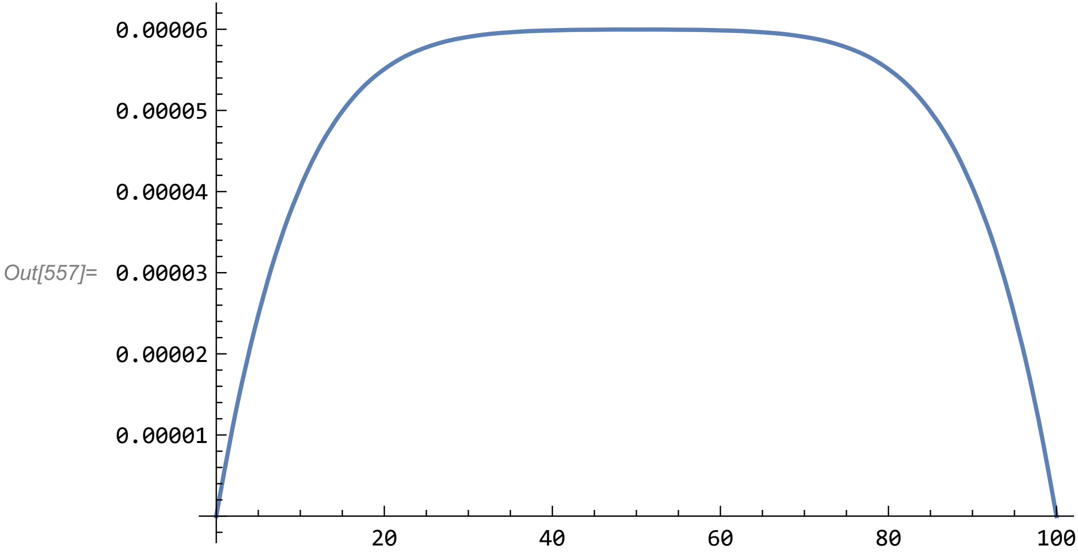
In[556]:= **Plot[u[x, 600, 20], {x, 0, 100}, PlotRange → All]**

Out[556]=

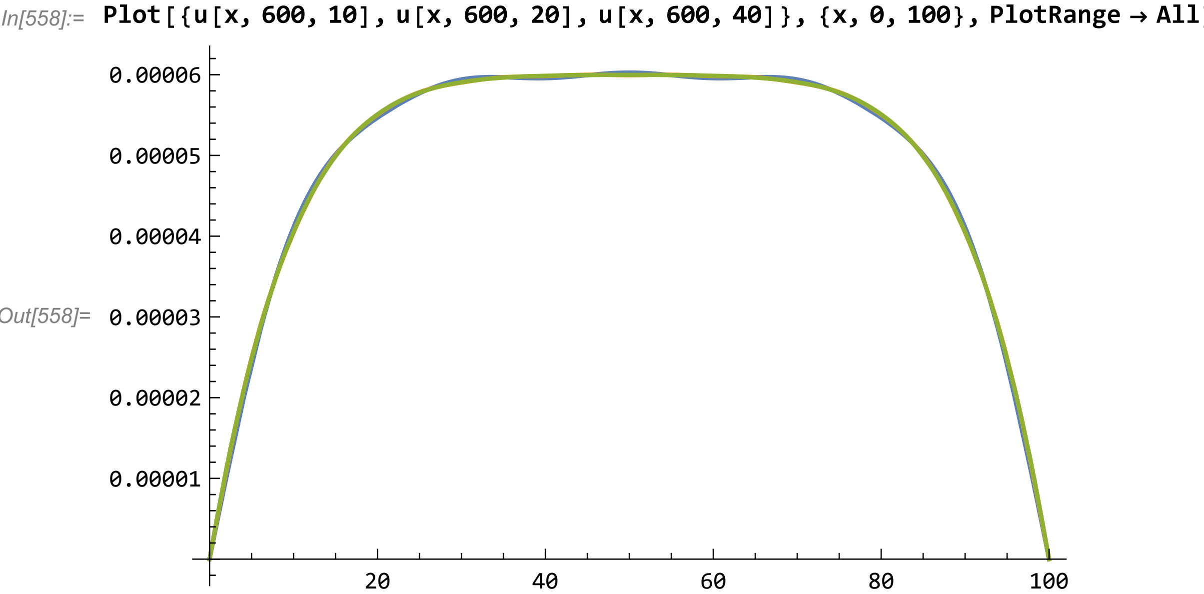


In[557]:= **Plot[u[x, 600, 40], {x, 0, 100}, PlotRange → All]**

Out[557]=



In[558]:=
```
Plot[{u[x, 600, 10], u[x, 600, 20], u[x, 600, 40]}, {x, 0, 100}, PlotRange → All]
```

Out[558]=


The above graphs suggest that 20 terms is enough for a qualitatively correct graph at *t*=600.

Section 6.4: Finite element methods for the heat equation

Now we show how to use the backward Euler method with the finite element method 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 we do not attempt to do so. Instead, the command **beuler** (defined below) applies the algorithm to the system of ODEs

$$M\frac{d\alpha}{dt}+K\alpha=f(t),\ \ t>0,$$
$$\alpha(0)=\alpha_{0,}$$

which is the result of applying the finite element method to the heat equation.

In[559]:=
```
ClearAll[M, K, f, a0, n, dt, beuler]
beuler[M_, K_, f_, a0_, n_, dt_] :=
 Module[{i, L}, U = Table[{i * dt, Null}, {i, 0, n}];
  U[[1, 2]] = a0;
  L = M + dt * K;
  Do[U[[i + 1, 2]] = LinearSolve[L, M.U[[i, 2]] + dt * f[U[[i + 1, 1]]]], {i, 1, n}];
  U]
```

To solve a specific problem, we have to compute the mass matrix *M*, the stiffness matrix *K*, the load vector *f*, and the initial data α_0. The techniques should by now be familiar.

Example 6.8

An 100 cm iron bar, with ρ=7.88 g/cm^3, c=0.437 J/(g K), and κ=0.836 W/(cm K), 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 following function:

In[561]:=
```
ClearAll[F, x, t]
F[x_, t_] = 10^(-8) t x (100 - x)^2
```

Out[562]= $\frac{t\ (100 - x)^2\ x}{100\,000\,000}$

The temperature distribution is the solution of the IBVP

$$\rho c\,\frac{\partial u}{\partial t} - \kappa\,\frac{\partial^2 u}{\partial x^2} = F\,(x, t),\ 0 < x < 100,\ t > 0,$$

$$u\,(x, 0) = 0,\ 0 < x < 100,$$

$$u\,(0, t) = 0,\ t > 0,$$

$$u\,(100, t) = 0,\ t > 0.$$

We begin by defining the mesh and the constants:

In[563]:=
```
ClearAll[n, h, k, p, c]
n = 100
h = 100. / n
k = 0.836
p = 7.88
c = 0.437
```

Out[564]= 100

Out[565]= 1.

Out[566]= 0.836

Out[567]= 7.88

Out[568]= 0.437

Next we define the stiffness matrix (for this constant coefficient problem, there is no need to perform any integrations---we already know the entries in the stiffness matrix):

In[569]:=
```
ClearAll[K, i, j]
K = Table[0.0, {i, 1, n - 1}, {j, 1, n - 1}];
Do[K[[i, i]] = 2 k / h, {i, 1, n - 1}]
Do[K[[i, i + 1]] = -k / h;
 K[[i + 1, i]] = K[[i, i + 1]], {i, 1, n - 2}]
```

Similarly, we know the entries of the mass matrix:

In[573]:=
```
ClearAll[M, i, j]
M = Table[0.0, {i, 1, n - 1}, {j, 1, n - 1}];
Do[M[[i, i]] = 2 h p c / 3, {i, 1, n - 1}]
Do[M[[i, i + 1]] = h p c / 6;
 M[[i + 1, i]] = M[[i, i + 1]], {i, 1, n - 2}]
```

The load vector $f = f(t)$ (which is a vector-valued function) can be computed with the **Table** command:

In[577]:=
```
ClearAll[phi1, phi2, x, i]
phi1[x_, i_] = (x - (i - 1) * h) / h
phi2[x_, i_] = - (x - (i + 1) * h) / h
```

Out[578]= `1. (-1. (-1 + i) + x)`

Out[579]= `1. (1. (1 + i) - x)`

In[580]:=
```
ClearAll[f, t]
f[t_] = Table[Integrate[F[x, t] × phi1[x, i], {x, i * h - h, i * h}] +
    Integrate[F[x, t] × phi2[x, i], {x, i * h, i * h + h}], {i, 1, n - 1}];
```

Finally, before invoking **beuler**, we need the initial vector:

In[582]:=
```
ClearAll[a0]
a0 = Table[0.0, {i, 1, n - 1}];
```

Now we choose the time step and the number of time steps, and invoke the backward Euler method:

In[584]:=
```
ClearAll[dt, steps, U]
dt = 2.
steps = 180. / dt
U = beuler[M, K, f, a0, steps, dt];
```

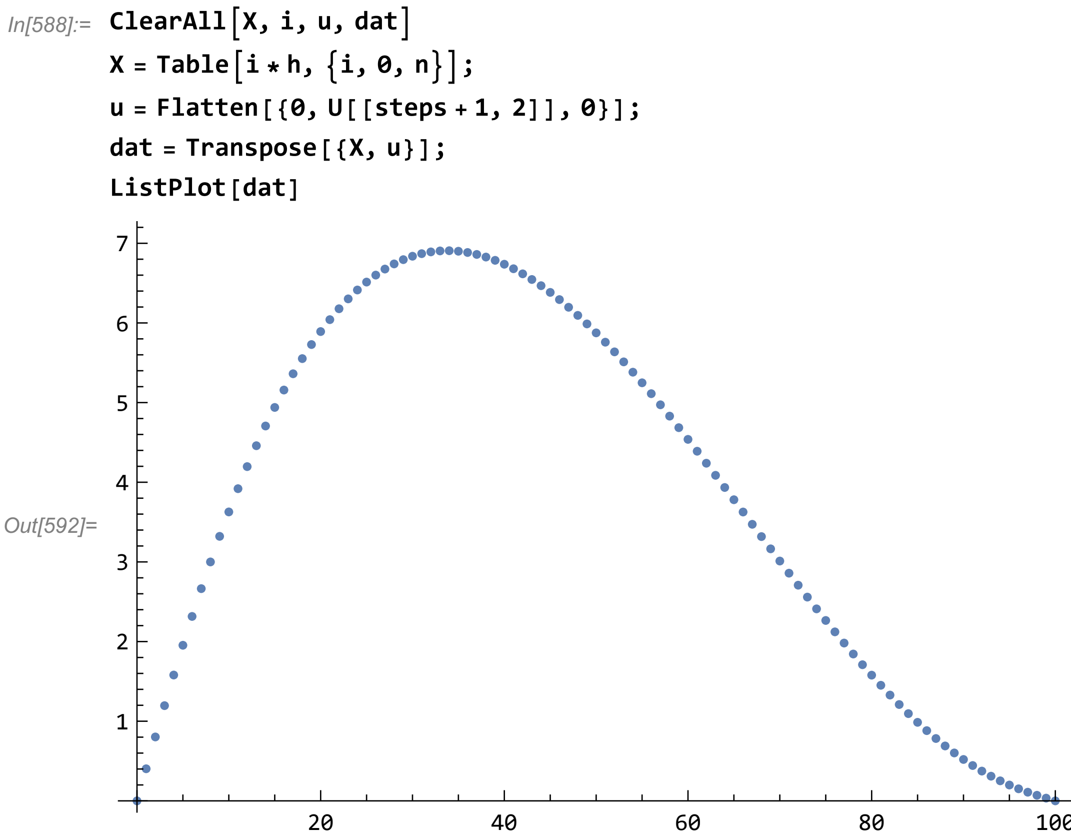
Out[585]= `2.`

Out[586]= `90.`

Finally, we set up a table of data and display the result at time t=10. Note that **U[[steps+1,1]]** is the final time (in this case, t = 180) and **U[[steps+1,2]]** is the vector of nodal values of the piecewise linear function approximating $u(x, t)$ for this value of t.

In[588]:=
```
ClearAll[X, i, u, dat]
X = Table[i * h, {i, 0, n}];
u = Flatten[{0, U[[steps + 1, 2]], 0}];
dat = Transpose[{X, u}];
ListPlot[dat]
```

Out[592]=



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.

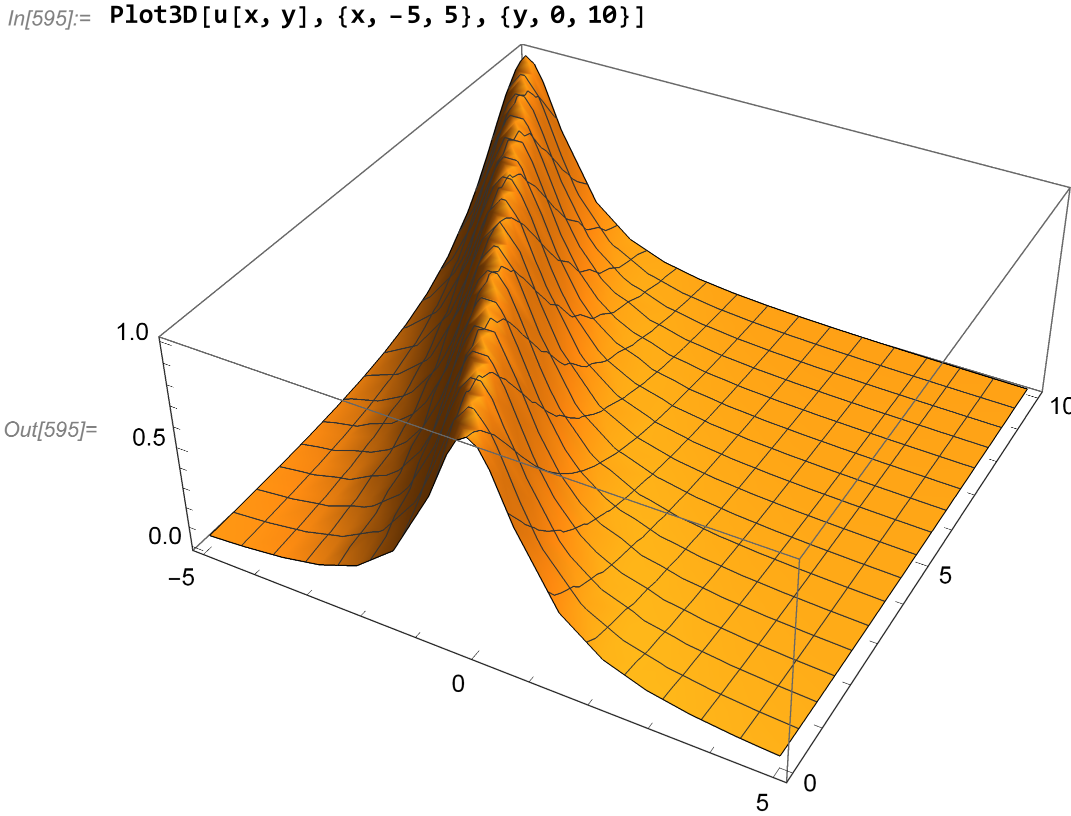
The **Plot3D** function plots a function of two variables:

In[593]:=
```
ClearAll[u, x, y]
u[x_, y_] = 1 / (1 + (x + y / 2) ^2)
```

Out[594]= $\frac{1}{1+\left(x+\frac{y}{2}\right)^2}$

In[595]:=
```
Plot3D[u[x, y], {x, -5, 5}, {y, 0, 10}]
```

Out[595]=



If you would like to see the surface from a different angle, you can click on the figure and rotate it by moving the mouse (i.e. put the pointer on the figure, hold down the left mouse button, and move the mouse).

■ 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. *Mathematica* has a function called **DSolve** that will solve many ordinary differential equations symbolically. We will illustrate the use of **DSolve** in the context of Example 8.6 from the text.

The method of characteristics reduces the PDE in Example 8.6 to the IVP

$$\frac{dv}{dt} = v^2, \; v(0) = \frac{1}{1+s^2}.$$

DSolve will solve this problem as follows:

In[596]:=
```
ClearAll[v, t, s]
DSolve[{v'[t] == v[t]^2, v[0] == 1 / (1 + s^2)}, v[t], t]
```

Out[597]= $\left\{\left\{v[t] \to \frac{1}{1+s^2-t}\right\}\right\}$

If no initial condition is given, **DSolve** will return the general solution of the ODE:

```
In[598]:= DSolve[v'[t] == v[t]^2, v[t], t]
```

Out[598]= $\left\{\left\{v[t] \to \frac{1}{-t - C[1]}\right\}\right\}$

DSolve will solve a system of ODEs (when possible). Here is the system of ODEs from Example 8.7:

$$\frac{dx}{dt} = v,\ x(0) = s,$$
$$\frac{dy}{dt} = y,\ y(0) = 1,$$
$$\frac{dv}{dt} = x,\ v(0) = 2s.$$

The solution is given as follows:

```
In[599]:= ClearAll[x, y, v, t, s, sol]
          sol = DSolve[{x'[t] == v[t], y'[t] == y[t], v'[t] == x[t],
             x[0] == s, y[0] == 1, v[0] == 2 s}, {x[t], y[t], v[t]}, t]
```

Out[600]= $\left\{\left\{v[t] \to \frac{1}{2} e^{-t}\left(1 + 3 e^{2t}\right) s,\ x[t] \to \frac{1}{2} e^{-t}\left(-1 + 3 e^{2t}\right) s,\ y[t] \to e^{t}\right\}\right\}$

The solutions can be defined for further use as follows:

```
In[601]:= x[t_] = (x[t] /. sol[[1]])
          y[t_] = (y[t] /. sol[[1]])
          v[t_] = (v[t] /. sol[[1]])
```

Out[601]= $\frac{1}{2} e^{-t}\left(-1 + 3 e^{2t}\right) s$

Out[602]= e^{t}

Out[603]= $\frac{1}{2} e^{-t}\left(1 + 3 e^{2t}\right) s$

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, *Mathematica* makes this easy. We do not need to learn any new commands, since a double integral over a rectangle can be computed as an iterated integral.

As an example, we compute the Fourier double sine series of the following function *f* on the unit square:

In[604]:=
```
ClearAll[f, x, y]
f[x_, y_] = x (1 - x) y (1 - y) ^2
```

Out[605]= $(1-x)\, x\, (1-y)^2\, y$

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

$$a_{mn} = 4\int_0^1\int_0^1 f(x, y) \sin(m\pi x) \sin(n\pi y)\, dy dx.$$

We calculate a_{mn} directly and define a function $a(m, n)$ for convenience. As usual, when performing Fourier series computations, we want *Mathematica* to know which symbols represent integers.

In[606]:=
```
$Assumption = Element[{m, n}, Integers]
```

Out[606]= m ∈ Integers

In[607]:=
```
ClearAll[m, n, x, y]
a[m_, n_] =
 4 Integrate[Integrate[f[x, y] Sin[m Pi x] Sin[n Pi y], {y, 0, 1}], {x, 0, 1}]
```

Out[608]= $-\frac{8\,(-1+(-1)^m)\,(2\,n\,\pi\,(2+\text{Cos}[n\,\pi])-6\,\text{Sin}[n\,\pi])}{m^3\,n^4\,\pi^7}$

Note that, when computing an iterated integral, there is no need to nest the **Integrate** commands (that is, a single **Integrate** suffices):

In[609]:=
```
4 Integrate[f[x, y] Sin[m Pi x] Sin[n Pi y], {y, 0, 1}, {x, 0, 1}]
```

Out[609]= $-\frac{8\,(-1+(-1)^m)\,(2\,n\,\pi\,(2+\text{Cos}[n\,\pi])-6\,\text{Sin}[n\,\pi])}{m^3\,n^4\,\pi^7}$

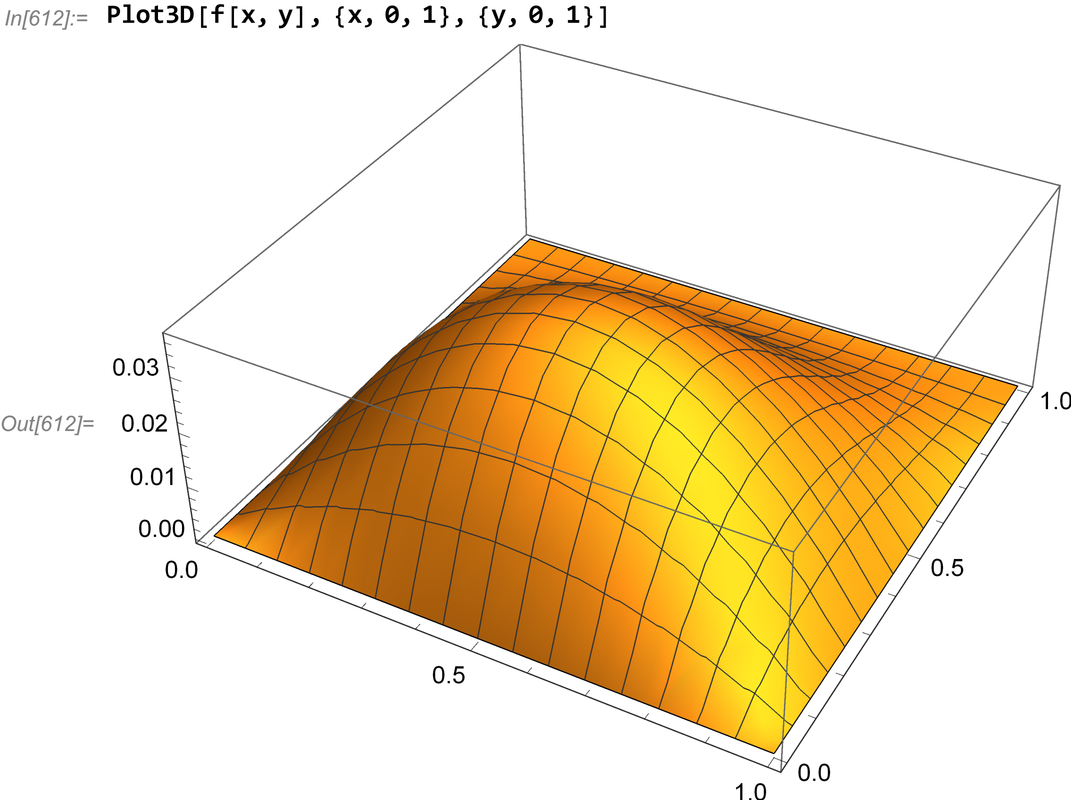
The same is true when defining a double sum. Here is the function representing the (partial) Fourier series:

In[610]:=
```
ClearAll[S, x, y, M]
S[x_, y_, M_] := Sum[a[m, n] Sin[m Pi x] Sin[n Pi y], {n, 1, M}, {m, 1, M}]
```

Here is the graph of f:

In[612]:= `Plot3D[f[x, y], {x, 0, 1}, {y, 0, 1}]`

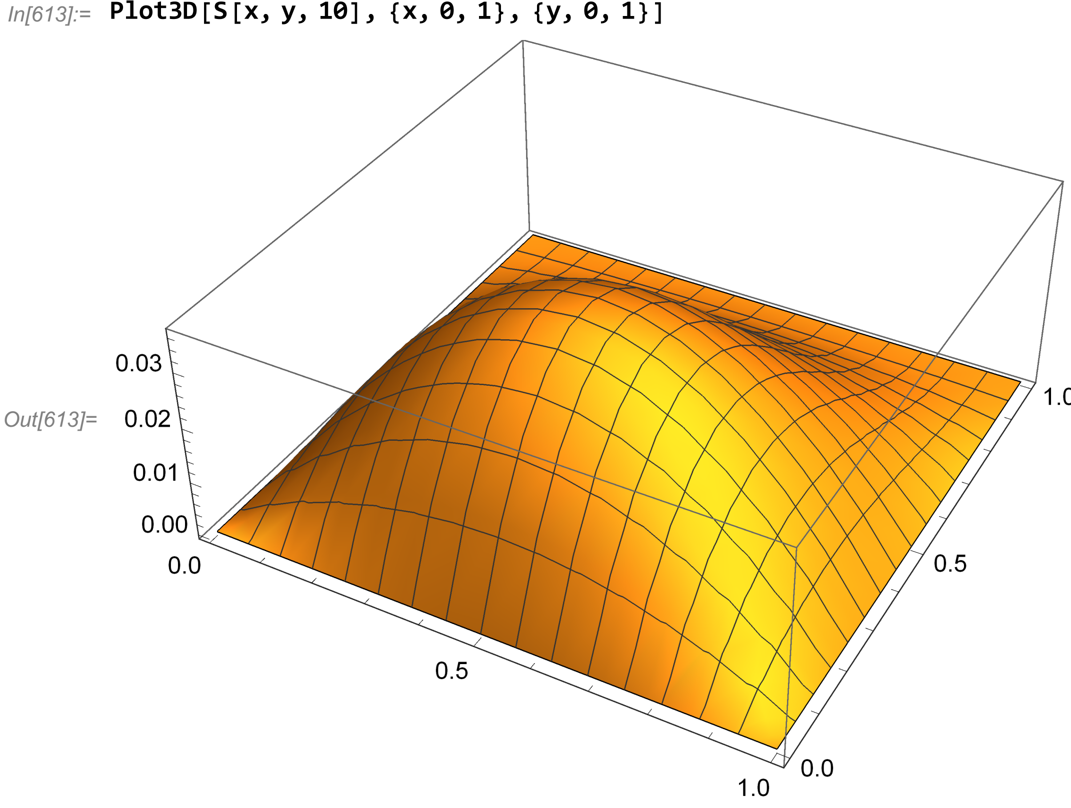
Out[612]=



We compare the above graph with the partial Fourier series having 100 terms:

In[613]:= `Plot3D[S[x, y, 10], {x, 0, 1}, {y, 0, 1}]`

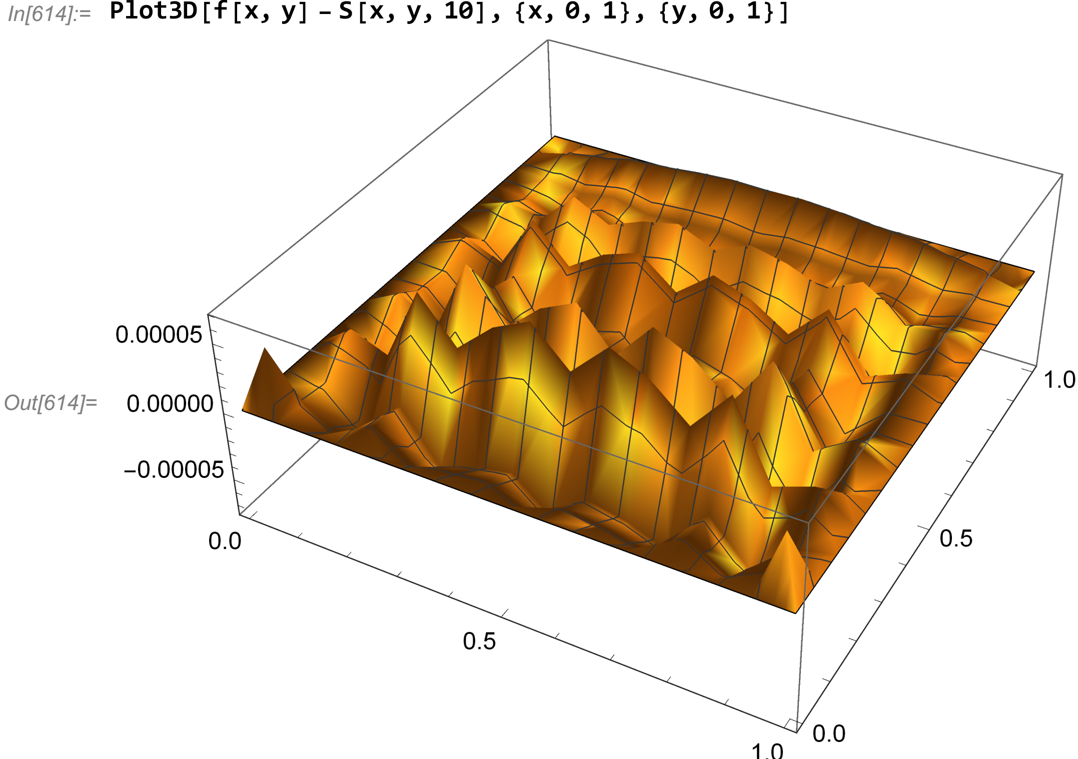
Out[613]=



The approximation looks pretty good. To confirm this, we graph the error:

In[614]:= `Plot3D[f[x, y] - S[x, y, 10], {x, 0, 1}, {y, 0, 1}]`

Out[614]=



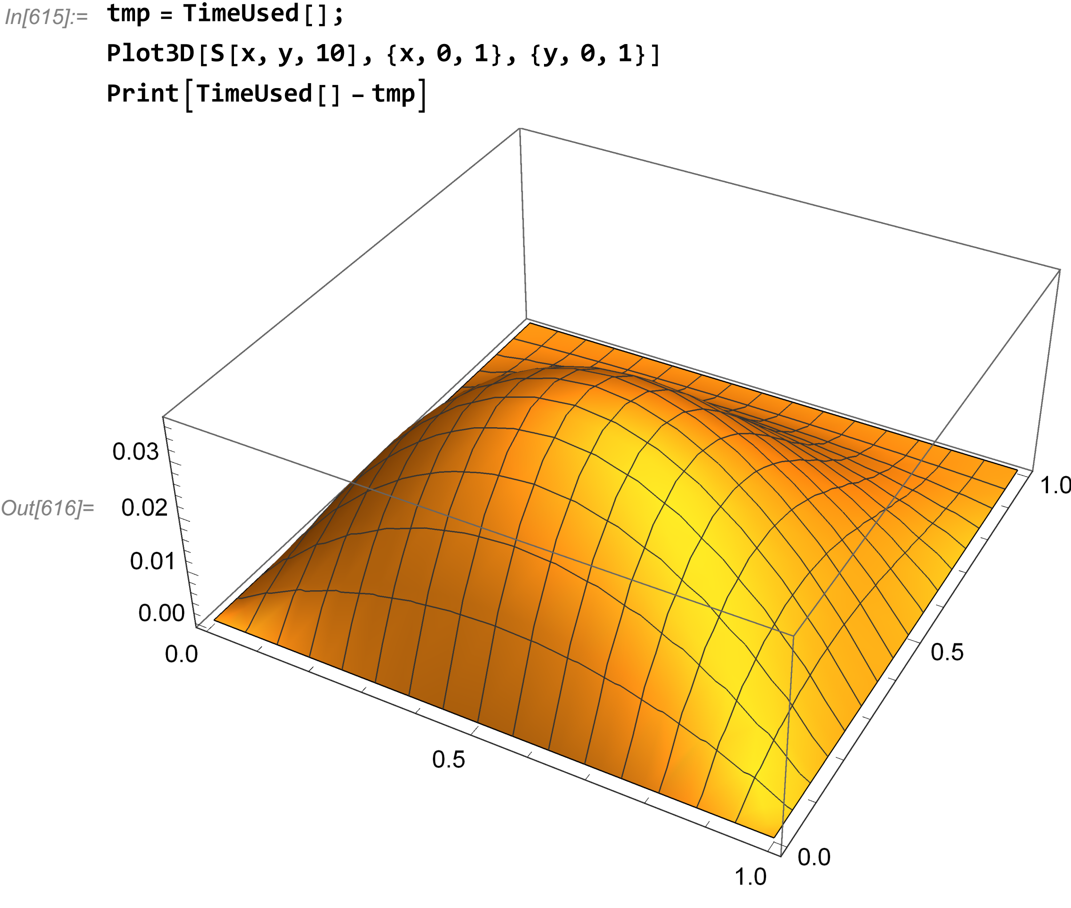
Looking at the vertical scale on the last two plots, we see that the relative difference between *f* and its partial Fourier series with 100 terms is only about 0.1%.

Improving efficiency of double Fourier series calculations

If you create a few plots of a double Fourier series, particularly experimenting with the number of terms in the series, you will soon notice that it takes quite a bit of time to produce the plots. This is not surprising, when you realize that **Plot3D** samples the function on a 15 by 15 grid (225 points) by default. Evaluating a sum with hundreds or thousands of terms that many times is bound to be time-consuming. To demonstrate this, we use the built-in **TimeUsed** function, which records the CPU time used by the *Mathematica* kernel since the beginning of the session.

```
In[615]:= tmp = TimeUsed[];
          Plot3D[S[x, y, 10], {x, 0, 1}, {y, 0, 1}]
          Print[TimeUsed[] - tmp]
```

Out[616]=




```
18.5977
```

So it took about 16 seconds to create the above graph. (When you execute the above on your computer, you will most likely get a different result; your computer may be faster or slower.)

One way to gain efficiency is to force Mathematicato evaluate expressions numerically early in its computations rather than late. Certainly, in computing points to plot, Mathematica must, in the end, obtain numerical values. However, it is liable to manipulate the expression symbolically as much as it can before doing a numerical evaluation. It is better to force it to do numerical computation throughout. In our example, we can do this by defining the coefficients as numerical rather than symbolic quantities:

In[618]:=
```
tmp = a[m, n]
```

Out[618]=
$$-\frac{8\,(-1+(-1)^m)\,(2\,n\,\pi\,(2+\mathrm{Cos}[n\,\pi])-6\,\mathrm{Sin}[n\,\pi])}{m^3\,n^4\,\pi^7}$$

In[619]:=
```
ClearAll[a, m, n]
a[m_, n_] = N[tmp]
```

Out[620]=
$$-\frac{0.00264875\,(-1.+(-1.)^m)\,(6.28319\,n\,(2.+\mathrm{Cos}[3.14159\,n])-6.\,\mathrm{Sin}[3.14159\,n])}{m^3\,n^4}$$

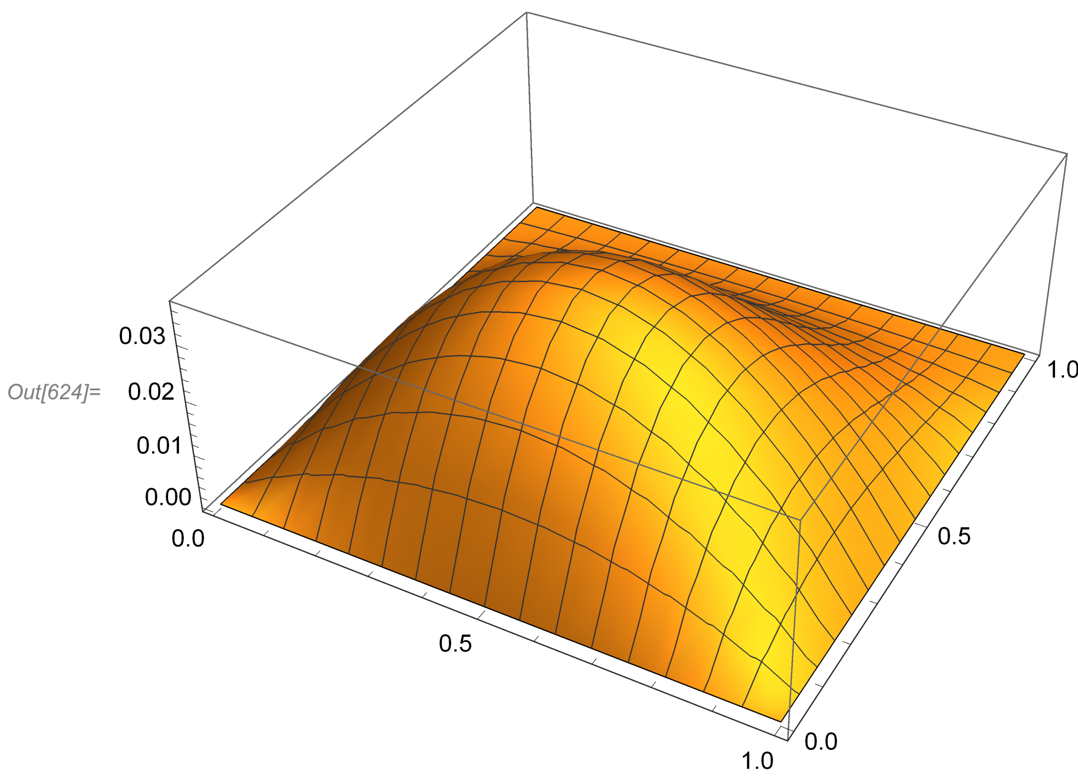
Now that a_{mn} is defined by numerical quantities, we redefine S:

In[621]:=
```
ClearAll[S, x, y, M, m, n]
S[x_, y_, M_] := Sum[Sin[m Pi x] Sum[a[m, n] Sin[n Pi y], {n, 1, M}], {m, 1, M}]
```

This allows *Mathematica* to produce the plot much more quickly:

In[623]:=
```
tmp = TimeUsed[];
Plot3D[S[x, y, 10], {x, 0, 1}, {y, 0, 1}]
Print[TimeUsed[] - tmp]
```

Out[624]=

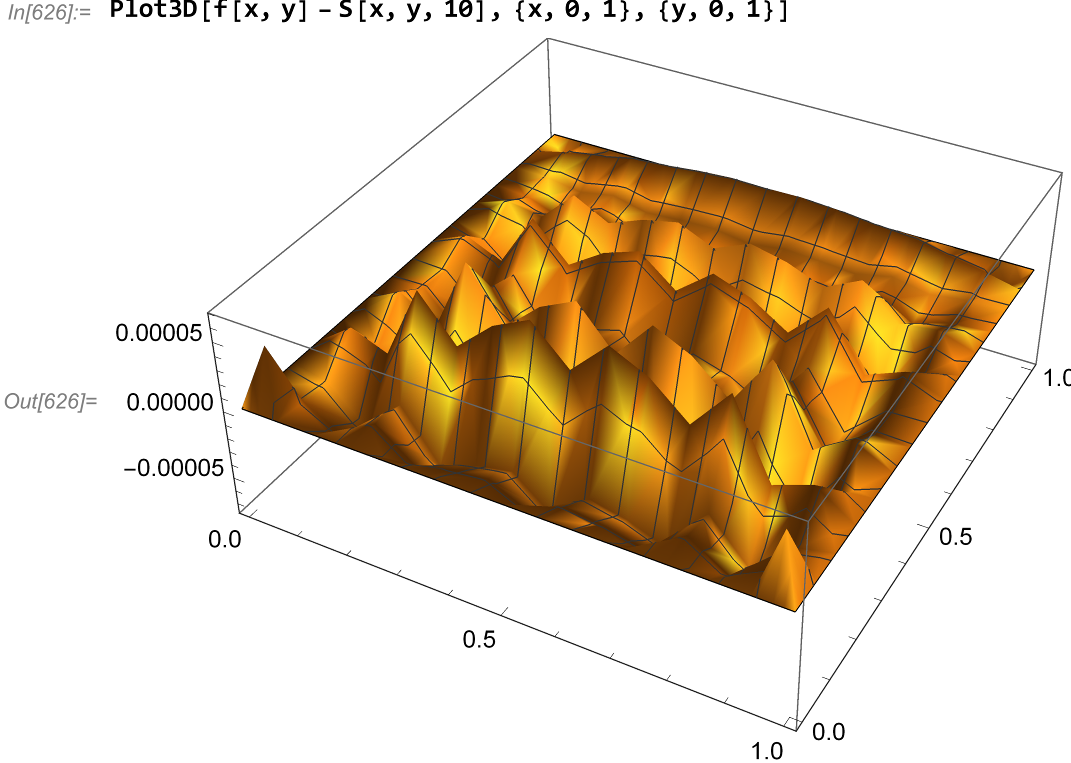



```
0.845885
```

By the way, let us look again at the error in the Fourier approximation:

In[626]:=
```
Plot3D[f[x, y] - S[x, y, 10], {x, 0, 1}, {y, 0, 1}]
```

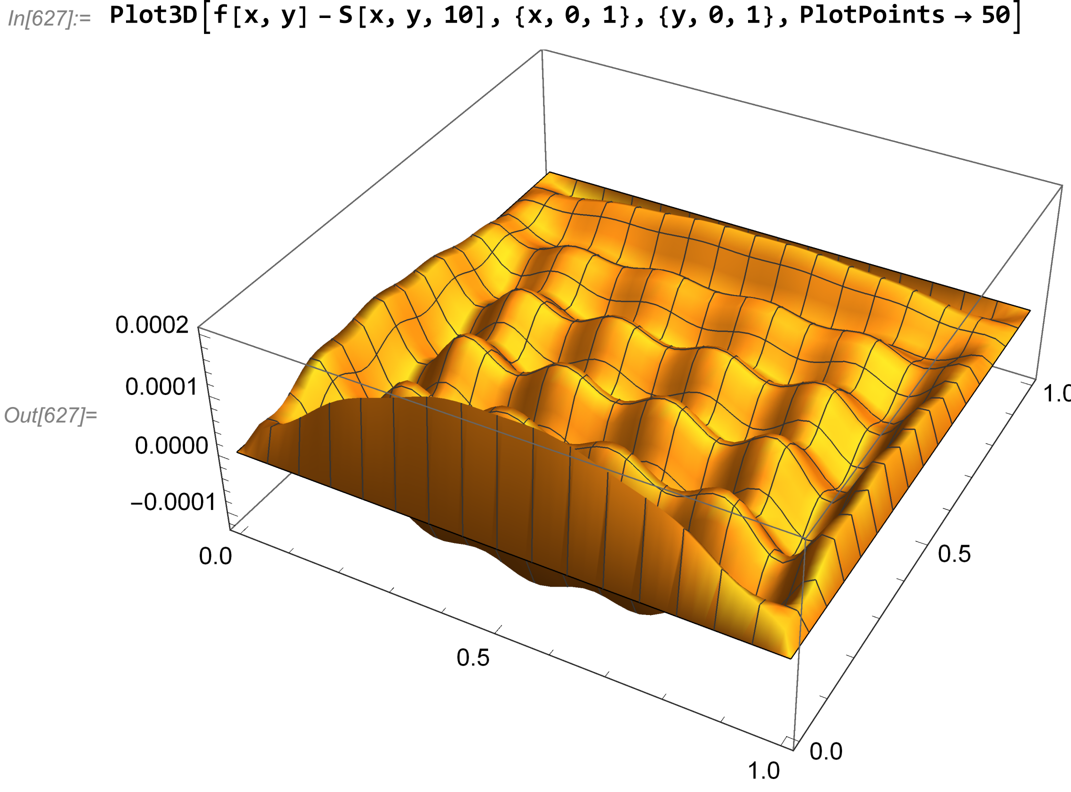
Out[626]=



Since both *f* and *S* are smooth functions, why does the surface appear so jagged? The answer is simply that the grid on which *Mathematica* sampled the function $f - S$ is not fine enough to give an accurate graph. We can produce a more accurate graph by refining this grid via the **PlotPoints** option. The option **PlotPoints->k** causes *Mathematica* to use a grid with k^2 points (the default is k=15). Of course, requesting a finer grid will result in a more time-consuming computation:

In[627]:=
```
Plot3D[f[x, y] - S[x, y, 10], {x, 0, 1}, {y, 0, 1}, PlotPoints → 50]
```

Out[627]=

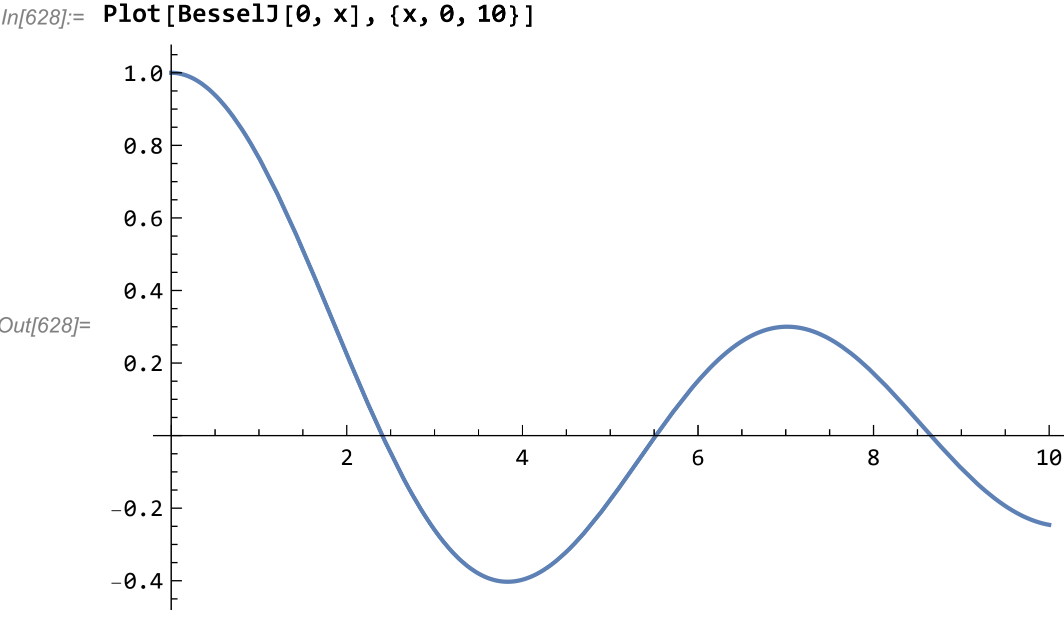


■ Section 11.3: Fourier series on a disk

The Bessel functions $J_n(s)$ are built-in functions in *Mathematica*, just as are the more common elementary functions sine, cosine, exp, and so forth, and can be used just as conveniently. For example, here is the graph of J_0:

In[628]:=
```
Plot[BesselJ[0, x], {x, 0, 10}]
```

Out[628]=



(Notice that the first argument to **BesselJ** is the index n.) As an example, we compute the smallest roots s_{01}, s_{02}, and s_{03} of

J_0. The above graph shows that these roots are about 2.5, 5.5, and 8.5, respectively.

In[629]:=
```
FindRoot[BesselJ[0, x] == 0, {x, 2.5}]
```
Out[629]=
```
{x → 2.40483}
```

In[630]:=
```
FindRoot[BesselJ[0, x] == 0, {x, 5.5}]
```
Out[630]=
```
{x → 5.52008}
```

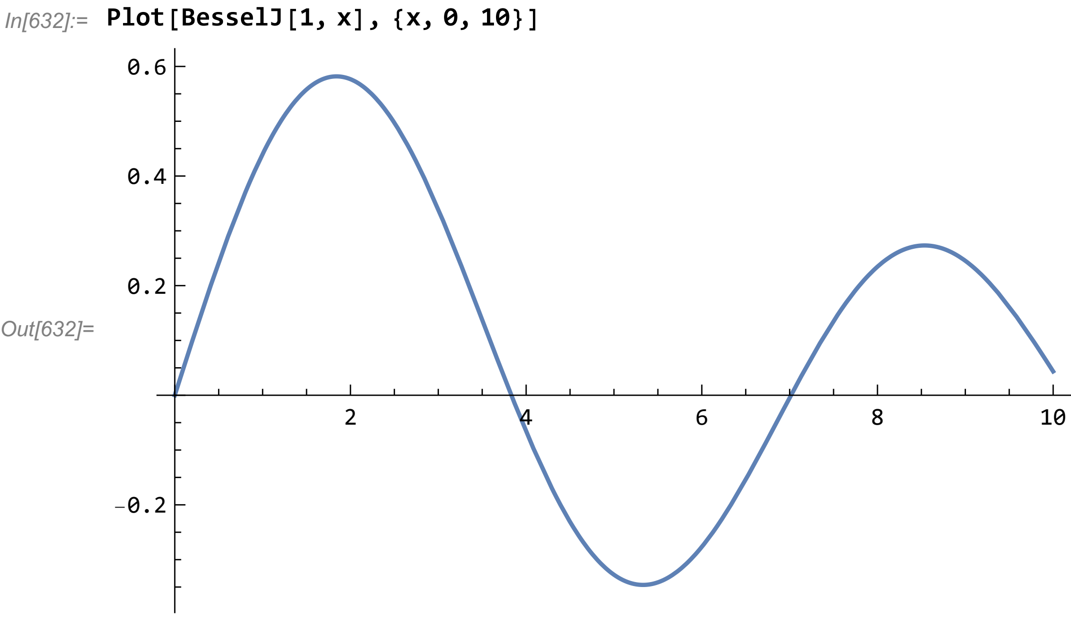
In[631]:=
```
FindRoot[BesselJ[0, x] == 0, {x, 8.5}]
```
Out[631]=
```
{x → 8.65373}
```

Graphics on the disk

Functions defined on a disk are naturally described by cylindrical coordinates, that is, as z=f(r,θ), where (r,θ) are polar coordinates. We must use a special command to draw the surface of such a function: **ParametricPlot3D**. This command will graph any surface parametrized by two variables. As an example, we will use it to graph a surface z=f(r,θ), thinking of r and θ as the parameters. **ParametricPlot3D** requires that we also parametrize the other cartesian coordinates, x and y. We do this using the standard relationships $x = r\cos(\theta)$ and $y = r\sin(\theta)$.

For example, consider the eigenfunction $\phi_{11}^{(1)}(r, \theta) = J_1(s_{11}\, r)\cos(\theta)$ of the Laplacian on the unit disk. First, we must compute the root s_{11} of J_1:

In[632]:=
```
Plot[BesselJ[1, x], {x, 0, 10}]
```
Out[632]=



In[633]:=
```
FindRoot[BesselJ[1, x], {x, 4.}]
```
Out[633]=
```
{x → 3.83171}
```

In[634]:=
```
s11 = (x /. %)
```
Out[634]=
```
3.83171
```

In[635]:=
```
ClearAll[r, t, phi]
phi[r_, t_] = BesselJ[1, s11 r] Cos[t]
```

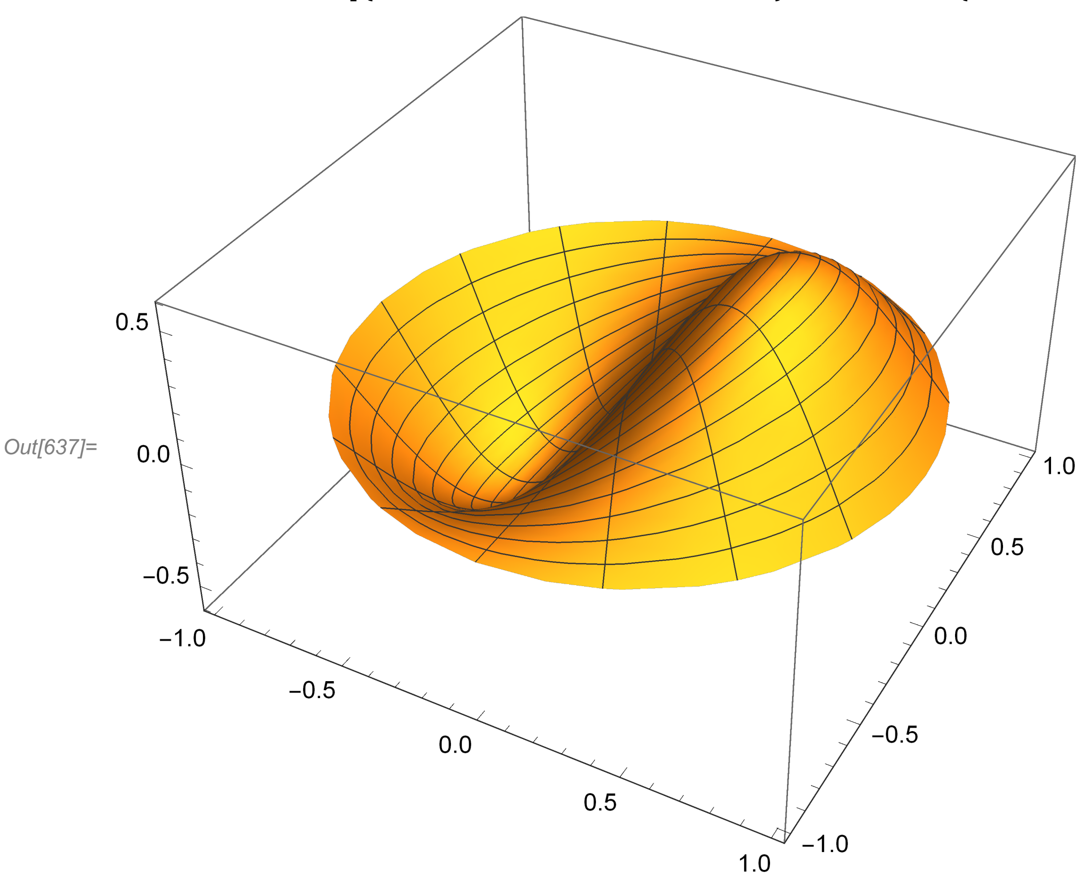
Out[636]=
```
BesselJ[1, 3.83171 r] Cos[t]
```

Here is the plot of the parametrized surface:

In[637]:=

```
ParametricPlot3D[{r Cos[t], r Sin[t], phi[r, t]}, {r, 0, 1}, {t, 0, 2 Pi}]
```

Out[637]=



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 represented by **I** in *Mathematica*. As an example, we compute the complex Fourier series of the function $f(x) = x^2$ on the interval [-1,1].

In[638]:=
```
$Assumptions = Element[{m, n}, Integers]
```
Out[638]= (m | n) ∈ Integers

In[639]:=
```
ClearAll[f, x, c, n]
f[x_] = x^2
c[n_] = Simplify[(1 / 2) Integrate[f[x] Exp[-I Pi n x], {x, -1, 1}]]
```
Out[640]= x^2

Out[641]= $\frac{2\,(-1)^{\text{Abs}[n]}}{n^2\,\pi^2}$

The foregoing formula obviously does not hold for n=0, and so we need to compute the coefficient c_0 separately:

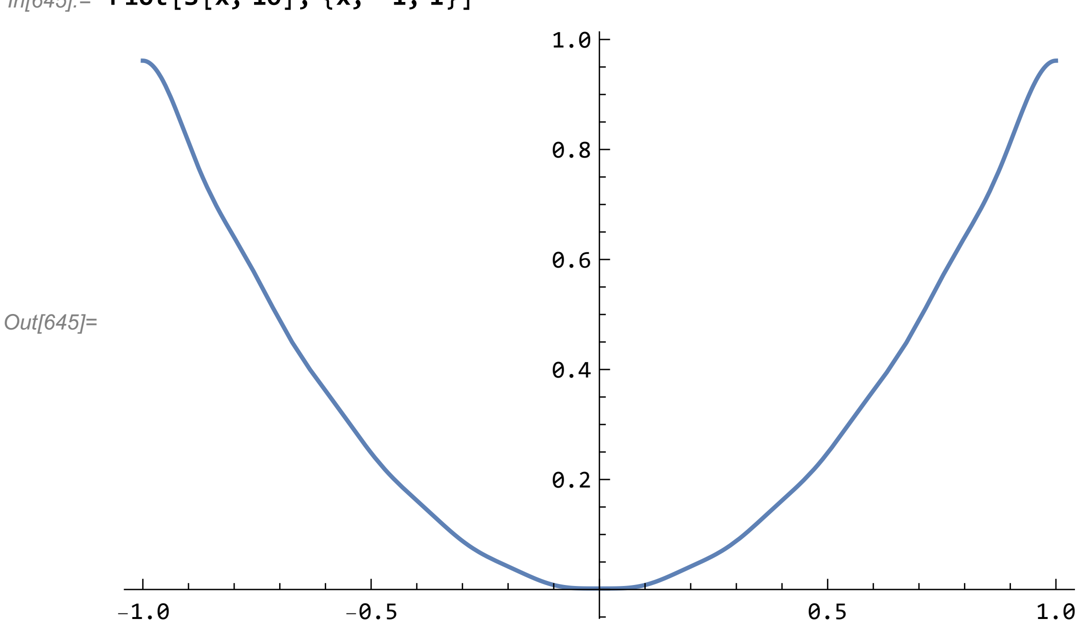
In[642]:=
```
c[0] = Simplify[(1 / 2) Integrate[f[x], {x, -1, 1}]]
```
Out[642]= $\frac{1}{3}$

Now we can define the partial Fourier series. Recall from Section 12.1 of the text that, since f is real-valued, the following partial Fourier series is also real-valued:

In[643]:=
```
ClearAll[S, x, M, n]
S[x_, M_] :=
 c[0] + Sum[c[n] Exp[I n Pi x], {n, -M, -1}] + Sum[c[n] Exp[I n Pi x], {n, 1, M}]
```

We check the computation with a graph:

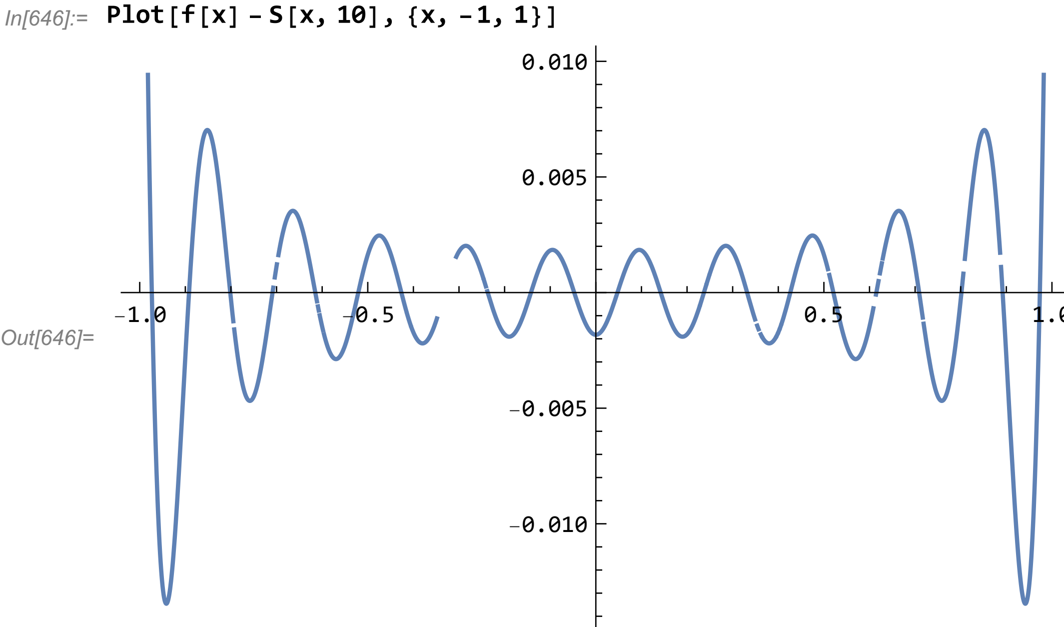
In[645]:=
```
Plot[S[x, 10], {x, -1, 1}]
```
Out[645]=



The approximation is not bad, as the following graph of the error shows:

In[646]:= Plot[f[x] - S[x, 10], {x, -1, 1}]

Out[646]=



■ Section 12.2: Fourier series and the FFT

Mathematica implements the fast Fourier transform in the command **Fourier**. As mentioned in Section 12.2.3 in the text, there is more than one way to define the FFT (although all are essentially equivalent), and *Mathematica* does not adopt the same definition as in the text. However, the **Fourier** command has an option called **FourierParameters** that allows you to select the definition of the FFT used. You can consult the documentation on **Fourier** for a complete description of the possibilities for **FourierParameters**, but, for our purposes, it is enough to know that **FourierParameters**->{-1,-1} gives the same definition of the FFT as is used in the textbook.

To illustrate the use of the FFT in *Mathematica*, here is part of Example 12.3 from Section 12.2. We begin with the sequence $f_{-3}, f_{-2}, f_{-1}, f_0, f_1, f_2$ as follows:

```
In[647]:= ClearAll[f]
          f = N[{0, -(2/3)^3, -(1/3)^3, 0, (1/3)^3, (2/3)^3}]

Out[648]= {0., -0.296296, -0.037037, 0., 0.037037, 0.296296}
```

We then rearrange the sequence as explained in Section 12.2.2:

```
In[649]:= ClearAll[f1]
          f1 = {f[[4]], f[[5]], f[[6]], f[[1]], f[[2]], f[[3]]}

Out[650]= {0., 0.037037, 0.296296, 0., -0.296296, -0.037037}
```

Now we find the FFT of **f1**. You should note the use of FourierParameters in this command; without it, *Mathematica* will use a slightly different definition of the FFT, and the results will be unexpected.

```
In[651]:= ClearAll[F1]
          F1 = Fourier[f1, FourierParameters → {-1, -1}]

Out[652]= {0. + 0. i, 0. - 0.096225 i, 0. + 0.0748417 i, 0. + 0. i, 0. - 0.0748417 i, 0. + 0.096225 i}
```

Finally, we rearrange **F1** to find the desired sequence **F**:

```
In[653]:= ClearAll[F]
          F = {F1[[4]], F1[[5]], F1[[6]], F1[[1]], F1[[2]], F1[[3]]}
Out[654]= {0. + 0. i, 0. - 0.0748417 i, 0. + 0.096225 i, 0. + 0. i, 0. - 0.096225 i, 0. + 0.0748417 i}
```

The results are the same as in Example 12.3, up to round-off error.

When using the FFT to perform Fourier series computations, it is necessary to swap the first and second halves of a sequence, as demonstrated in the above example. Using the **Take** and **Flatten** commands, this can easily be done. **Take** extracts a sublist from a list; **Take[f,{m,n}]** creates a list whose entries are entries m,m+1,...,n from list f. You should recall that **Flatten** creates a single list from a nested list. Thus

```
In[655]:= Flatten[{Take[f, {4, 6}], Take[f, {1, 3}]}]
Out[655]= {0., 0.037037, 0.296296, 0., -0.296296, -0.037037}
```

gives the same result as

```
In[656]:= {f[[4]], f[[5]], f[[6]], f[[1]], f[[2]], f[[3]]}
Out[656]= {0., 0.037037, 0.296296, 0., -0.296296, -0.037037}
```

If you are going to do this manipulation often, it makes sense to create a command to do it:

```
In[657]:= ClearAll[shift]
          shift[f_] := Module[{F, n},
            n = Length[f];
            F = Flatten[{Take[f, {n / 2 + 1, n}], Take[f, {1, n / 2}]}];
            F]
```

```
In[659]:= shift[f]
Out[659]= {0., 0.037037, 0.296296, 0., -0.296296, -0.037037}
```

By the way, we can also define a command to reduce the typing associated with the **Fourier** command:

```
In[660]:= ClearAll[fft]
          fft[f_] := Fourier[f, FourierParameters → {-1, -1}]
```

Here is the calculation from Example 12.3, condensed to a single line!

```
In[662]:= shift[fft[shift[f]]]
Out[662]= {0. + 0. i, 0. - 0.0748417 i, 0. + 0.096225 i, 0. + 0. i, 0. - 0.096225 i, 0. + 0.0748417 i}
```

Chapter 13: More about finite element methods

■ Section 13.1: Implementation of finite element methods

In this section, we do more than just explain *Mathematica* commands, as has been our policy up to this point. We define a collection of *Mathematica* functions that implement piecewise linear finite elements on 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 given 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 finite element code provided consists of about a dozen commands, all of which are defined in the file fempack. Therefore, before using these commands, you must read in the file (recall the earlier comments about the **Directory** and **SetDirectory** commands):

```
In[663]:= SetDirectory["/Users/msgocken/books/pdebook2/tutorial/mathematica"]

Out[663]= /Users/msgocken/books/pdebook2/tutorial/mathematica
```

```
In[664]:= << fempack
```

Each command is documented using the "usage" mechanism. This mechanism defines a message that will be displayed when the ? operator is used. Here is a simple example:

```
In[665]:= ClearAll[f]
          f[x_] = Sin[x]
          f::usage = "f is another name for the sine function."

Out[666]= Sin[x]

Out[667]= f is another name for the sine function.
```

```
In[668]:= ?f

f is another name for the sine function.
```

Describing a mesh

The main commands are **stiffness** and **load**, which assemble the stiffness and load vectors for the BVP

$$-\nabla\cdot(a(x,y)\nabla u) = f(x,y) \text{ in } \Omega,$$
$$u = 0 \text{ on } \Gamma_1,$$
$$\frac{\partial u}{\partial n} = 0 \text{ on } \Gamma_2,$$

where Γ_1 and Γ_2 partition the boundary of Ω. However, before any computation can be done, the mesh must be described. Three routines are provided for creating meshes: **rectangleMeshD, rectangleMeshN,** and **rectangleMeshTopD** (these routines differ in the boundary conditions that are assumed). The data structure is described by the dummy command **mesh** (**mesh** is not actually defined, but it does have a usage message):

In[669]:=
```
?mesh
```

A triangulation of a planar, polygonal region is described by the following arrays, which are stored together in a structure T:

T[NodeList]: An Mx2 array of real numbers, the coordinates of the nodes in the mesh.

T[NodePtrs]: An Mx1 array of integers; the ith entry equals the index of the ith node in T[FNodePtrs] if the node is free, and the negative of its index in T[CNodePtrs] if the node is constrained.

T[FNodePtrs]: An NNx1 array of integers, where NN is the number of free nodes. T[FNodePtrs][[i]] is the index of the ith free node in T[NodePtrs].

T[CNodePtrs]: An Kx1 array of integers, where K is the number of constrained nodes. T[CNodePtrs][[i]] is the index of the ith constrained node in T[NodePtrs].

T[ElList]: An Lx3 array of integers, where L is the number of triangular elements. Each row corresponds to one element and contains pointers to the nodes of the triangle in T[NodeList].

T[ElEdgeList]: An 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 edges of the triangle are, in order, those joining vertices 1 and 2, 2 and 3, and 3 and 1.

T[FBndyList]: A Bx2 matrix, where B is the number of free boundary edges (i.e. those not constrained by Dirichlet conditions). Each row corresponds to one edge and contains pointers into T[NodeList], yielding the two vertices of the edge.

For more details, see Section 13.1.1 of 'Partial Differential Equations: Analytical and Numerical Methods' by Mark S. Gockenbach.

The command **rectangleMeshD** creates a regular triangulation of a rectangle of the form $[0, l_x]\, x\, [0, l_y]$, assuming Dirichlet conditions on the boundary.

In[670]:=
```
? rectangleMeshD
```

T=rectangleMeshD[nx,ny,lx,ly] creates a regular triangulation of the rectangle [0,lx]x[0,ly], with nx and ny subdivisions in the x and y directions, respectively. Dirichlet conditions are assumed on the boundary.
For a description of the data structure for T, see ?mesh.

The commands **rectangleMeshN** and **rectangleMeshTopD** create the same mesh, but assuming Neumann conditions on the entire boundary, and mixed boundary conditions (Dirichlet on the top edge, Neumann elsewhere), respectively.

Thus we only provide the means to deal with a single domain shape, a rectangle, and only under three combinations of boundary conditions. To use this code to solve BVPs on other domains, you will have to write code to generate the mesh yourself.

Here is a mesh:

In[671]:=
```
T = rectangleMeshD[4, 4, 1.0, 1.0];
```

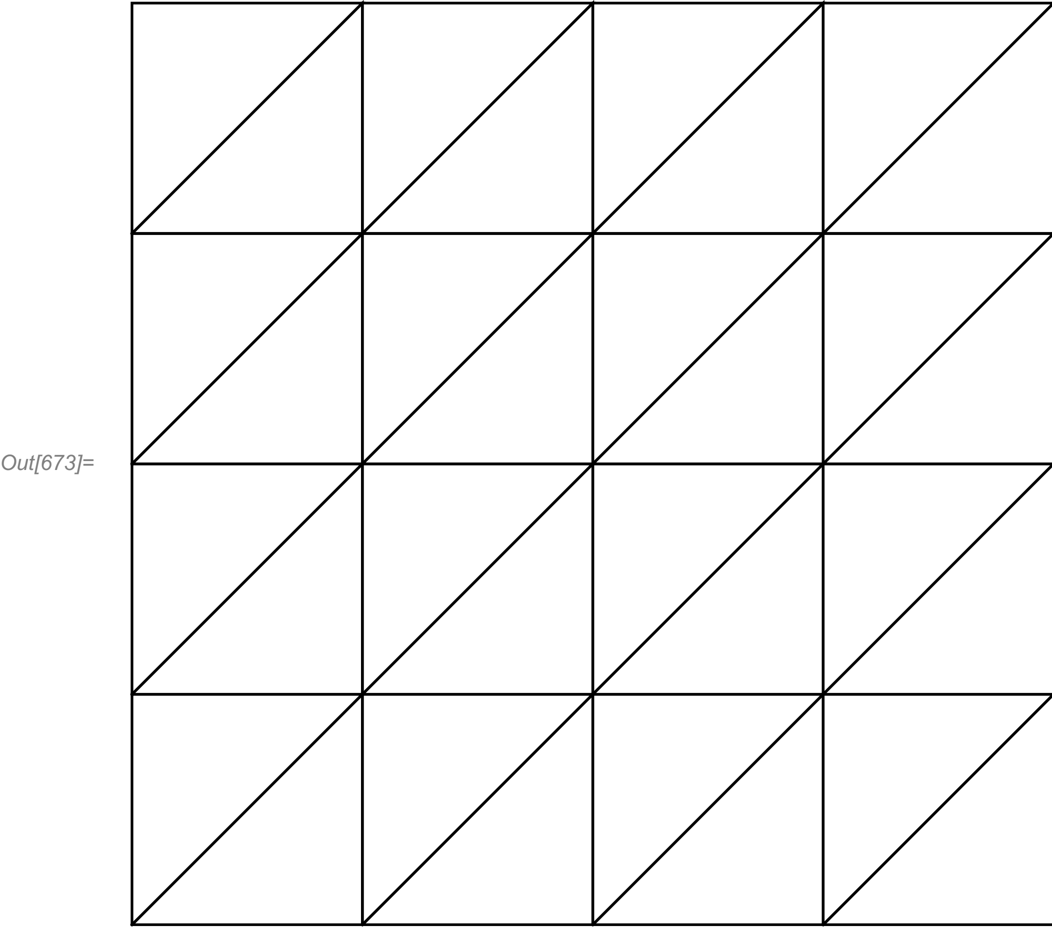
This mesh is shown in Figure 13.1 in the text. The **showMesh** command displays the mesh:

In[672]:=
```
? showMesh
```

showMesh[T] graphs the mesh T. For a description of the data structure for T, see ?mesh.

In[673]:= `showMesh[T]`

Out[673]=



Computing the stiffness matrix and the load vector

Here are the main commands:

In[674]:= `? stiffness`

K=stiffness[T,a] assembles the stiffness matrix for the BVP
−div(a(x,y)grad u)=f(x,y) in Omega, subject to some combination of
homogeneous Dirichlet and Neumann conditions. Omega and the boundary
conditions are defined by the mesh T. For a description of the
data structure for T, see ?mesh.

In[675]:=
```
? load
```

F=load[T,f] assembles the load vector for the BVP
−div(a(x,y)grad u)=f(x,y) in Omega, subject to some combination of
homogeneous Dirichlet and Neumann conditions. Omega and the boundary
conditions are defined by the mesh T. For a description of the
data structure for T, see ?mesh.

Thus, to apply the finite element method to solve the BVP given above, it is necessary to define the coefficient a and the forcing function f. As an example, we reproduce the computations from Example 11.10 from the text, in which case $a(x,y)=1$ and $f(x,y)=x$:

In[676]:=
```
ClearAll[a, f, x, y]
a[x_, y_] = 1
f[x_, y_] = x
```

Out[677]= 1

Out[678]= x

Here is the computation of the stiffness matrix:

In[679]:=
```
ClearAll[K]
K = stiffness[T, a];
MatrixForm[K]
```

Out[681]//MatrixForm=

$$\begin{pmatrix} 4. & -1. & 0. & -1. & 0. & 0. & 0. & 0. & 0. \\ -1. & 4. & -1. & 0. & -1. & 0. & 0. & 0. & 0. \\ 0. & -1. & 4. & 0. & 0. & -1. & 0. & 0. & 0. \\ -1. & 0. & 0. & 4. & -1. & 0. & -1. & 0. & 0. \\ 0. & -1. & 0. & -1. & 4. & -1. & 0. & -1. & 0. \\ 0. & 0. & -1. & 0. & -1. & 4. & 0. & 0. & -1. \\ 0. & 0. & 0. & -1. & 0. & 0. & 4. & -1. & 0. \\ 0. & 0. & 0. & 0. & -1. & 0. & -1. & 4. & -1. \\ 0. & 0. & 0. & 0. & 0. & -1. & 0. & -1. & 4. \end{pmatrix}$$

Next, we compute the load vector:

In[682]:=
```
ClearAll[F]
F = load[T, f];
MatrixForm[F]
```

Out[684]//MatrixForm=

$$\begin{pmatrix} 0.015625 \\ 0.03125 \\ 0.046875 \\ 0.015625 \\ 0.03125 \\ 0.046875 \\ 0.015625 \\ 0.03125 \\ 0.046875 \end{pmatrix}$$

Finally, we solve the system *Ku*=*F* to get the nodal values:

In[685]:=
```
ClearAll[u]
u = LinearSolve[K, F];
MatrixForm[u]
```

Out[687]//MatrixForm=

$$\begin{pmatrix} 0.015904 \\ 0.0273437 \\ 0.0270647 \\ 0.0206473 \\ 0.0351563 \\ 0.0340402 \\ 0.015904 \\ 0.0273438 \\ 0.0270647 \end{pmatrix}$$

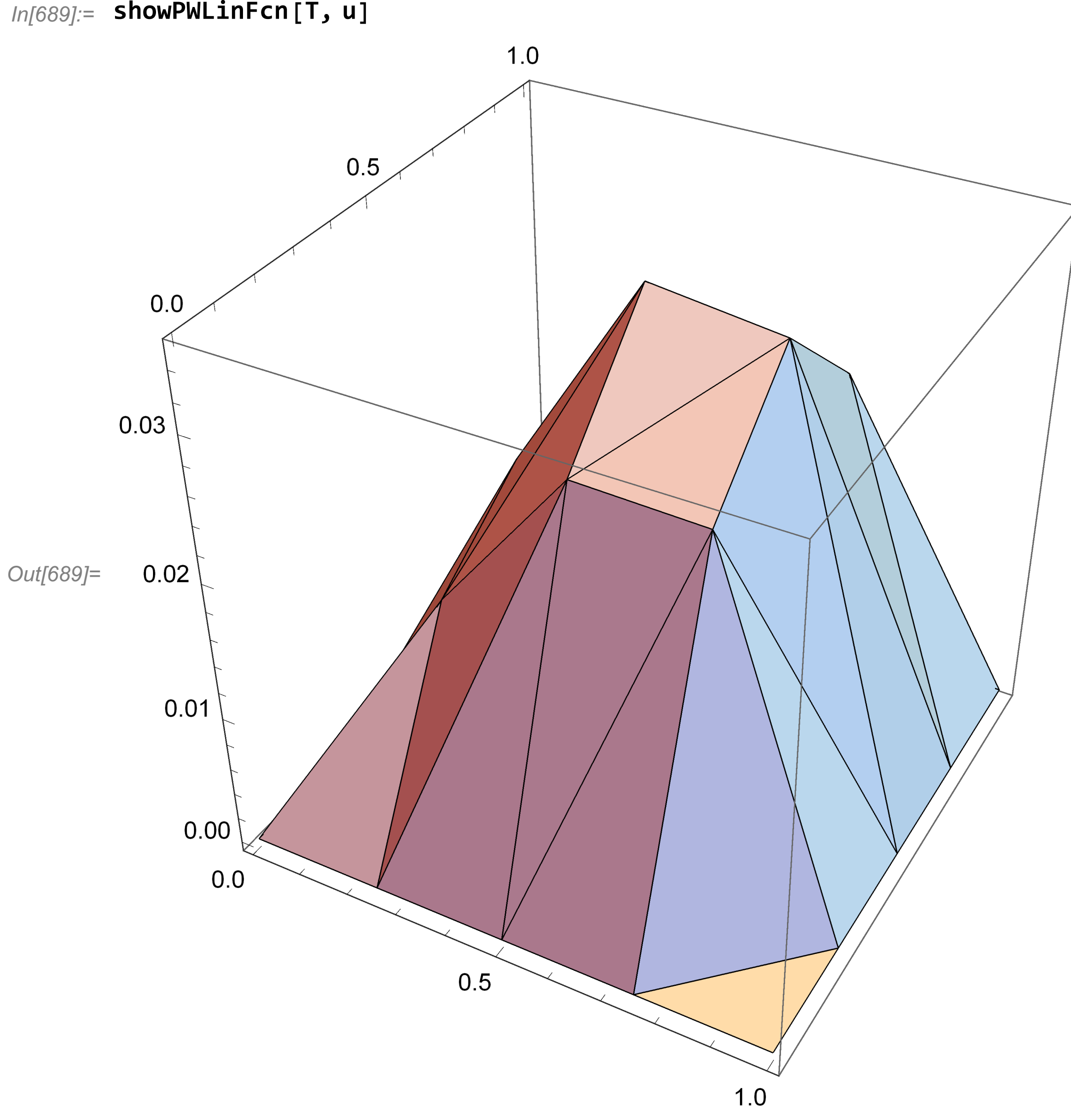
Given the vector of nodal values (and the mesh), you can graph the computed solution using the **showPWLinFcn** command:

In[688]:=
```
?showPWLinFcn
```

showPWLinFcn[T,u] graphs the piecewise linear function defined by the mesh T and the nodal values u. For a description of the data structure for the mesh T, see ?mesh. u must be a vector of N real numbers, where N is the number of free nodes in the mesh T. The nodal values at the constrained nodes are taken to be zero.

In[689]:= `showPWLinFcn[T, u]`

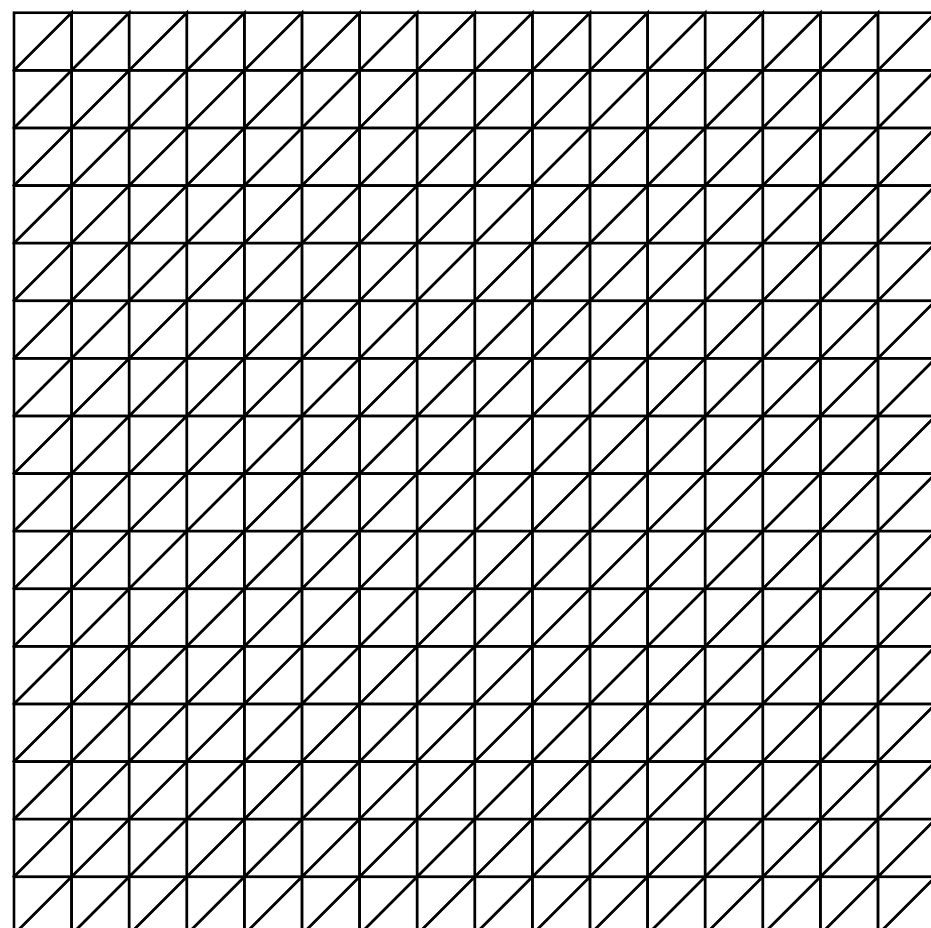
Out[689]=



The above solution does not look very good (not smooth, for instance); this is because the mesh is rather coarse. We repeat the calculation on a finer mesh:

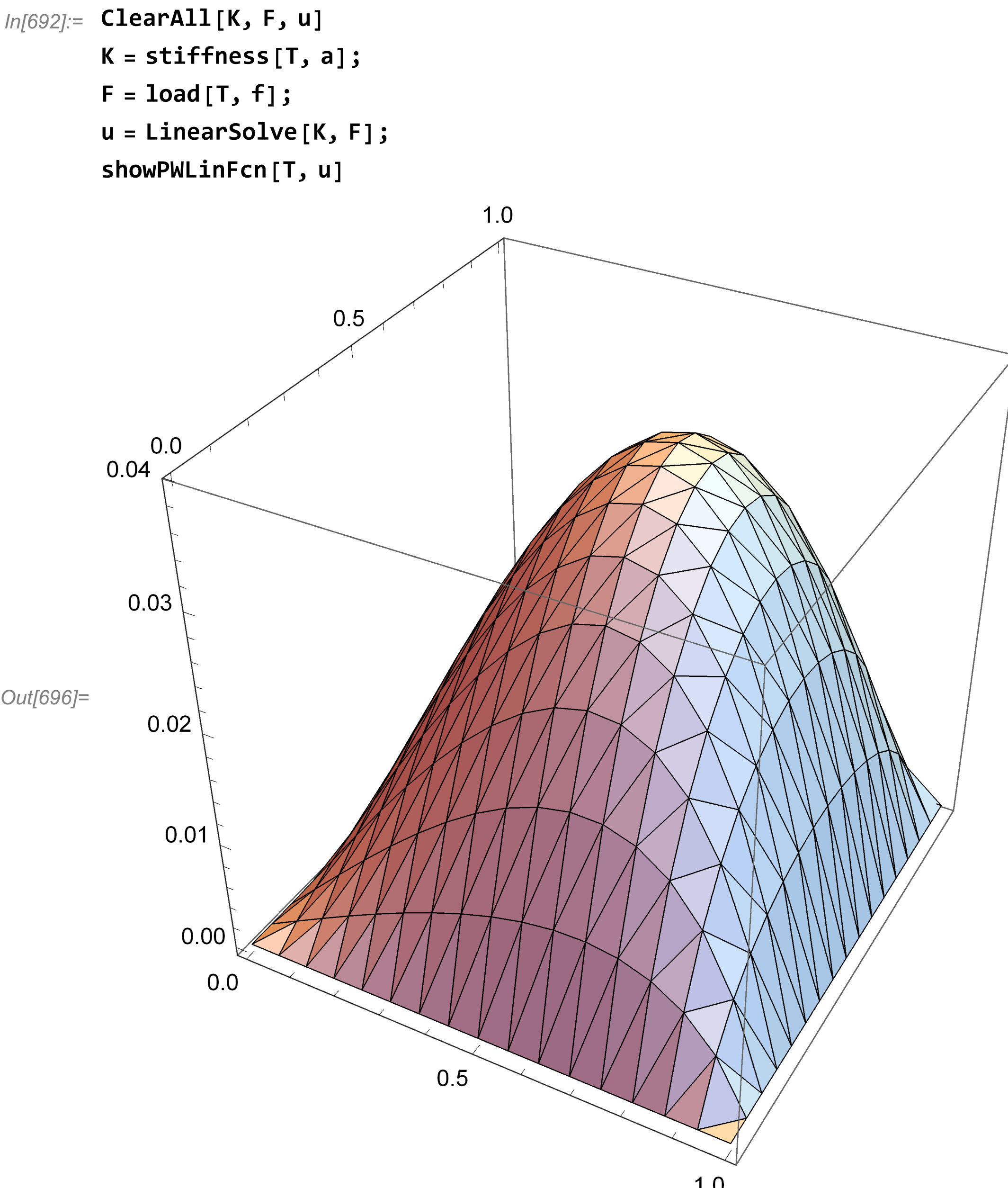
In[690]:=
```
T = rectangleMeshD[16, 16, 1.0, 1.0];
showMesh[T]
```

Out[691]=



In[692]:=
```
ClearAll[K, F, u]
K = stiffness[T, a];
F = load[T, f];
u = LinearSolve[K, F];
showPWLinFcn[T, u]
```

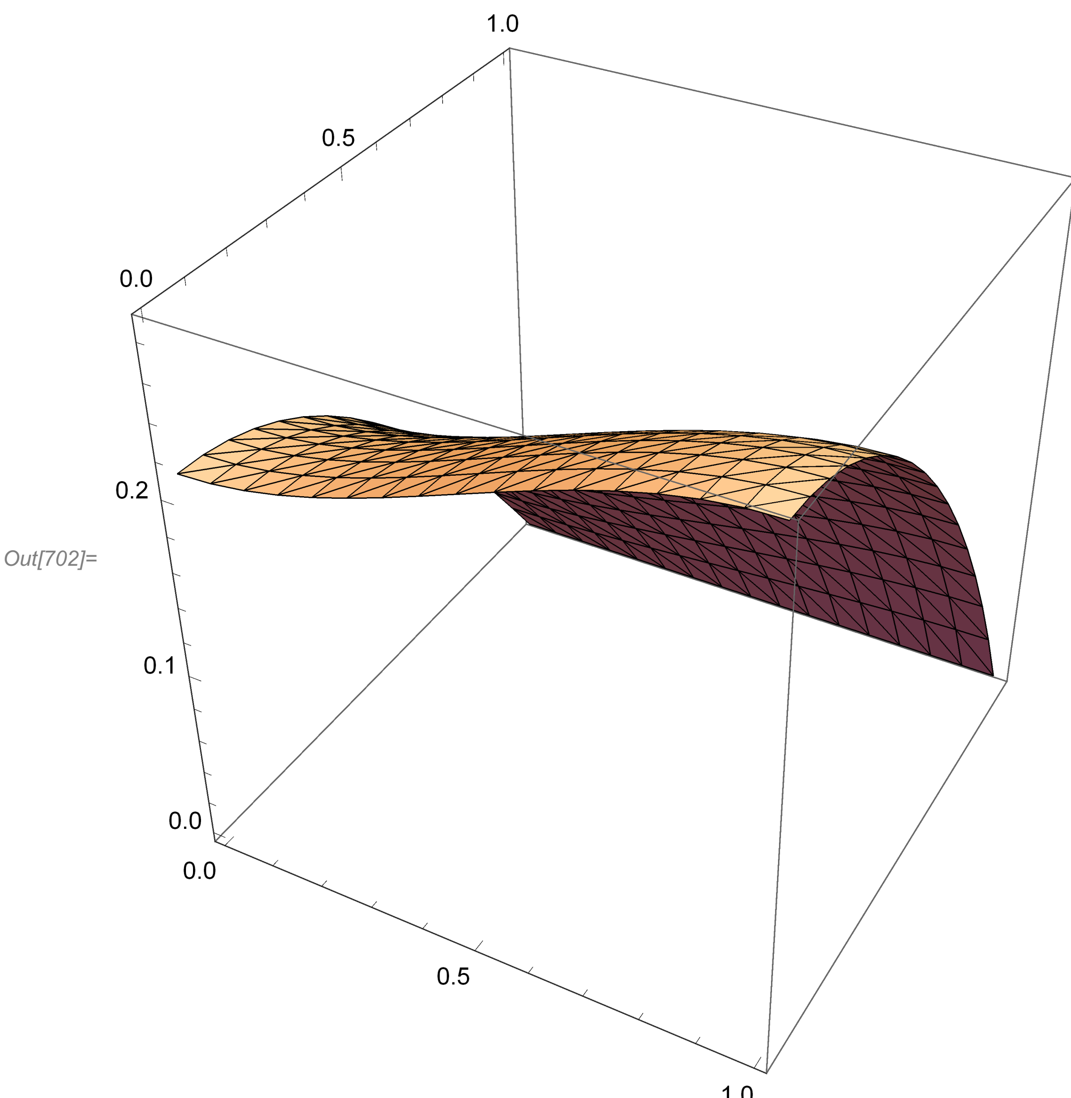
Out[696]=



For the sake of illustrating mixed boundary conditions, we solve the same PDE with mixed boundary conditions:

In[697]:=
```
ClearAll[T1, K1, F1, u1]
T1 = rectangleMeshTopD[16, 16, 1.0, 1.0];
K1 = stiffness[T1, a];
F1 = load[T1, f];
u1 = LinearSolve[K1, F1];
showPWLinFcn[T1, u1]
```

Out[702]=



Testing the code

To see how well the code is working, we solve a problem whose solution is known, and compare the computed solution with the exact solution. We can easily create a problem with a known solution; we just choose $a(x,y)$ and any $u(x,y)$ satisfying the boundary conditions, and then compute

$$-\nabla\cdot(a(x, y)\nabla u)$$

to get the right-hand side function $f(x,y)$. For example, suppose we choose the following functions a and u:

```
In[703]:= ClearAll[a, u, x, y]
          a[x_, y_] = 1 + x^2
          u[x_, y_] = x (1 - x) Sin[Pi y]
Out[704]= 1 + x^2
Out[705]= (1 - x) x Sin[π y]
```

(notice that u satisfies homogeneous Dirichlet conditions on the unit square). Then we compute f as follows:

```
In[706]:= ClearAll[f, x, y]
          f[x_, y_] = Simplify[-D[a[x, y] × D[u[x, y], x], x] - D[a[x, y] × D[u[x, y], y], y]]
Out[707]= -(-2 - (-2 + π^2) x + (-6 + π^2) x^2 - π^2 x^3 + π^2 x^4) Sin[π y]
```

Now we create a coarse mesh and compute the finite element approximate solution:

In[708]:=
```
ClearAll[T, K, F, U]
T = rectangleMeshD[2, 2, 1.0, 1.0];
K = stiffness[T, a];
F = load[T, f];
U = LinearSolve[K, F];
showPWLinFcn[T, U]
```

Out[713]=

(The mesh is so coarse that there is only one free node!)

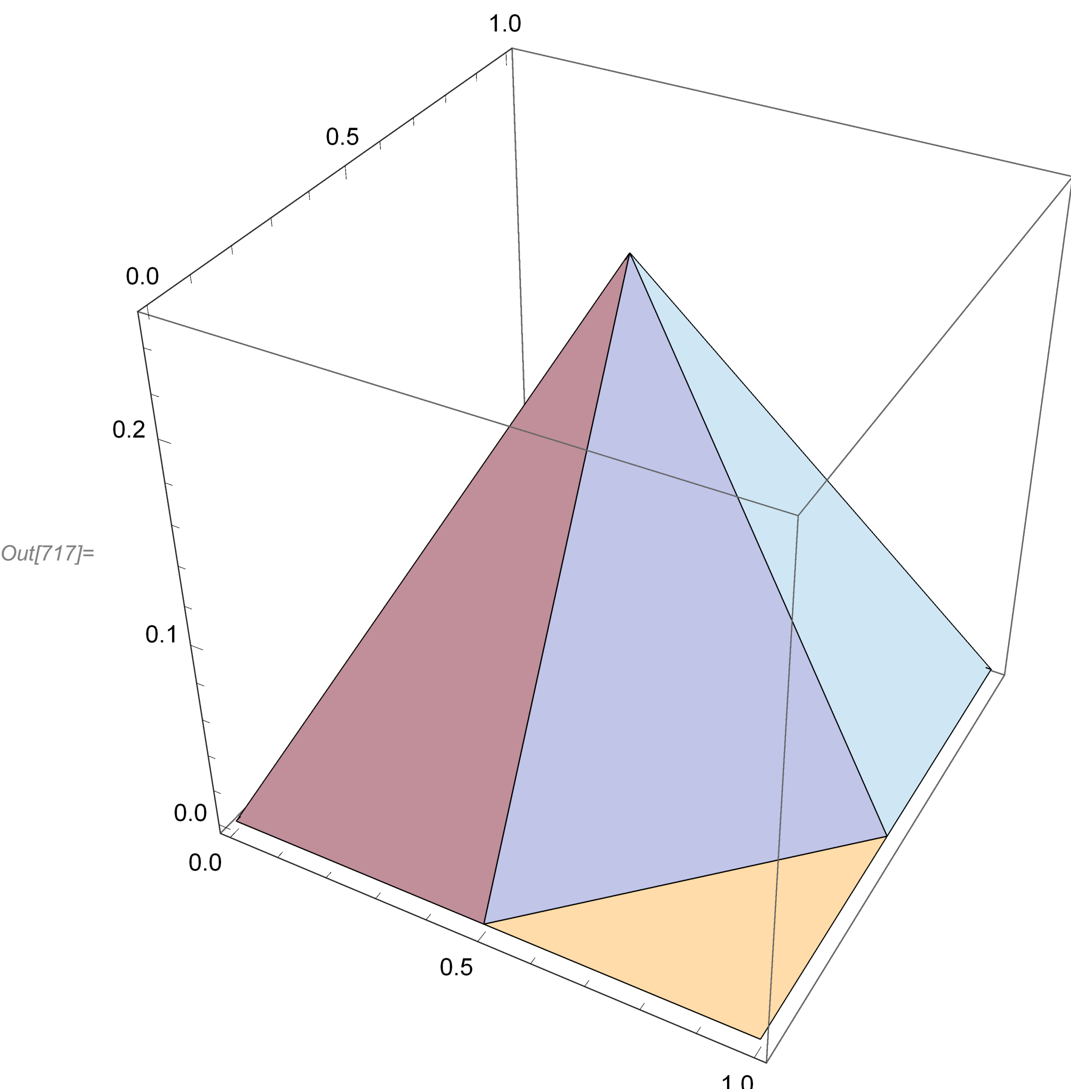
For comparison purposes, let us compute the nodal values of the exact solution on the same mesh. The command **nodalValues** does this:

In[714]:=
```
? nodalValues
```

v=nodalValues[T,f] sets v equal to the vector of values of f at
the free nodes of the mesh T. See ?mesh for a description of the
data structure for T.

In[715]:=
```
ClearAll[V]
V = nodalValues[T, u];
showPWLinFcn[T, V]
```

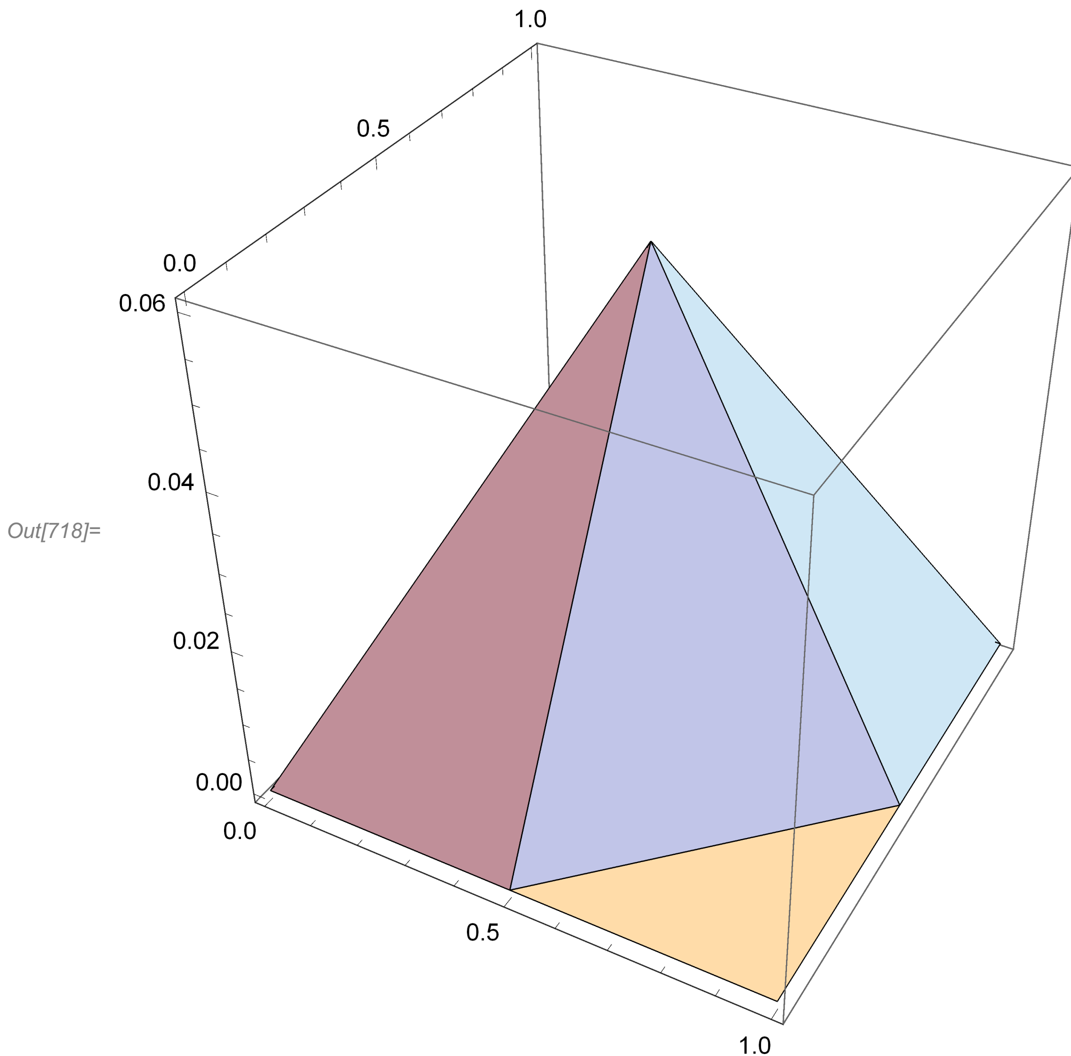
Out[717]=



To compare the two more directly, we plot the difference:

In[718]:= `showPWLinFcn[T, V - U]`

Out[718]=

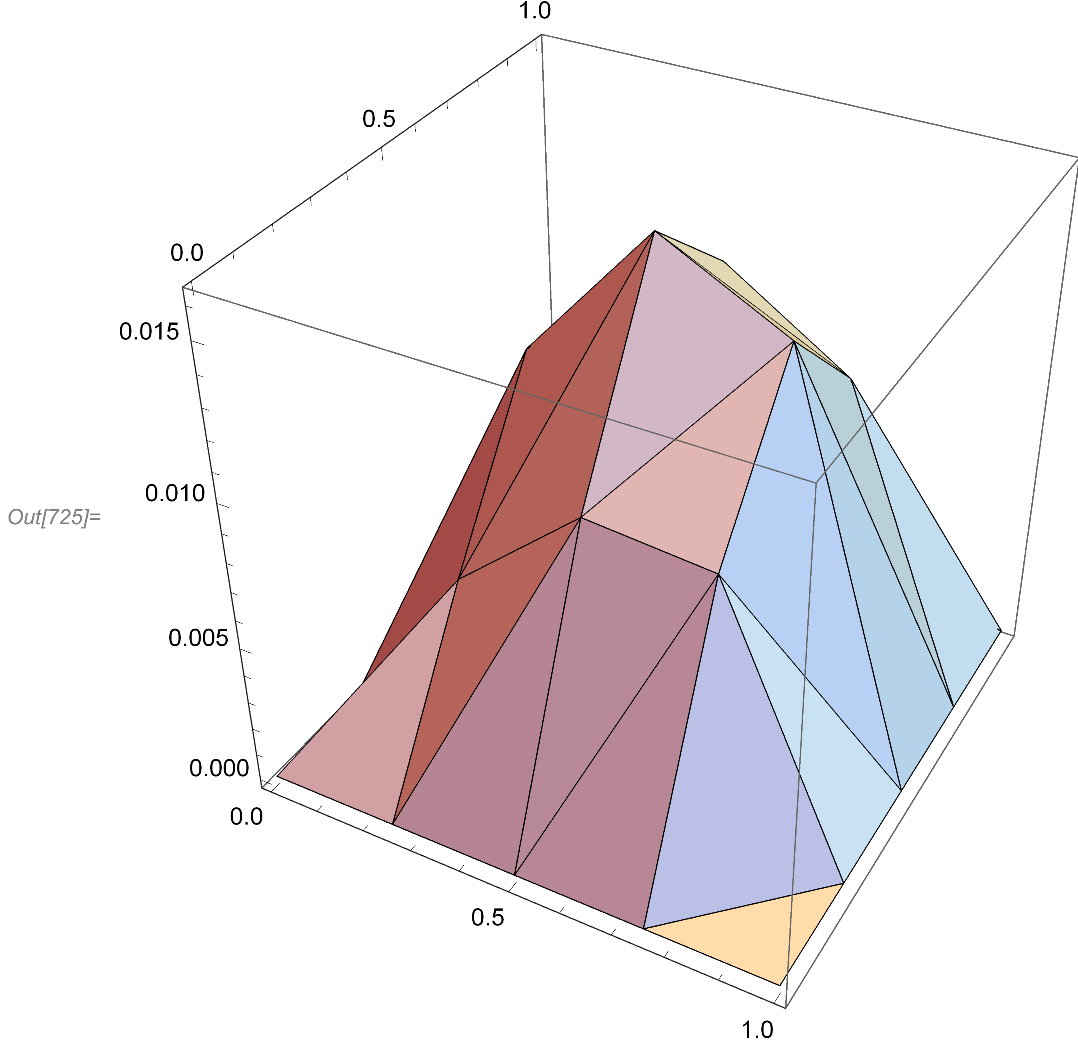


(Notice the scale on the vertical axis.)

We now repeat with a series of finer meshes:

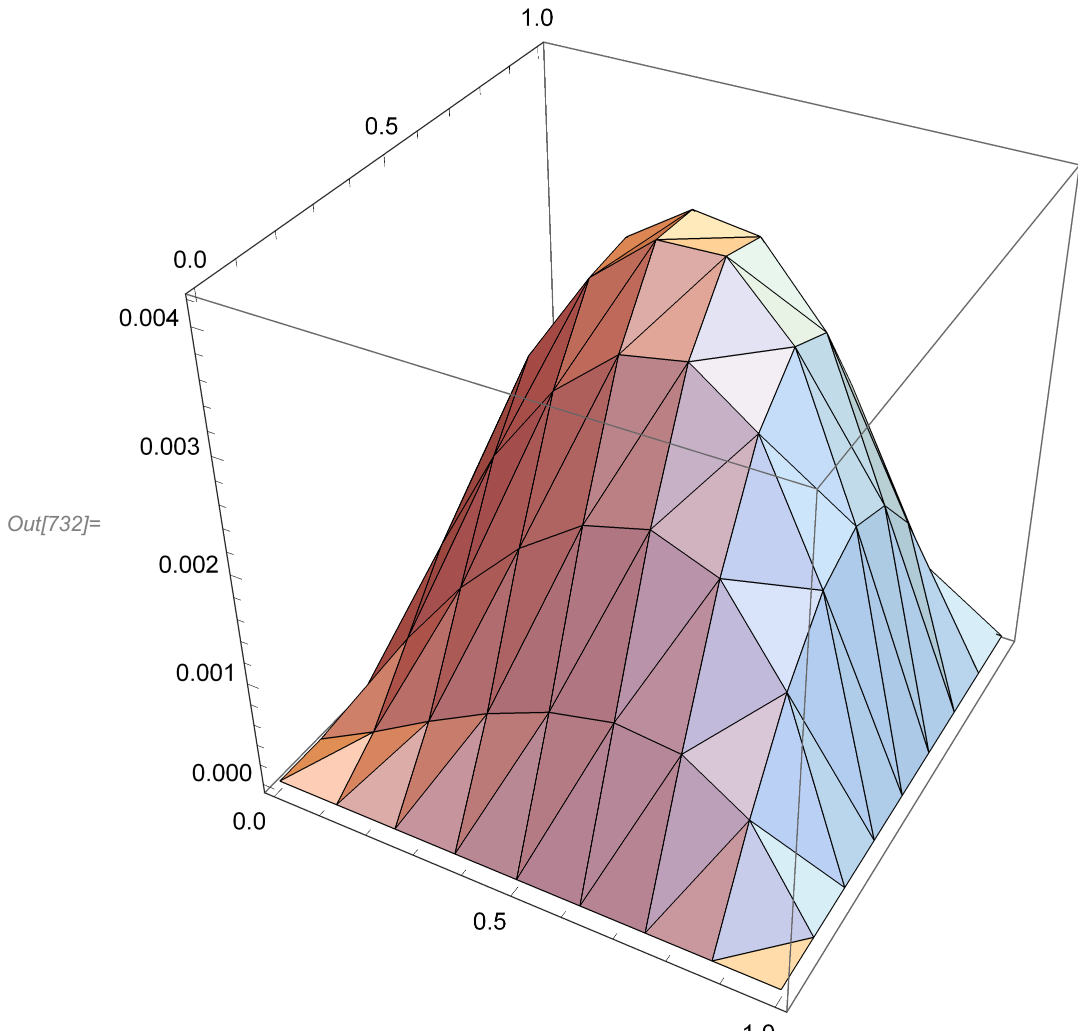
In[719]:=
```
ClearAll[T, K, F, U, V]
T = rectangleMeshD[4, 4, 1.0, 1.0];
K = stiffness[T, a];
F = load[T, f];
U = LinearSolve[K, F];
V = nodalValues[T, u];
showPWLinFcn[T, V - U]
```

Out[725]=



In[726]:=
```
ClearAll[T, K, F, U, V]
T = rectangleMeshD[8, 8, 1.0, 1.0];
K = stiffness[T, a];
F = load[T, f];
U = LinearSolve[K, F];
V = nodalValues[T, u];
showPWLinFcn[T, V - U]
```

Out[732]=



In[733]:=
```
ClearAll[T, K, F, U, V]
T = rectangleMeshD[16, 16, 1.0, 1.0];
K = stiffness[T, a];
F = load[T, f];
U = LinearSolve[K, F];
V = nodalValues[T, u];
showPWLinFcn[T, V - U]
```

Out[739]=

1.0
0.5
0.0
0.0010
0.0005
0.0000
0.0
0.5
1.0

These plots show that the solution becomes increasingly accurate as the mesh is refined (be sure to compare the vertical scaled on the plots).

Using the code

The purpose of providing this code is so that you can see how finite element methods are implemented in practice. To really benefit from the code, you should study it and 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 **mass** that computes the mass matrix. The calling sequence should be simply M=mass[T], where T is the triangulation.

2. Choose some other geometric shapes and/or combinations of boundary conditions and write mesh generation routines analogous to **rectangleMeshD** and **rectangleMeshTopD**.

3. Extend the code to handle inhomogeneous Dirichlet conditions. Recall that such boundary conditions change the load vector, so the routine **load** 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 13.1.4 of the textbook.

As mentioned above, the mesh data structure described in **mesh** includes the information necessary to solve exercises 3, 4, and 5.

■ Section 13.2: Solving sparse linear systems

In this section, we briefly mention *Mathematica*'s functionality for working with sparse matrices; specifically, it has a data structure for representing a sparse matrix and its linear algebra commands can be applied to sparse matrices. A sparse matrix is stored as a **SparseArray** object.

Representing a sparse matrix

One way to create a sparse matrix is by converting an ordinary matrix to sparse format:

```
In[740]:= ClearAll[A, B]
          A = {{1, 0, 0, 0}, {1, 2, 0, 0}, {0, 0, 3, 4}, {0, 0, 0, 4}};
          B = SparseArray[A]

Out[742]= SparseArray[ Specified elements: 6
                       Dimensions: {4, 4} ]
```

```
In[743]:= MatrixForm[B]

Out[743]//MatrixForm=
 1 0 0 0
 1 2 0 0
 0 0 3 4
 0 0 0 4
```

Another way to define a sparse matrix is to specify its nonzero entries using transformation rules of the form $(i, j) \to a_{ij}$.

```
In[744]:= ClearAll[M]
          M =
           SparseArray[{{1, 1} → 1, {2, 1} → 1, {2, 2} → 2, {3, 3} → 3, {3, 4} → 4, {4, 4} → 4}]
          MatrixForm[M]

Out[745]= SparseArray[ Specified elements: 6
                       Dimensions: {4, 4} ]

Out[746]//MatrixForm=
   1 0 0 0
   1 2 0 0
   0 0 3 4
   0 0 0 4
```

There are more ways to create sparse matrices in *Mathematica*; see the documentation for **SparseArray** for more information.

Solving a sparse system

Mathematica's LinearSolve command will solve a sparse linear system:

```
In[747]:= ClearAll[b, x]
          b = {1, -1, 2, 4}
          x = LinearSolve[B, b]

Out[748]= {1, -1, 2, 4}

Out[749]= {1, -1, -2/3, 1}

In[750]:= B.x - b

Out[750]= {0, 0, 0, 0}
```

Other linear algebra commands also work on sparse matrices:

```
In[751]:= {evals, evecs} = Eigensystem[B]

Out[751]= {{4, 3, 2, 1}, {{0, 0, -4, -1}, {0, 0, 1, 0}, {0, -1, 0, 0}, {-1, 1, 0, 0}}}

In[752]:= Det[B]

Out[752]= 24

In[753]:= B[[1, 1]]

Out[753]= 1
```

Some projects

You may want to do one of the following projects to explore the advantages of sparsity:

1. Write a command to create a sparse, nonsingular matrix with random entries. The command should return the same matrix stored in both dense matrix form (i.e. as a list of row vectors, not taking advantage of the fact that many

entries are zero) and sparse matrix form (i.e. as a list of transformation rules, listing the nonzeros entries). Then compare the efficiency of **LinearSolve** when applied to the same matrix in the different formats.

2. Rewrite the **stiffness** command in the fempack code to create the stiffness matrix in sparse format rather than dense form.