Reprint from
The *Mathematica* Conference

June, 1992   Boston, MA

876.15.10.1992

Numerical Computation I

Jerry Keiper

1 Introduction

The area of numerical computation tends to break somewhat naturally into three subareas which, for want of better terms, we will call sampling, linear algebra, and theory. This is a gross oversimplification and these terms are not very descriptive, but they are useful labels for our discussion here. The course Numerical Computation II covers some of the theoretical aspects of numerical computation. This course talks about numerical methods that are based on sampling and linear algebra.

Before we get going we need to briefly talk about arithmetic in *Mathematica* and a few functions that relate to arithmetic. This material is covered in depth in Numerical Computation II, but we need some of the basics to be able to talk about the topics in this course.

Mathematica uses three different kinds of arithmetic: correct arithmetic and two different types of incorrect arithmetic. There is little to say about the correct arithmetic other than that you get correct results when you use it. Arithmetic with integers and rational numbers is correct and you can rely on the result being correct. The other two types of arithmetic make deliberate errors for reasons of efficiency. To see why this is expedient consider multiplying $1.2345678 \times 1.2345678$. The correct result is 1.52415765279684, but no calculator that I know of will give you that answer. Multiplication almost always increases the number and the number of digits soon becomes burdensome and for most purposes unimportant. Even worse is division: $10/3$ should be $3.33333333333\ldots$ where the decimal expansion is infinitely long.

Clearly compromises need to be made. *Mathematica* makes two different types of compromises resulting in two different types of incorrect arithmetic. One way to compromise is to do the arithmetic as fast as possible while maintaining a certain standard of correctness. This is achieved by using the arithmetic that comes with the computer, either built into the hardware or done very efficiently in the software provided. Since this arithmetic is not under the control of *Mathematica* we refer to it as machine arithmetic. Likewise, numbers used in this arithmetic, which are represented in a format directly accessible by the machine, are called machine numbers.

Although machine arithmetic is very fast, it has its limitations. For one thing the numbers are of limited magnitude and precision. If you want to use very large numbers or more than say 20 digits, machine arithmetic will simply not work. The alternative is to use bignum arithmetic. Bignum arithmetic allows you to work with numbers as large as $10^{323228010}$ or as small as $10^{-323228015}$. It also allows you to work with numbers that are many thousands of digits long. The disadvantage of bignum arithmetic is that it is done in software and is much slower than machine arithmetic.

An additional feature of bignum arithmetic is that by default it always produces results that contain only those digits that are significant based on the significant digits of the operands. This is a mixed blessing because although it relieves the user of the burden of analyzing the size of the error in the result, it is often overly pessimistic[1].

The numbers used by bignum arithmetic are called bignums. The number of significant digits in a bignum is known as its *precision*[2]. Likewise the *accuracy* of a bignum is the position to the right of the decimal of the least significant digit. Thus the number 123.45 has precision 5 and accuracy 2. The functions `Precision[ ]` and `Accuracy[ ]` give the precision and accuracy of any bignum. For machine numbers, since *Mathematica* does not keep track of which digits are significant, the precision is simply the number of digits in any machine number and accuracy is precision minus magnitude, where magnitude is the number of digits to the left of the decimal.

[1] There is a way to get around this feature. Numerical Computation II discusses how.

[2] The precision of a number is actually *not* defined by counting digits. The precision is really defined in terms of a bound on the relative error in the number. See Numerical Computation II for details.

You can get a number with a different precision or accuracy by applying `SetPrecision[ ]` or `SetAccuracy[ ]` to a number. These functions do not change the original number (destructive operations in *Mathematica* are prohibited), but return a new number with altered precision or accuracy.

Make a number approximately equal to 10/3 with 30 digits precision.

```
In[1]:= SetPrecision[10/3, 30]

Out[1]= 3.33333333333333333333333333333
```

Make another number with the same approximate value, but 10 digits accuracy.

```
In[2]:= SetAccuracy[%, 10]

Out[2]= 3.3333333333
```

You should realize that for technical reasons bignums generally include about 10 digits beyond the last significant digit. These digits are included in calculations, but they are not considered significant. There is a field in the data structure of a bignum that specifies where the significant digits end and the insignificant digits begin. If you start with a bignum and apply `SetPrecision[ ]` or `SetAccuracy[ ]` to it all that happens is the data structure is copied over (to avoid destroying the original copy) and the the field specifying the boundary between significant and insignificant digits gets altered. If you start with a number that is not already a bignum *Mathematica* creates a bignum with the appropriate precision and accuracy.

The functions `SetPrecision[ ]` and `SetAccuracy[ ]` only work on numbers, although they do recursively go inside of complicated expressions and do their thing on all numbers that they encounter. But some expressions are not numbers even though they may represent numerical quantities. For example `Pi` is not a number in *Mathematica*; it is simply a symbol. If you enter `SetPrecision[Pi, 20]` you just get `Pi` back. To get a numerical approximation to π you need to use the function `N[ ]`. `SetPrecision[ ]` and `SetAccuracy[ ]` are very simple in their operation: any numbers they encounter in an expression are simply converted to bignums. Of course once numbers become bignums the evaluator often takes over and makes further changes until nothing more can be done, but the operation of `SetPrecision[ ]` and `SetAccuracy[ ]` is quite simple. The function `N[ ]` on the other hand is more complicated. There are internal rules for what `N[ ]` should do with certain symbols like `Pi` and `Integrate`. In addition you can add your own rules to symbols describing what `N[ ]` should do with them.

Another major difference between `N[ ]` and `SetPrecision[ ]` is that `N[ ]` will never raise the precision of anything and may give a result with less precision than was requested. `SetPrecision[ ]` gives whatever precision is specified, even if it has to use digits that are incorrect.

Define `a` to be the difference of two nearly equal quantities.

```
In[1]:= a = Sin[1] - 295031332/350613791

             295031332
Out[1]= -(---------) + Sin[1]
             350613791
```

The argument of `Sin[ ]` becomes a bignum and then `Sin[ ]` evaluates to a number. Since the difference is small some 18 digits are lost due to cancelation.

```
In[2]:= N[a, 30]

                          -18
Out[2]= 2.661305922855 10
```

Here again digits are lost to cancelation, but the precision of the result is set to 30 before returning.

```
In[3]:= SetPrecision[a, 30]

                                            -18
Out[3]= 2.66130592285544304993174873850 3 10
```

Here we see the correct result to 32 digits. Again some 18 digits were lost, but we started with 50 this time. We see that about 10 of the extra digits in the previous result were in fact correct, but the last 8 were wrong.

```
In[4]:= N[a, 50]

                                              -18
Out[4]= 2.6613059228554430499317535512545 8 10
```

The function `N[ ]` can be used with a single argument. If you don't specify the precision to use or you specify a value that is less than the precision of machine numbers (*i.e.*, `$MachinePrecision`), `N[ ]` will attempt to return machine numbers when it encounters numerical quantities. There is a field in the data structure for machine numbers that specifies how many digits to print. This is quite different from the precision of a number. The precision of any nonzero machine number is just `$MachinePrecision` and is fixed by the manufacturer of the computer. The number of digits to be printed for a machine number is determined by the number of digits entered when the number was typed in or the second argument to `N[ ]` if the number was created by `N[ ]`. If the number was created by any arithmetic operation it doesn't know how many digits to print and so by default prints 6. In any case trailing zeros, even if they are significant, are not included.

Form a table of machine numbers displayed with various numbers of digits.

```
In[1]:= a = Table[N[Pi, k], {k, 5}]

Out[1]= {3., 3.1, 3.14, 3.142, 3.1416}
```

Each of those machine numbers is the same value. The only difference is the number of digits to be printed.

```
In[2]:= a - N[Pi]

Out[2]= {0., 0., 0., 0., 0.}
```

Even the simplest arithmetic operation causes the information on the number of digits to be printed to be lost. In such cases 6 digits are displayed.

```
In[3]:= -a

Out[3]= {-3.14159, -3.14159, -3.14159, -3.14159, -3.14159}
```

2 Numerical Sampling

There are many mathematical operations that are inherently infinite in nature: limits, infinite sums, and Riemann integrals. Even a derivative is defined as a limit of a difference quotient. Because these operations are essentially infinite it is not possible to perform them numerically; the best we can do is evaluate some finite analogue that is designed to have a relatively small error in most practical situations. Of course *Mathematica* can sometimes actually do the operation symbolically and it is generally recommended that you make use of this ability when possible[3].

The main point of the mathematical concept of a limit is to rigorously define things that give problems otherwise. For example, the function $\sin x/x$ is undefined at $x = 0$, but by taking the limit as x approaches 0 it is possible to give a value to the function $x = 0$ in a natural way. Now if it were possible to deal with such problems in a finite way then it would also be possible to design algorithms to numerically perform infinite numerical operations that would be rather foolproof. In fact such is not the case; each of the *Mathematica* functions we will discuss here can give incorrect answers. Of course such behavior is designed to be rather rare: there is a tradeoff between efficiency and robustness. It is also the case that by choosing the options to these functions appropriately you can almost always get the correct answer.

Before we go on to look in detail at some of these functions let us look more closely at how they might fail by considering an example with the function `Plot`.

Form a list of the points that `Plot` initially considers.

```
In[1]:= xlist = { }; Plot[AppendTo[xlist, x];
                    0, {x, 0, 4}, Compiled -> False,
                    DisplayFunction -> Identity];
```

Form a polynomial with zeros at the points in `xlist`.

```
In[2]:= p = Times @@ (xlist - x)

Out[2]= (0. - x) (0.166667 - x) (0.333333 - x) (0.5 - x)
   (0.666667 - x) (0.833333 - x) (1. - x) (1.16667 - x)
   (1.33333 - x) (1.5 - x) (1.66667 - x) (1.83333 - x) (2. - x)
   (2.16667 - x) (2.33333 - x) (2.5 - x) (2.66667 - x)
   (2.83333 - x) (3. - x) (3.16667 - x) (3.33333 - x) (3.5 - x)
   (3.66667 - x) (3.83333 - x) (4. - x)
```

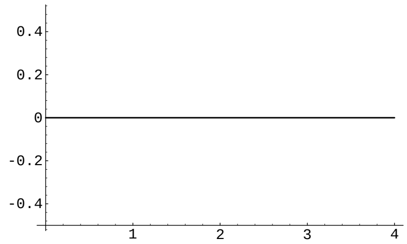
The polynomial `p` is certainly not identically 0.

```
In[3]:= p /. {{x -> .1}, {x -> 1.9}}

Out[3]= {-872.533, 0.00251452}
```

[3] There are exceptions to this, *e.g.*, a finite sum with 10^8 terms may be just too big or take too long to be of practical use.

But its plot is.

```
In[4]:= Plot[p, {x, 0, 4}, AxesOrigin -> {0,-.5}]
```

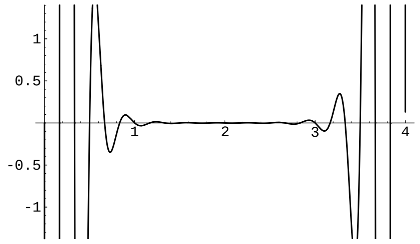


The problem here is similar to what can happen in any algorithm that only samples a few points: based on the sampled points, *Mathematica* incorrectly assumes that the behavior of the function is much simpler than what it actually is.

Of course good software will try to avoid such errors. If we change the plot just slightly it is able to detect that the function is *not* identically zero.

This is a reasonably good plot of the true graph. Of course the extrema of the function vary widely so at the ends it gets clipped and in the middle we can't even see that variation, but this is a mathematical phenomenon rather than a software bug.

```
In[5]:= Plot[p, {x, 0, 3.999999}]
```



3 Numerical Quadrature

One of the problems with symbolic algebra programs is that, while they may be powerful, their power often results in huge answers. Some of the most common examples of this occur as the results of integration. If the value of an integral requires tens of pages to be expressed it is for most purposes not useful. And of course there are many definite integrals for which there simply is no "closed form" for the value. An alternative to symbolic definite integration is numerical quadrature[4]. Here we explain how *Mathematica* does quadrature and how you can make `NIntegrate[ ]` more effective by appropriately choosing its options.

■ 3.1 Integrate[ ] or N[Integrate[ ]]

There are several ways to do definite integration in *Mathematica*: `Integrate[ ]`, `N[Integrate[ ]]`, `NIntegrate[ ]`, and `Integrate[N[ ], ]`. Because there are so many different ways to evaluate integrals in *Mathematica* you may wonder what is the best way for a particular problem and what each of them actually does.

The primary difference between `Integrate[ ]` and `NIntegrate[ ]` is that while `Integrate[ ]` tries to evaluate the integral symbolically much as you might do by hand, `NIntegrate[ ]` works by sampling points in the region of integration and taking a weighted average of the values given by the integrand at

[4]The term "numerical integration" is generally reserved for the numerical solution of an ordinary differential equation. Although `NIntegrate[ ]` can be used to simulate indefinite integration by repeatedly evaluating definite integrals with varying limits of integration it will be *much* slower than is necessary. For indefinite integration you should write the problem as an ordinary differential equation and use `NDsolve[ ]` since this works incrementally rather than repeatedly evaluating the integral over nearly the same interval.

those points. If `Integrate[ ]` is unable to find the value of the integrand in closed form it will return itself. If you want a value for the integral you may still be able to get a numerical approximation to it by applying `N[ ]` to `Integrate[ ]`, but *Mathematica* will first re-attempt to evaluate `Integrate[ ]`. Eventually `NIntegrate[ ]` will be used (assuming the integral could not be evaluated symbolically). However, since `N[ ]` allows no options, only the options `WorkingPrecision`, `AccuracyGoal`, and `PrecisionGoal` of `NIntegrate[ ]` can be controlled even indirectly; the rest just use default values. If you want any degree of control over the options of `NIntegrate[ ]` you should use `NIntegrate[ ]` directly rather than through `N[Integrate[ ], ]`.

The form `Integrate[N[ ], ]` should be used very rarely if at all. Forcing the integrand to be approximate will usually only cause problems for the symbolic maniplations required to evaluate the integral.

■ 3.2 NIntegrate

Several of the numerical operations have many options. `NIntegrate[ ]` is one such function; it has nine different options. While default values have been given for them that are reasonable for many problems, significant efficiency improvements can often be achieved by changing them for specific problems. This subsection is presented in an effort to help you better understand how to use `NIntegrate[ ]`.

WorkingPrecision, AccuracyGoal, and PrecisionGoal

The three options `WorkingPrecision`, `AccuracyGoal`, and `PrecisionGoal` are used for several other numerical operations in addition to `NIntegrate[ ]`. All three deal with the arithmetic in some way.

`WorkingPrecision` is simply the number of digits used in the arithmetic as the integrand is evaluated. In particular, the integrand must be evaluated at many different values of the integration variable. The value of the integration variable will have precision `WorkingPrecision` when the integrand is evaluated. If bignum arithmetic is being used, *i.e.*, if `WorkingPrecision` is greater than machine precision, the precision of the evaluated integrand may be more or less than the precision of the integration variable; it just depends on how ill-conditioned the integrand is. If machine arithmetic is being used, you will not get control of the amount of error in evaluating the integrand at any particular point (except inasmuch as you can find a well-conditioned algorithm to evaluate it), but you will get a significant improvement in speed. The default for `WorkingPrecision` is to use machine precision.

Using machine precision you get a machine-precision answer.

```
In[6]:= NIntegrate[Sin[x],{x,0,Pi}] - 2

                    -16
Out[6]= 4.44089 10
```

Giving `WorkingPrecision` a value of 30 causes bignum arithmetic to be used.

```
In[7]:= y = NIntegrate[Sin[x],{x,0,Pi},
                         WorkingPrecision->30]

Out[7]= 2.
```

With bignum arithmetic the accuracy of the result is reduced to what is justified by the estimate of the error.

```
In[8]:= Precision[y]

Out[8]= 20
```

The actual error often is smaller than the estimate of the error.

```
In[9]:= SetPrecision[y,40] - 2

                -39
Out[9]= -2.9 10
```

`AccuracyGoal` and `PrecisionGoal` are used to specify how much error you are willing to accept in the result. `AccuracyGoal` specifies the accuracy desired in the result and `PrecisionGoal` specifies the precision desired in the result, but `NIntegrate[ ]` will return its result as soon as *either* goal is satisfied. If you don't care about the accuracy of the result, but want to get the answer correct to four significant digits, you can set `AccuracyGoal -> Infinity` (ensuring that it will never be achieved) and set `PrecisionGoal -> 4`. The default for `AccuracyGoal` is `Infinity` and the default for `PrecisionGoal` is `Automatic`, *i.e.*, *Mathematica* stops when the estimated relative error is less than $10^{-(w-10)}$ where w is the `WorkingPrecision`[5].

It really is necessary to have two different ways to specify the acceptable error. To show why this is so we consider what it would be like to have only one way. If we had only `PrecisionGoal`, `NIntegrate[ ]` would fail to converge for any integral that had an exact value of 0. For example:

Mathematica makes a valiant effort, but fails to achieve convergence.

```
In[10]:= Timing[ NIntegrate[1/x^2 - 1/6, {x,2,3},
                        AccuracyGoal -> Infinity,
                        PrecisionGoal -> 6]]

NIntegrate::ploss:
   Numerical integration stopping due to loss of precision.
     Achieved neither the requested PrecisionGoal nor
     AccuracyGoal; suspect one of the following: highly
     oscillatory integrand or the true value of the integral is
     0.

Out[10]= {0.25 Second, 0.}
```

The `AccuracyGoal` is achieved quite quickly. Note that the result (which is all error) is quite small.

```
In[11]:= Timing[ NIntegrate[1/x^2 - 1/6, {x,2,3},
                         AccuracyGoal -> 6,
                         PrecisionGoal -> 6]]

                                         -17
Out[11]= {0.0333333 Second, 1.14925 10   }
```

If we had only `AccuracyGoal`, we would have to have a pretty good idea of the magnitude of the integral in order to specify a reasonable value for it. For example:

We asked for a large accuracy, but we cannot be sure that this answer is correct.

```
In[12]:= NIntegrate[Exp[-x] Sin[x], {x,200,260},
                         AccuracyGoal -> 80,
                         PrecisionGoal -> Infinity]

                     -88
Out[12]= -3.23028 10
```

In fact an `AccuracyGoal` of 80 is not sufficient, but 110 could not have been achieved at all.

```
In[13]:= NIntegrate[Exp[-x] Sin[x], {x,200,260},
                         AccuracyGoal -> Infinity,
                         PrecisionGoal -> 6]

                     -88
Out[13]= -2.67168 10
```

Compiled

The option `Compiled` is also common to many numerical functions in *Mathematica*. It can take only two values: `True`, which is the default, and `False`. `Compiled` specifies whether the integrand is to be compiled. However if the `WorkingPrecision` is more than machine precision no compilation will be performed even if `Compiled` is set to `True` since it would not make sense to do so.

Method

Currently there are four different methods that can be used for quadrature[6]. In each of these methods the basic idea is to evaluate the integrand at several points and approximate the integral by a weighted average of the values of the integrand at the various points. The error in the approximation is estimated by taking another weighted average of the values of the integrand over a different set of points and with different

[5]There is an exception to this. Different machines have different precisions for their machine arithmetic. To have some consistency between machines, when machine arithmetic is being used, the value of `Automatic` for `PrecisionGoal` means that the precision goal is to be six digits.

[6]Quadrature refers to the process of numerically evaluating an integral. The term cubature is used to refer to the process of numerically evaluating a multidimensional integral when the multidimensional aspects are being emphasized.

weights. If the estimate of the error is sufficiently small, the value that is supposed to have less error is returned as the value of the integral.

The different methods are `GaussKronrod`, `DoubleExponential`, `Trapezoidal`, and `MultiDimensional`. The default value for `Method` is `Automatic`, which uses `GaussKronrod` for one dimensional quadrature and `MultiDimensional` for two or more dimensions. The methods `GaussKronrod`, `DoubleExponential`, and `Trapezoidal` can also be used for cubature, but the result is evaluated as a Cartesian product of one dimensional quadratures and is usually much slower than `MultiDimensional`. `MultiDimensional` is not valid for one-dimensional quadrature.

`DoubleExponential` and `Trapezoidal` are fundamentally different from the other two methods of quadrature. `GaussKronrod` and `MultiDimensional` are adaptive; that is, they are able to concentrate their efforts where strange things are happening, *e.g.*, at a discontinuity or some other type of singularity. Although `DoubleExponential` and `Trapezoidal` quadrature will eventually converge even for discontinuous integrands, they will take a *very* long time since they will not try to work only near the discontinuity[7].

`DoubleExponential` quadrature depends on the fact that the integrand is analytic on an open set in the complex plane containing the interval of integration[8]. The idea with `DoubleExponential` quadrature is to apply the trapezoid rule to the problem after a certain reparameterization has occurred. If the estimated error in the approximation to the integral is too large, the stepsize is halved and the trapezoid rule applied again. By halving the stepsize it is easy to incorporate the previous evaluations of the integrand into the approximation to the integral without having to save them explicitly.

`DoubleExponential` quadrature converges as $\exp(-cN/\log N)$, where N is the number of function evaluations. This means that for large N, the accuracy varies nearly linearly with N. The only trouble is that if for a certain value of N the accuracy is too low, you will have to approximately double N to get more accuracy. Of course in doing so you will approximately double the accuracy as well, but doubling the accuracy may be much more than what you want. Since accuracy is nearly linear in N, very high-precision quadrature of analytic integrands should usually be done using `DoubleExponential` quadrature; other methods converge much more slowly. This is especially true when the integrand itself takes a long time to evaluate. The only exception to this is when the integrand is so smooth and well behaved that the numerical simplicity of `GaussKronrod` can give it the advantage.

Another case where `DoubleExponential` quadrature has a distinct advantage is when the integrand is highly oscillatory. In such cases `GaussKronrod` must resolve each of the wiggles; this requires many function evaluations. On the other hand `DoubleExponential` quadrature can depend on the fact that the integrand is analytic and get a good estimate of the integral with many fewer function evaluations. In fact, in evaluating `NIntegrate[Sin[x]/x,{x,300,700}, Method ->DoubleExponential, WorkingPrecision -> 120]`, only 87 of the function evaluations occur between 400 and 600, yet the error in the final result is less than 10^{-107}.

`Trapezoidal` quadrature can be used to examine the trapezoidal method where the stepsize is indirectly controlled by the options `MinRecursion` and `MaxRecursion`. However it is especially useful for problems where the integrand is analytic and periodic and the interval of quadrature is exactly one period. In such cases the error decreases in much the same way as with `DoubleExponential` quadrature except that evaluation of the integrand is much faster since there is essentially no change of variable. If the integrand is not analytic, is not periodic, or the interval of quadrature is not an integral number of periods `Trapezoidal` quadrature will perform rather poorly.

---

[7]Of course, if you are using `DoubleExponential` quadrature and a singularity is known to occur at a particular point in the interior of the interval of integration, the form `NIntegrate[ f[x], {x, x0, xsingularity, x1}]` should be used. See Section 3.3.

[8]Actually we don't need the integrand to satisfy a condition quite so strong; it is acceptable for integrable singularities to occur at the endpoints of the interval of integration.

GaussPoints

The option `GaussPoints` is only used with `GaussKronrod`. To understand what `GaussPoints` means, it is necessary to understand the idea behind Gaussian quadrature and Gauss-Kronrod quadrature. Recall that what we will ultimately do is evaluate the integrand at several points and take a weighted average of those values as an approximation to the integral. With Gaussian quadrature the idea is to choose the points and the weights in a certain "optimal" way, and `GaussPoints` refers to the number of points used in the basic Gauss rule. Gauss-Kronrod quadrature uses the same points that were used in the Gauss rule (so the integrand values there don't have to be re-evaluated), but then it goes on to evaluate the integrand at `GaussPoints` + 1 additional points and uses an entirely different set of weights to approximate the integral. As we would expect the Gauss-Kronrod result usually has less error than the Gauss result, but they are both evaluated and the difference is used as an estimate of the error. If the error is too large to achieve either `AccuracyGoal` or `PrecisionGoal`, the interval is bisected and the process repeated on each half with an entirely different set of points; the old points are discarded. `GaussPoints` refers to the number of points in the fundamental Gauss rule on each subinterval.

If `GaussKronrod` is being used for high-precision evaluation of an integral it is probably a good idea to set `GaussPoints` fairly high, but just how high can only be found by experimentation. However, care should be used with experimentation. Calculating high-precision weights and abscissas for the Gauss rule is fairly time consuming for a large number of `GaussPoints`. For this reason they are saved, but not forever. Calculating an integral exactly the same way a second time can be significantly faster if the weights and abscissas do not have to be recalculated.

We time the quadrature as a function of `GaussPoints`.

```
In[1]:= f[n_] := Timing[NIntegrate[1/x, {x,1,2},
                                   WorkingPrecision->50,
                                   PrecisionGoal->45,
                                   GaussPoints->n]]
```

We might guess that 20 (or fewer) `GaussPoints` is optimal.

```
In[2]:= Table[First[f[n]]/Second, {n,20,50,10}]

Out[2]= {7.61667, 15.1333, 28.8333, 48.1833}
```

But a lot of the time was just spent finding the weights and abscissas.

```
In[3]:= Table[First[f[n]]/Second, {n,20,50,10}]

Out[3]= {1.66667, 0.483333, 0.65, 0.8}
```

MinRecursion and MaxRecursion

`MinRecursion` and `MaxRecursion` have slightly different meanings for the each of the methods. With `MultiDimensional` quadrature, if the estimated error is too large, the region of integration is bisected *in the dimension that is estimated to be responsible for most of the error*; with `GaussKronrod` quadrature there is only one dimension from which to choose. With both `GaussKronrod` and `MultiDimensional` quadrature, each recursive bisection counts toward the level of recursion, which is not permitted to exceed `MaxRecursion`. Setting `MinRecursion` to a positive value forces recursive bisection *before* the integrand is ever evaluated. This can be done to ensure that a narrow spike in the integrand is not missed. It can also speed things up a little since if bisection were found necessary to reduce the error, time would have been spent evaluating the integrand at points that would ultimately be discarded anyway. Of course it also can make things slower by making unnecessary evaluations. In `MultiDimensional` quadrature there is no way of telling *a priori* which dimension should be bisected, so an effort is made to bisect in each dimension for each level of recursion in `MinRecursion`. However, this quickly leads to combinatorial explosion in higher dimensions, so a limit is effectively placed on the depth of recursion that actually occurs.

With `DoubleExponential` and `Trapezoidal` quadrature, no recursive bisection of the interval of integration occurs, so `MinRecursion` and `MaxRecursion` have a different meaning. Recall that with these methods the trapezoid rule is used and the stepsize is halved. The recursion level refers to the number of times the stepsize has been halved and `MinRecursion` and `MaxRecursion` bound the level of recursion.

SingularityDepth

Both `GaussKronrod` and `MultiDimensional` quadrature are adaptive, that is bisection is only done on those subregions where the error is estimated to be large. If there is an integrable singularity on the boundary of the given region of integration, bisection could easily recurse to depth `MaxRecursion` before convergence occurred. To deal with these situations there is the option `SingularityDepth`. If the level of recursion reaches `SingularityDepth` on a subregion containing a boundary point of the original interval, a change of variable will be performed to make the assumed singularity easier to integrate. Note that the singularity is only assumed to exist. If `SingularityDepth` is set too small the reparameterization will often occur when there is no singularity. Since the change of variable is expensive we want to avoid unnecessary reparameterizations. If `SingularityDepth` is set too large, too much time will be spent trying to achieve convergence via recursive bisection while a change of variable would allow convergence much more quickly.

Define a function depending on `SingularityDepth`. Note the integrable singularities in the integrand at the endpoints of the interval of integration.

```
In[1]:= f[n_] := NIntegrate[1/Sqrt[x(1-x)], {x,0,1},
                          MaxRecursion -> 50,
                          SingularityDepth->n]
```

If there are singularities at the endpoints, it is generally faster to deal with them sooner rather than later.

```
In[2]:= Table[First[Timing[f[n]]]/Second, {n,6}]

Out[2]= {1., 1., 1.08333, 1.18333, 1.26667, 1.36667}
```

Define a different function depending on `SingularityDepth`.

```
In[3]:= f[n_] := NIntegrate[Sin[x], {x,0,11Pi},
                          MaxRecursion -> 50,
                          SingularityDepth->n]
```

If there are no singularities at the endpoints, it can be very time consuming to do the change of variable.

```
In[4]:= Table[First[Timing[f[n]]]/Second, {n,6}]

Out[4]= {1.16667, 1.13333, 1.11667, 0.533333, 0.566667,
  0.566667}
```

Singularities internal to the region of integration cannot be dealt with in any special way because it must be known exactly where each singularity occurs. Of course, if you know that a singularity occurs at some internal point, you can specify it in the range of integration. Then the region of integration will just be treated as two (or more) regions of integration and the results will be added together. (See Section 3.3 below.)

`SingularityDepth` is not used with either `DoubleExponential` or `Trapezoidal` quadrature. This is because the reparameterization inherent in `DoubleExponential` quadrature is already able to deal with integrable singularities at endpoints of the interval of integration and `Trapezoidal` quadrature is designed for periodic integrands.

■ 3.3 Discontinuous Integrands

Often one wants to integrate a function that has some sort of singularity in the region of integration. A convenient way to do this is with a `Which` statement. However, quadrature and cubature necessarily assume a certain amount of continuity in the integrand and its derivatives. If the integrand or it's derivatives are not continuous, convergence can be very slow.

Define a function to be integrated.

```
In[1]:= f[x_] := Which[ x < 0,    0,
                        x < 1,    x^2,
                        x < 2,    2x,
                        x < 3,    4,
                        x >= 3,   0]
```

Discontinuities in the integrand and it's derivatives cause the integration to fail.

```
In[2]:= NIntegrate[f[x], {x,-3,6}]

NIntegrate::slwcon:
   Numerical integration converging too slowly; suspect one of
     the following: singularity, oscillatory integrand, or
     insufficient WorkingPrecision.

NIntegrate::ncvb:
   NIntegrate failed to converge to prescribed accuracy after 7
     recursive bisections in x near x = 3.01172.

Out[2]= 7.3
```

Evaluating the integral in individual pieces solves the convergence problem.

```
In[3]:= NIntegrate[f[x], {x,-3,0,1,2,3,6}]

Out[3]= 7.33333
```

3.4 Integration of Vector Fields

Since the first argument to `NIntegrate[ ]` must be a function rather than a list `NIntegrate[ ]` will not integrate vector fields without some modification. The most obvious change would be to simply make `NIntegrate[ ]` be `Listable`, but that won't work because the ranges of integration are also lists. The required change is still quite simple however.

Unprotect it so we can change it.

```
In[1]:= Unprotect[NIntegrate]

Out[1]= {NIntegrate}
```

Make `NIntegrate[ ]` map over the elements of its first argument if it is a list.

```
In[2]:= NIntegrate[f_List, r__] := NIntegrate[#, r]& /@ f
```

`NIntegrate[ ]` still does what it is supposed to for functions.

```
In[3]:= NIntegrate[Sin[x], {x, 0, 3}]

Out[3]= 1.98999
```

For lists it integrates each element separately.

```
In[4]:= NIntegrate[{Sin[x], Cos[x]}, {x, 0, 2}]

Out[4]= {1.41615, 0.909297}
```

It also works for multiple integrals.

```
In[5]:= NIntegrate[{Sin[x+y], Cos[x-y]},
                         {x, 0, 1}, {y, 0, 1}]

Out[5]= {0.773645, 0.919395}
```

3.5 Oscillatory Integrands

Occasionally we want to integrate an integrand that is oscillatory and the integral is not absolutely convergent. To deal effectively with these integrals we must know the zeros of the integrand, but often we only need the first few to get a quite reasonable answer. The idea is to look at the sequence of partial integrals from some fixed point to successive zeros of the integrand. This can be accomplished by simply using `NIntegrate[ ]` over the various ranges of integration, but a faster way is to integrate only between successive zeros and then accumulate these alternating pieces. Consider the problem of trying to evaluate

$$\int_0^\infty \frac{x \sin x}{x^2+4}\,dx = \frac{\pi}{2e^2}$$

Define a function that gives the integral between successive zeros.

```
In[1]:= f[n_] := NIntegrate[x Sin[x]/(x^2 + 4),
                            {x, n Pi, (n+1) Pi}]
```

Form a list of the integrals over the first 10 intervals.

```
In[2]:= a = Table[f[n], {n, 0, 10}]

Out[2]= {0.434001, -0.362003, 0.240436, -0.176644, 0.138989,
  -0.114386, 0.0971148, -0.084344, 0.0745262, -0.0667473,
  0.0604339}
```

Form the sequence of partial sums.

```
In[3]:= b = FoldList[Plus, 0, a]

Out[3]= {0, 0.434001, 0.0719988, 0.312435, 0.135791, 0.274781,
  0.160394, 0.257509, 0.173165, 0.247691, 0.180944, 0.241378}
```

Find the numerical limit of the sequence.

```
In[4]:= SequenceLimit[b]

Out[4]= 0.212584
```

The error is reasonably small and including more terms in the sequence would make it even smaller.

```
In[5]:= % - N[Pi/(2 E^2)]

                     -8
Out[5]= 4.79579 10
```

An alternative is just to use `NSum[ ]` with `Method -> SequenceLimit`.

```
In[6]:= NSum[f[n], {n,0,Infinity},
                Method -> SequenceLimit,
                VerifyConvergence -> False]

Out[6]= 0.212584
```

The error is quite small.

```
In[7]:= % - N[Pi/(2 E^2)]

                     -15
Out[7]= 6.80012 10
```

It is not uncommon to need to integrate expressions involving Bessel functions. Because of the oscillatory nature of Bessel functions the above technique is often needed. To find the required zeros you can use the package `NumericalMath`BesselZeros'` described briefly below in subsection 7.5 and in the *Guide to Standard Mathematica Packages*.

■ 3.6 Singularities along a Curve

When a singularity occurs at an isolated point *Mathematica* can usually deal appropriately with it if it is at an endpoint of an interval of integration or a vertex of a region of integration. If singularities occur all along a curve (or along a higher dimensional manifold) in the region of integration *Mathematica* will probably have difficulties with it. The first thing that you need to do is arrange to get the manifold of singularity to the boundary. This can usually be done by simply splitting the region in some variable.

Mathematica does do special things at the boundaries to try to evaluate integrals numerically, but in some cases numerical roundoff is a severe problem. For example, consider trying to numerically evaluate the following integral

$$f(a,h) = \int_{x=a}^{x=a+h} \int_{y=-\sqrt{1-x^2}}^{y=\sqrt{1-x^2}} \frac{dy}{\sqrt{1-x^2-y^2}} dx$$

The singularities all occur on the boundary $x^2+y^2=1$, but because of a combination of roundoff error and subtractive cancelation of digits, convergence near the singularities requires extremely high precision arithmetic.

By reparameterizing and breaking up the region of integration we can rewrite the integral as

$$f(a,h) = 2(f_1(a,h) + f_2(a,h) - f_3(a,h))$$

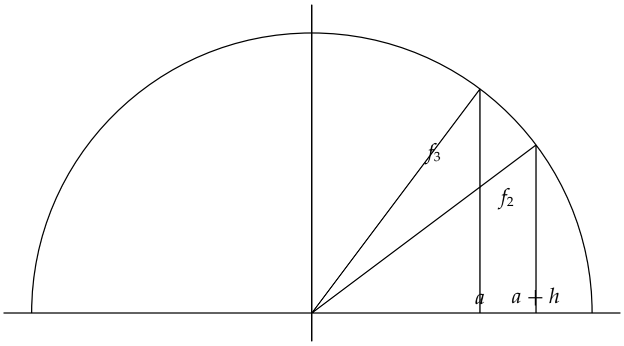
where symmetry about the x axis has been exploited, f_1 deals with the circular sector, f_2 deals with the region $a \le x \le a+h$ not included in the sector, and f_3 deals with the part of the sector with $x < a$. In this particular problem the integral over the sector can be found symbolically, but here we wish to demonstrate that the reparameterization is useful even if we still need to evaluate it numerically.

Define the integral over the sector. Note the reparameterization. The singularity occurs along the boundary $r = 1$.

```
In[1]:= f1[a_,h_] := NIntegrate[r/Sqrt[1-r^2],
          {r, 0, 1}, {t, ArcCos[a+h], ArcCos[a]}]
```

Define the integral over the region the sector missed.

```
In[2]:= f2[a_,h_] := Module[{c = Sqrt[1-(a+h)^2]/(a+h)},
          NIntegrate[1/Sqrt[1-x^2-y^2],
              {x,a,a+h}, {y,0,c x}]]
```



Define the integral over the region in the sector that shouldn't have been included.

```
In[3]:= f3[a_,h_] := Module[{c1 = Sqrt[1-(a+h)^2]/(a+h),
        c2 = Sqrt[1-a^2]/a},
        NIntegrate[1/Sqrt[1-x^2-y^2],
            {x,0,a}, {y,c1 x,c2 x}]]
```

Define the desired function f.

```
In[4]:= f[a_,h_] := 2 (f1[a,h]+f2[a,h]-f3[a,h])
```

Evaluate the function f.

```
In[5]:= f[.3, .2]

Out[5]= 0.628319
```

The essential part of the reparameterization is to get the singularity to involve as few of the variables as possible. In this way roundoff errors in those variables that are unrelated to the singularity have no effect.

■ 3.7 Cauchy Principal Values

Cauchy principal values of divergent integrals can be evaluated using the package `NumericalMath`CauchyPrincipalValue``. The idea here is that the integral is evaluated normally except near the singularity `s` where points to the left of the singularity are paired with points to the right of it. Then `NIntegrate[f[x], {x, s-eps, s+eps}]` is evaluated as `NIntegrate[f[s-t] + f[s+t], {t, 0, eps}]`. Since the two parts `f[s-t]` and `f[s+t]` tend to cancel each other it is sometimes the case that `WorkingPrecision` needs to be set unexpectedly high to achieve the requested `PrecisionGoal` on some subinterval. Note that `CauchyPrincipalValue[ ]` uses separate calls to `NIntegrate[ ]` so `PrecisionGoal` refers to the precision of each separate part, not the precision of the entire integral. The default `AccuracyGoal` is `Infinity`, but it may be necessary to set it to some other value if the `PrecisionGoal` cannot be achieved.

Load the package.

```
In[1]:= << NumericalMath`CauchyPrincipalValue`
```

Define our own version of `LogIntegral`. The `{1, 1/4}` indicates that a singularity occurs at 1 and the interval $1 - \frac{1}{4}$ to $1 + \frac{1}{4}$ is to be treated specially.

```
In[2]:= li[x_] := CauchyPrincipalValue[1/Log[t],
                        {t, 0, {1, 1/4}, x}]
```

```
In[3]:= li[3.] - LogIntegral[3.]

                   -10
Out[3]= 1.50152 10
```

`CauchyPrincipalValue` is more general than the above example exhibits. First, the size of the small interval that is to be treated specially need not be specified; a default will be chosen. For example, we could have defined the above function with the range of integration given only as `{t, 0, {1}, x}`. Second, the behavior with respect to absolutely integrable singularities is unaltered.

Integrate `1/((x-2)(x-4)(x-7))` from 0 to 8, breaking the interval of integration at the points 1, 5, and 6 and taking the Cauchy principal value at the points 2, 4, and 7.

```
In[4]:= CauchyPrincipalValue[1/((x-2)(x-4)(x-7)),
                    {x,0,1,{2},{4,1/2},5,6,{7},8}]

Out[4]= -0.0198661
```

■ 3.8 ListIntegrate

Occasionally one wants to approximate the integral of a function about which only a few values are known. If these values are known to contain errors and `Fit[ ]` can be applied to the data to give an expression for the function, it is probably better to do so and then use either `Integrate[ ]` or `NIntegrate[ ]` to evaluate the integral. If you want, however, you can use `ListIntegrate[ ]`. What `ListIntegrate[ ]` does is find piecewise interpolating polynomials and integrate the resulting function over the desired interval.

Load the package.

```
In[1]:= << NumericalMath`ListIntegrate`
```

Define some equally spaced data. The implied abscissas are 1, 2, 3, . . ., and 8.

```
In[2]:= data = Table[3x^2 - 4x + 2, {x, 0, 7}]

Out[2]= {2, 1, 6, 17, 34, 57, 86, 121}
```

The second argument specifies the spacing of the equally spaced data. The last argument specifies how many points to include in the interpolation for each piece.

```
In[3]:= ListIntegrate[data, 1, 4]

Out[3]= 259

In[4]:= Integrate[3x^2 - 4x + 2, {x, 0, 7}]

Out[4]= 259
```

To explain more clearly what the last argument means, let's look at the above example. On the interval $2 \le x \le 3$ an interpolating polynomial is found using the specified number of points. Since 4 was specified, two points to the left of the interval (*i.e.*, the points at 1 and 2) and two points to the right of the interval (*i.e.*, the points at 3 and 4) are used to find a (cubic) interpolating polynomial. This polynomial is integrated over the interval $2 \le x \le 3$. Likewise for all of the other intervals except the end intervals where it is not possible to find two points on either side of the interval. For the interval $0 \le x \le 1$ the four points used are 0, 1, 2, and 3. A similar non-centered interpolating polynomial is used at the upper endpoint. If the third argument to `ListIntegrate[ ]` is an odd integer it is incremented by one for purposes of symmetry. If no third argument is given a default of 4 is used.

If the data are not evenly spaced there is no constant stepsize and the data must be given as a list of ordered pairs.

Start with data that are unevenly spaced.

```
In[5]:= data = Transpose[{x, 3x^2 - 4x + 2} /.
                    x -> {0,1,2,4,5,7}]

Out[5]= {{0, 2}, {1, 1}, {2, 6}, {4, 34}, {5, 57}, {7, 121}}
```

Use the default value for the number of points.

```
In[6]:= ListIntegrate[data]

Out[6]= 259
```

In the above example the function being integrated was only quadratic and the interpolation with four points found that quadratic exactly on each interval. So the integral must be correct. If the interpolating polynomials do not exactly match the function then the value of `ListIntegrate[ ]` may differ from the "true" value of the integral.

Make a table of data.

```
In[7]:= data = Table[Exp[x], {x,0.,7.}]

Out[7]= {1., 2.71828, 7.38906, 20.0855, 54.5982, 148.413,
  403.429, 1096.63}
```

Find the approximate integral using 6-point interpolation.

```
In[8]:= ListIntegrate[data, 1, 6]

Out[8]= 1097.76
```

This is the "correct" value of the integral.

```
In[9]:= N[Integrate[Exp[x],{x,0,7}]]

Out[9]= 1095.63
```

4 SequenceLimit[ ]

The *Mathematica* function `SequenceLimit[ ]` is perhaps the one function in *Mathematica* that is most easily abused. The basic purpose of `SequenceLimit[ ]` is to take a finite sequence of numbers and extrapolate to the limit. For example, given the sequence

$$\{3+\frac{1}{2},\ 3+\frac{1}{4},\ 3+\frac{1}{8},\ 3+\frac{1}{16},\ 3+\frac{1}{32},\ 3+\frac{1}{64},\ 3+\frac{1}{128}\}$$

`SequenceLimit[ ]` should return the result 3. It is clear that in general there is no way that a finite segment of an infinite sequence can determine the behavior of the sequence at infinity. It is equally clear, however, that we would often like the ability to do so if the sequence is well behaved in some sense. Because it purports to do the impossible, `SequenceLimit[ ]` can give complete garbage if used on a sequence to which it is not applicable. On the other hand it can give amazingly good results very quickly if used correctly.

`SequenceLimit[ ]` is able to find the limit of any sequence whose n^{th} term is of the form

$$\sum_{i=1}^{k} P_i(n)\lambda_i^n$$

where the P_i are polynomials in n and the λ_i are distinct "eigenvalues". The algorithm used is Wynn's ϵ algorithm and in order to get the correct value for the limit the option `WynnDegree` must be set to $k+\sum_{i=1}^{k}\deg(P_i)$ where $\deg(P_i)$ is the degree of the polynomial P_i. In other words `WynnDegree` is the number of degrees of freedom in the model for the data. If none of the λ_i is equal to 1 the limiting value, as given by the algorithm, will be 0. This is the correct result by standard analysis only if the magnitude of each λ_i is less than 1. If for some i, λ_i is 1, then if P_i is a constant the limit of the sequence will be that constant. If P_i has positive degree the limit of the sequence will be `ComplexInfinity`.

The algorithm for `SequenceLimit[ ]` works by transforming a sequence of length n into a sequence of length $n-2$. This transformation is done `WynnDegree` times although the first iteration is slightly different. If the resulting sequence is still of length 3 or greater the cycle of transformations starts over and this process continues until the sequence is too short (*i.e.*, length 1 or 2) to continue. The basic algorithm requires that the length of the original sequence be at least of length 2 `WynnDegree` + 1 so that at least one cycle may complete. However, in order to estimate the error in the result, the original sequence needs to be at least of length 2 `WynnDegree` + 2. The only exception to this is if `WynnDegree` is set to `Infinity` in which case it just iterates as many times as it can.

Define a sequence with total degree 3 and length 8.

```
In[10]:= seq = Table[a0 + (b0 + b1 n) lambda^n, {n, 8}];
```

The limit of the sequence is `a0` *regardless* of the size of `lambda`. Note that `SequenceLimit` does work on symbolic sequences as well. However, it leads to useless results if the model does not fit exactly.

```
In[11]:= SequenceLimit[seq, WynnDegree -> 3]

Out[11]= a0
```

The amazing thing about `SequenceLimit[ ]` is that it often gets the right answer even when the terms of the sequence are not of the correct form.

Form a sequence of numbers known to converge to $\pi/4$.

```
In[12]:= seq = FoldList[Plus, 0,
                     Table[(-1)^k/(2k+1), {k,0,16}]];
```

Using the default value for the option `WynnDegree` gives a pretty good approximation to the limit.

```
In[13]:= SequenceLimit[N[seq]] - N[Pi/4]

                      -13
Out[13]= 1.83853 10
```

Using a different value for `WynnDegree` gives an approximation not much worse.

```
In[14]:= SequenceLimit[N[seq], WynnDegree -> Infinity] -
                      N[Pi/4]

                       -13
Out[14]= -7.25864 10
```

The accuracy of the result is reduced to what *Mathematica* thinks is justified.

```
In[15]:= SequenceLimit[N[seq, 30]]

Out[15]= 0.785398163397448
```

`SequenceLimit[ ]` underlies one of the methods for `NSum[ ]`, `NProduct[ ]`, and `NLimit[ ]`.

5 Numerical Sums and Products

Numerical summation and multiplication in *Mathematica* are done with the functions `NSum[ ]` and `NProduct[ ]`. These two functions are very similar except that one is a sum and the other is a product. Unless otherwise indicated, everything said below regarding `NSum[ ]` is also true for `NProduct[ ]`.

■ 5.1 Sum[ ] and NSum[ ]

As with integration, there are several ways to evaluate sums in *Mathematica*: `Sum[ ]`, `N[Sum[ ]]`, `NSum[ ]`, and `Sum[N[ ], ]`. Because there are so many possibilities, you may wonder what is the best way for a particular problem and what each of them actually does.

The primary difference between `Sum[ ]` and `NSum[ ]` is that, except for sums with only a few terms, `NSum[ ]` attempts to find the numerical result by approximate means rather than by explicitly adding up the terms as `Sum[ ]` does. Of course this requires a certain amount of good behavior on the part of the summand; certainly `NSum[Prime[n]^(-2), {n, Infinity}]` cannot be expected to give very good results. Likewise, `Sum[Prime[n]^(-2), {n, 10000}]` can be expected to take a very long time since the result will be rational with a very large denominator. The only way that *Mathematica* can be expected to find an approximation to `Sum[Prime[n]^(-2), {n, Infinity}]` that is reasonably correct and does not take a very long time is to use `Sum[N[Prime[n]]^(-2), {n, 1, 10000}]` or `NSum[Prime[n]^(-2), {n, 10000}, NSumTerms -> 10000]`.

The form `N[Sum[ ], ]` is special in that `Sum[ ]` is first evaluated and if `Sum[ ]` returns itself the `N[Sum[ ], ]` is changed to `NSum[ ]`. Since there are many options to `NSum[ ]`, if you want complete control over the behavior of `N[Sum[ ], ]` you should either use `NSum[ ]` directly or set the default values for the options to `NSum[ ]` prior to using `N[Sum[ ], ]`.

■ 5.2 NSum[ ] and NProduct[ ]

Several of the numerical operations have many options. `NSum[ ]` is one such function; it has nine different options. While default values have been given for them that are reasonable for many problems, significant efficiency improvements can often be achieved by changing them for specific problems. This section is presented in an effort to help you better understand how to use `NSum[ ]` and `NProduct[ ]`.

WorkingPrecision

`WorkingPrecision` is simply the number of digits used in the arithmetic as the summand is evaluated. In particular, the summand must be evaluated at different values of the summation variable. The value of the summation variable will have precision `WorkingPrecision` when the summand is evaluated. If bignum arithmetic is being used, *i.e.*, if `WorkingPrecision` is greater than machine precision, the precision of the evaluated summand may be more or less than the precision of the summation variable; it just depends on how ill-conditioned the summand is. If machine arithmetic is being used, you will not get control of the amount of error in evaluating the summand at any particular point (except inasmuch as you can find a well-conditioned algorithm to evaluate it), but you will get a significant improvement in speed. The default for `WorkingPrecision` is to use machine precision.

Get a high-precision approximation to the sum that can be used for comparison.

```
In[1]:= truesum = Sum[Sin[N[i,30]], {i,1,100}];
```

Using machine precision you get a machine-precision answer. Note that the sum here is *not* found by simply summing 100 terms.

```
In[2]:= NSum[Sin[x],{x,1,100}] - truesum

Out[2]= 0.
```

Repeat the sum with a larger value of `WorkingPrecision`.

```
In[3]:= y = NSum[Sin[x],{x,1,100}, WorkingPrecision->30]

Out[3]= -0.127171013660420115436 8
```

With bignum arithmetic the accuracy of the result is reduced to what *Mathematica* thinks is justified by the estimate of the error.

```
In[4]:= Precision[y]

Out[4]= 21
```

In fact the error is smaller than *Mathematica* thought is was.

```
In[5]:= SetPrecision[y,30] - truesum

                   -24
Out[5]= -4.23 10
```

Compiled

As in `NIntegrate[ ]`, `Compiled` in `NSum[ ]` and `NProduct[ ]` can take only two values: `True`, which is the default, and `False`. `Compiled` specifies whether the summand is to be compiled. However if the `WorkingPrecision` is more than machine precision no compilation will be performed even if `Compiled` is set to `True`.

VerifyConvergence

In the case of infinite sums it is sometimes the case that the series diverges but the algorithm used will return a finite value anyway, *e.g.*, if `SequenceLimit` is the method used. There may be times when you want this sort of behavior (the value will be the same as that given by many of the definitions for divergent series, *e.g.*, Abel's sum or Euler's sum) or you may already know that the series is convergent. The default value of `VerifyConvergence` is `True` which means that *Mathematica* will attempt to verify that the terms tend to zero and that the series converges by the ratio test. If it is clear that the series diverges a result of `ComplexInfinity` will be returned. If the results of these tests indicate convergence or it is not clear whether the series diverges the evaluation will proceed. If `VerifyConvergence` is `False` these tests will not be performed. This can result in a substantial savings in time. In the case of `NProduct` the factors are tested for convergence to unity, and a test analogous to the ratio test is performed.

Mathematica realizes that the series diverges.

```
In[1]:= NSum[2^i, {i,1,Infinity}]

Out[1]= ComplexInfinity
```

This is the correct result by many of the definitions of summation for divergent series.

```
In[2]:= NSum[2^i, {i,1,Infinity},
                               VerifyConvergence->False]

SequenceLimit::seqlim:
   Warning: The general form of the sequence could not be
     determined, and the result may be incorrect.

Out[2]= -2.
```

NSumTerms and NProductFactors

The basic idea with NSum[ ] is to explicitly add up the first several terms and to estimate accurately the sum of the terms that have been neglected. The option NSumTerms is used to specify how many terms are to be explicitly evaluated and summed. For NProduct[ ] the analogous option is NProductFactors. If NSumTerms is greater than the total number of terms in a finite sum only the terms that actually appear in the sum will be evaluated and the sum of the "neglected" terms will be zero. By increasing the value of NSumTerms you can often get a more accurate result.

Method

Currently there are three different methods that can be used for numerical summation and multiplication. These methods are Integrate, NIntegrate, and SequenceLimit[9]. The default value for Method is Automatic, which means that *Mathematica* will use some heuristics to try to decide which of the two methods Integrate and SequenceLimit would be more appropriate. There is no really good way to choose which algorithm to use and it can happen that the other method would have been more appropriate. You can choose the method yourself by setting the Method option. This can save time even if *Mathematica* would have chosen the same method.

The method Integrate is just Euler-Maclaurin summation. The value of the sum of the neglected terms is estimated by integration and corrections are made that involve derivatives of the summand at the endpoints. Clearly, if the summand cannot be differentiated, or the summand cannot be evaluated at arbitrary points in the interval covered by the neglected terms, this method will not work. Integrate[ ] is used to do the integration and if it fails NIntegrate[ ] is used instead. (The only way to control the options to NIntegrate[ ] that are not also options to NSum[ ] is to change their default values as options to NIntegrate[ ]. It was thought to be cleaner to do this than to encumber NSum[ ] with many more options than it already has.) If differentiation fails we do not use numerical differentiation, since that would nearly be equivalent to just summing the series in the first place. The method Integrate has the advantage that divergent series will be found to diverge. For NProduct[ ] the method Integrate means that the product is evaluated as Exp[NSum[Log[...], ...]] where the method for the NSum[ ] is Integrate.

The exact result would be 0.

```
In[1]:= NSum[1/(n(n+1)(n+2)), {n, Infinity},
                    Method -> Integrate] - 1/4

Out[1]= 0.
```

The exact result would be 0.

```
In[2]:= NProduct[1+1/n^2, {n, Infinity},
                    Method -> Integrate] - N[Sinh[Pi]/Pi]

                       -10
Out[2]= -3.18519 10
```

The method NIntegrate is very similar to the method Integrate except that symbolic integration is never even attempted. This can save time if it is known that symbolic integration will fail anyway. NIntegrate will not be chosen as the method when Method is Automatic.

The method SequenceLimit applied to infinite numerical sums forms a sequence of partial sums of the NSumExtraTerms terms and passes this sequence to SequenceLimit[ ]. The option NSumExtraTerms refers to the number of terms that will be used in this sequence and the option WynnDegree is the value that will be passed as the option to SequenceLimit[ ]. The method SequenceLimit applied to the finite sum NSum[f[x], {x,a,b}] means to evaluate NSum[f[x], {x,a,Infinity}] and NSum[f[x], {x,b+1,Infinity}] using SequenceLimit and then find the difference between the two sums. It turns out that it is a good thing that SequenceLimit[ ] returns finite values for some sequences that diverge because in such cases taking the difference of the two divergent series results in the correct answer. In the case of NProduct[ ] a sequence of partial products of length NProductExtraFactors is passed to SequenceLimit[ ].

[9] The method Fit is really SequenceLimit, but it is misleading: it has nothing to do with the function Fit[ ]. Fit is provided as a valid method for reasons of compatibility with older versions of *Mathematica*.

The exact result would be 0.

```
In[1]:= NSum[(n^2+n+1)/2^n, {n, Infinity},
                  Method -> SequenceLimit] - 9

                  -15
Out[1]= 7.10543 10
```

Here we use the correct value for WynnDegree and only 8 terms.

```
In[2]:= s = NSum[(n^2+n+1)/2^n, {n, Infinity},
                  NSumTerms -> 0,
                  NSumExtraTerms -> 8,
                  WynnDegree -> 3,
                  Method -> SequenceLimit,
                  WorkingPrecision -> 60]

Out[2]= 9.
```

The only error is arithmetic error.

```
In[3]:= Precision[s]

Out[3]= 56
```

The exact result would be nearly 0.

```
In[4]:= NProduct[1+2^-n, {n, Infinity},
                  Method -> SequenceLimit] -
                  2.384231029031371724149899288678397238 77

                  -15
Out[4]= 1.77636 10
```

■ 5.3 Error Control

Controlling the amount of error in NSum and NProduct is very indirect. The way to control roundoff error is by varying the option WorkingPrecision. The control of truncation error (*i.e.*, the error resulting from the algorithm itself) is different for the two methods.

If the method used is Integrate, the truncation error is controlled by the options NSumTerms, AccuracyGoal, and PrecisionGoal. Here again it is generally true that larger values for NSumTerms will result in smaller error. The options AccuracyGoal and PrecisionGoal are used to control the termination of NIntegrate[ ] (if it is used) and to control how many derivative terms of the Euler-Maclaurin summation will be used. It should be noted, however, that for many problems Euler-Maclaurin summation results in an asymptotic series, *i.e.*, the terms decrease for a while and then start increasing again; the series is divergent. (*Mathematica* will notice when the terms start increasing in size, will stop, and will reduce the accuracy of the result to what it thinks is justified.) The series is asymptotic, however, and as NSumTerms is increased the size of the smallest term decreases. Of course in the case of finite series it may turn out that NSumTerms has to be set to include all of the terms in the series before the error in the sum of the neglected terms can be made sufficiently small.

The default value for PrecisionGoal is Automatic, which means 10 fewer digits than WorkingPrecision[10], and the default value for AccuracyGoal is Infinity. The options NSumExtraTerms and WynnDegree are not used with the method Integrate.

With too few NSumTerms we don't get convergence.

```
In[1]:= NSum[1/k^2, {k, Infinity},
                    Method -> Integrate,
                    NSumTerms -> 0] - N[Pi^2/6]

NSum::emcon:
   Euler-Maclaurin sum failed to converge to requested error
     tolerance.

Out[1]= 0.0122088
```

[10]There is an exception to this. Different machines have different precisions for their machine arithmetic. To have some consistency between machines, when machine arithmetic is being used, the value of Automatic for PrecisionGoal means that the precision goal is to be six digits.

With the default value for `PrecisionGoal` we cannot expect a very precise result.

```
In[2]:= NSum[1/k^2, {k, Infinity},
              Method -> Integrate,
              NSumTerms -> 20] - N[Pi^2/6]

                    -11
Out[2]= -1.31775 10
```

Here is the same computation with a larger value for `PrecisionGoal`. This goal can be achieved only if the other options have values that make it possible.

```
In[3]:= NSum[1/k^2, {k, Infinity},
              Method -> Integrate,
              PrecisionGoal -> 14,
              NSumTerms -> 20] - N[Pi^2/6]

                   -16
Out[3]= 2.22045 10
```

If the method used is `SequenceLimit` the only knobs you have to twiddle that affect the truncation error are `NSumTerms`, `NSumExtraTerms`, and `WynnDegree`. Assuming that the method `SequenceLimit` is appropriate for the problem it is generally true that larger values for `NSumTerms` and `NSumExtraTerms` will result in smaller error. The best value for `WynnDegree` is difficult to decide upon except in the case where the terms in the series are of the form

$$\sum_{i=1}^{k} P_i(n)\lambda_i^n$$

where the P_i are polynomials and the λ_i are distinct "eigenvalues". In this case the value of `WynnDegree` should be $k + \sum_{i=1}^{k} \deg(P_i)$ where $\deg(P_i)$ is the degree of the polynomial P_i. The options `AccuracyGoal` and `PrecisionGoal` are not used with the method `SequenceLimit`.

Find the exact value for comparison.

```
In[1]:= truesum = Sum[(1+k)/2^k - k/3^k, {k, 1, 50}];
```

The numerical sum is not very accurate.

```
In[2]:= NSum[(1+k)/2^k - k/3^k, {k, 1, 50},
             Method -> SequenceLimit,
             NSumTerms -> 2,
             NSumExtraTerms -> 4] - truesum

Out[2]= 0.0530365
```

Increasing `NSumTerms` improves the result.

```
In[3]:= NSum[(1+k)/2^k - k/3^k, {k, 1, 50},
             Method -> SequenceLimit,
             NSumTerms -> 20,
             NSumExtraTerms -> 4] - truesum

                   -8
Out[3]= 2.18151 10
```

Increasing `NSumExtraTerms` also improves the result.

```
In[4]:= NSum[(1+k)/2^k - k/3^k, {k, 1, 50},
             Method -> SequenceLimit,
             NSumTerms -> 2,
             NSumExtraTerms -> 9] - truesum

Out[4]= 0.0000121496
```

With enough `NSumExtraTerms` and the correct value for `WynnDegree` the only error is due to the arithmetic.

```
In[5]:= NSum[(1+k)/2^k - k/3^k, {k, 1, 50},
             Method -> SequenceLimit,
             WorkingPrecision -> 30,
             NSumTerms -> 2, NSumExtraTerms -> 10,
             WynnDegree -> 4] - truesum

Out[5]= 0.

In[6]:= Accuracy[%]

Out[6]= 32
```

■ 5.4 An Example

Consider the problem of numerically evaluating

$$\sum_{i=0}^{\infty} f_i e^{-i}$$

where f_i is the i^{th} Fibonacci number. Some points that one might not think of are:

- We must define the starting values for the recurrence using inexact values. If we define `f[0] = f[1] = 1` we will never get a match and we will hit `$RecursionLimit`; the summation index is inexact when it gets used. More precisely, it uses values of the summation index with precision given by `WorkingPrecision`. Since `N[0]` is exact, we also need to give a starting value for `f[0]`.
- We must specify `Method -> SequenceLimit`. If we try to use the default value of `Automatic` *Mathematica* will try to use the ratio test to decide which method to use. Since there is no obvious way to find the limit of the ratio of the terms, *Mathematica* will attempt to do it numerically. But then *Mathematica* will need to find f_i for very large i and `$RecursionLimit` will be a problem.
- We must also specify `VerifyConvergence -> False` to avoid the ratio test.

Furthermore we should specify a value for `WynnDegree`. Since the Fibonacci numbers satisfy a three-term recurrence relation the characteristic polynomial for the relation is quadratic. Thus there are two "eigenvalues". (We don't need to worry about whether they are distinct, since if they are not their multiplicity needs to be counted anyway.)

Define the initial conditions.

```
In[1]:= f[0] = f[0.] = f[1.] = 1;
```

Define the recurrence relation.

```
In[2]:= f[n_] := f[n] = f[n-1] + f[n-2]
```

Find the numerical value of the series. Notice that a total of only seven terms was used. A skeptic might want to try using a few hundred terms with `Sum[ ]` to verify this result.

```
In[3]:= NSum[f[i] Exp[-i], {i, 0, Infinity},
                WorkingPrecision -> 60,
                Method -> SequenceLimit,
                VerifyConvergence -> False,
                NSumTerms -> 1, NSumExtraTerms -> 6,
                WynnDegree -> 2]

Out[3]= 2.01294210825481695463443794395887350933148375409636663\
  26161
```

■ 5.5 Alternating Series

Alternating series (and other quasi-periodic series) are not particularly problematic for the method `SequenceLimit`, but they wreak havoc with Euler-Maclaurin summation. This, however, is not sufficient reason to discard Euler-Maclaurin summation without further investigation. `NSum[ ]` can often use the method `Integrate` quite well if the summand is expressed appropriately. Consider the alternating series

$$1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \cdots$$

which we saw above with the method `SequenceLimit`. By pairing the terms together we get the sum

$$\frac{2}{1 \cdot 3} + \frac{2}{5 \cdot 7} + \frac{2}{9 \cdot 11} + \cdots.$$

Attempt to sum the alternating series. Simply replacing $(-1)^n$ by $\cos(n\pi)$ gives an error that is unacceptable. (Note that $\cos(n\pi)$ is preferable to $(-1)^n$ because it avoids complex numbers for nonintegral n.)

```
In[1]:= NSum[Cos[n Pi]/(2n+1), {n, 0, Infinity},
                    Method -> Integrate,
                    VerifyConvergence -> False] - N[Pi/4]

NIntegrate::ncvb:
   NIntegrate failed to converge to prescribed accuracy after 7
                                                         56
     recursive bisections in n near n = 2.28833 10   .

Out[1]= 0.000210461
```

Pairing the terms makes the error quite small.

```
In[2]:= NSum[2/(n^2 - 1), {n, 2, Infinity, 4},
                    Method -> Integrate,
                    VerifyConvergence -> False] - N[Pi/4]

                     -11
Out[2]= -1.37999 10
```

Another example of the advantage that can be gained with a little thought is

$$\sum_{n=1}^{\infty} \frac{\cos(\sqrt{n(n-1)}\pi)(-1)^n}{n^2}$$

Note that evaluating the terms at nonintegral values of n would involve complex numbers. But by rewriting it as

$$\sum_{n=1}^{\infty} \frac{\cos((n-\sqrt{n(n-1)})\pi)}{n^2}$$

we can get the sum quite nicely. The general rule for choosing a particular form is to choose the form that is as smooth as possible.

■ 5.6 EulerSum

There are many specialized techniques for accelerating the convergence of a sequence or series. Wynn's ϵ-algorithm is fairly general and is incorporated into `NSum[ ]` as the method `SequenceLimit`. Another technique is Euler's transformation. This method is different enough that it is not used in `NSum[ ]`, but rather is incorporated in a separate function called `EulerSum[ ]` in the package `NumericalMath`NLimit``.

Options

There are four options to `EulerSum[ ]`: `WorkingPrecision`, `Terms`, `ExtraTerms`, and `EulerRatio`. The first three of these options correspond to similar options in `NSum[ ]`. The option `EulerRatio` needs some explanation, however. The generalized Euler transformation used in `EulerSum[ ]` has a parameter that ideally should be the ratio limit in the ratio test for convergence of the series. The option that specifies this parameter is `EulerRatio`, and if it is set to `Automatic`, *Mathematica* will use the ratio of the last two terms that are included. You should realize that even if the value you choose is in error the Euler transformation often will still accelerate the convergence of the series.

Euler's transformation is used to transform a series into a separate term plus a different series with the same total sum. This new series can then likewise be transformed by Euler's transformation. Mathematically, Euler's transformations can be used to sum *exactly* series whose terms are of the form

$$\sum_{i=1}^{k} P_i(n)\lambda_i^n$$

where the P_i are polynomials and the λ_i are distinct "eigenvalues". This works by successively transforming the series to eliminate the various λ_i until the series part of the total sum is the series with all terms equal to zero. However, unlike `NSum[ ]` with `Method -> SequenceLimit`, Euler transformations require the specification of the distinct λ_i and the number of degrees of freedom in each of the polynomials P_i (*i.e.*, the degree plus 1 of each of the polynomials.) The way to specify all of this in the function `EulerSum[ ]` is to set the option `EulerRatio` to a list of pairs of the form `{`λ_i`,` $deg_i + 1$`}`. If any of the deg_i is 0, the corresponding pair can be replaced by simply λ_i. Another difference between `NSum[ ]` with `Method -> SequenceLimit` and

EulerSum[ ] is that the number of extra terms in EulerSum[ ] can be as few as the total number of degrees of freedom rather than two more than twice that number. You should note however that EulerSum[ ] does not attempt to estimate the error in the result and simply returns whatever it gets rather than reducing accuracy of its answer.

EulerSum[ ] can be used for symbolic sums of the form for which it has no truncation error (WorkingPrecision should be set to Infinity). Since it requires fewer terms and the limiting ratios of the terms is obvious from the form of the summand, it will tend to be faster than using SequenceLimit. EulerSum[ ] is not designed for such use, however, and an implementation intended for symbolic sums would be faster.

An Example

Load the package containing EulerSum[ ].

```
In[1]:= << NumericalMath`NLimit`
```

The error is rather small considering that we used only the first 12 terms of the series and the wrong EulerRatio.

```
In[2]:= EulerSum[(-1)^k/(2k+1),{k, 0, Infinity}] - N[Pi/4]

                    -7
Out[2]= 1.93662 10
```

EulerSum[ ] can be used to give exact results to certain problems.

```
In[3]:= EulerSum[(n^2-4n+1)/2^n-(4n^2-n+2)/3^n,
                 {n, Infinity},
                 EulerRatio -> {{1/3, 3}, {1/2, 3}},
                 ExtraTerms -> 6, Terms -> 0,
                 WorkingPrecision -> Infinity]

            29
Out[3]= -(--)
            4
```

6 NLimit[ ] and ND[ ]

The package NumericalMath`NLimit` does more than find numerical sums. It also includes the functions NLimit[ ] and ND[ ]. NLimit[ ] is a numerical analogue of the function Limit[ ]. It works by forming a sequence of function values at various values of the limit variable and then using either SequenceLimit[ ] or EulerSum[ ] to extrapolate to the limit. If EulerSum[ ] is used, the sequence is first differenced so that when EulerSum[ ] transforms its argument and sums the result the sum will be the required limit. ND[ ] also forms a sequence, but instead of containing function values, the sequence contains divided differences with successively smaller stepsizes.

■ 6.1 Options

NLimit[ ] has five options. WorkingPrecision is used in the same way as it is in many numerical functions. Terms is the number of function values that are included in the sequence that is formed prior to extrapolation. Method can be either SequenceLimit or EulerSum. WynnDegree is the value that is to be passed to SequenceLimit[ ] if that method is used, otherwise it is ignored. Finally, Scale sets the scale for the points that are to be examined. If the limit variable is to approach Infinity, the sequence includes the function values at Scale, 10 Scale, 100 Scale, 1000 Scale, ... If the limit variable is to approach some finite value x0, the sequence includes the function values at x0+Scale, x0+Scale/10, x0+Scale/100, x0+Scale/1000, ... Note that directional limits can be achieved by giving Scale the appropriate value.

ND[ ] has the three options WorkingPrecision, Scale, and Terms. Their meaning is similar to their meaning in NLimit[ ] except that Scale refers to the size of the largest step in the divided differences and the steps are successively halved rather than divided by 10. It should be noted that although one might be tempted to choose an extremely small `Scale`, doing so will result in massive loss of precision when the function values are differenced.

■ 6.2 Examples

Load the package.

```
In[1]:= << NumericalMath`NLimit`
```

The error is very small.

```
In[2]:= NLimit[Sin[x]/x, x->0] - 1

                     -15
Out[2]= 8.21565 10
```

Using only the values x = 1.0, 0.1, and 0.01 still gives reasonably small error.

```
In[3]:= NLimit[Sin[x]/x, x->0, Terms -> 3] - 1

                     -7
Out[3]= 8.56007 10
```

This is the derivative of $e^{\sin x}$ evaluated numerically at $x = 2$.

```
In[4]:= ND[Exp[Sin[x]], x, 2]

Out[4]= -1.03312
```

This is the error in the third derivative of $e^{\sin x}$ evaluated numerically at $x = 2$.

```
In[5]:= ND[Exp[Sin[x]], {x, 3}, 2, Terms -> 10] -
                    N[D[Exp[Sin[x]], {x, 3}] /. x -> 2]

                     -7
Out[5]= 2.23989 10
```

7 Root Finding

■ 7.1 FindRoot[ ]

`FindRoot[ ]` is used to numerically solve equations or systems of equations for individual solutions. These equations must not be differential equations or integral equations; for such problems `DSolve[ ]`, `NDSolve[ ]`, `Integrate[ ]`, or `NIntegrate[ ]` may be most appropriate. Of course if the equation or system of equations is linear or even polynomial you are probably better off using `Solve[ ]`, `NSolve[ ]`, `Roots[ ]`, `NRoots[ ]`, or `LinearSolve[ ]`. As a general rule `FindRoot[ ]` should be used when the other more specialized functions will not work. There are several reasons for this. First of all `FindRoot[ ]` will only solve for a single solution at a time. Second, unless you know how many solutions there are you will never know whether you have found them all. Third, each solution requires that you give a starting point that is sufficiently close to the solution that the algorithm will converge to that solution. On the other hand, if you only want a particular solution and you have a reasonable idea of where it might be (such as from a graph), `FindRoot[ ]` might be just what you want to use.

Mathematically it makes sense to have the number of equations different from the number of unknowns. If there are more equations than unknowns and if the equations are consistent, there will be a solution; if they are not consistent, there will not be a solution. If the number of equations is less than the number of unknowns (and the equations are consistent) the solution will be not a single point, but rather a higher dimensional manifold such as a curve or a surface. However, `FindRoot[ ]` requires that the number of equations be the same as the number of unknowns and that the equations be consistent. Furthermore, `FindRoot[ ]` requires that the functions be continuous and assumes that the first derivatives also be continuous, although convergence sometimes will occur without continuity of the first derivatives. If at the solution point the Jacobian of the system is singular (*i.e.* if the linearized equations at the solution form a singular system of linear equations) trouble can arise that usually manifests itself in the form of slow convergence. However, as we shall see below, there is a way to deal with this problem to a certain extent. For purposes of this discussion let us denote the number of equations and unknowns by n. This will simplify the discussion somewhat.

7.2 Syntax

`FindRoot[ ]` uses several different algorithms and the choice of algorithm is determined by the syntax; there is no option called `Method` for `FindRoot[ ]`. In all cases the first argument to `FindRoot[ ]` can be a single function or equation (if $n = 1$) or a list of length n of functions or equations ($n \geq 1$). Actually the equations in the first argument are first converted to functions by subtracting the right-hand side from the left-hand side; what we really solve for is a zero of the resulting list of functions or vector-valued function. Arguments 2 through $n + 1$ refer to the individual unknowns and their form (which must be the same) determines the algorithm that will be used.

Newton's Method

If each of the arguments 2 through $n + 1$ is of the form `{`*var*`,` *start*`}` where *var* is a variable and *start* is a numerical value that is to be used as a starting point for that variable, the method that will be used is Newton's method. Now Newton's method requires the ability to evaluate the derivative of the function whose zero is being sought. In the case of $n > 1$ this is the Jacobian of the function with respect to the variables. Note that the order of the variables is important; the order in which they appear as arguments 2 through $n + 1$ is the order that is used. If the Jacobian can be found without any trouble and the option `Jacobian` is set to `Automatic` then *Mathematica* will find the Jacobian and use it without any concern on your part. However, if the function involves nondifferentiable functions such as `Abs[ ]` there will be a problem. If you have special knowledge about the function that will enable you to define the Jacobian where *Mathematica* could not find it you can set the option `Jacobian` to an expression that will evaluate to the Jacobian of the function. Of course, if you give it an incorrect expression for the Jacobian it will still attempt to use it and it will not work.

Another reason you might want to specify the Jacobian is if you are repeatedly solving the same equations with slightly different parameters. Then, rather than allowing *Mathematica* to repeat itself by finding the Jacobian each time, you can save time by finding it once and specifying it with the option `Jacobian`. If a common subexpression appears many times in the expression for the Jacobian, you might want to write a function that efficiently evaluates the Jacobian at any point and specify the Jacobian using that function. Finally, if the function really needs to be held (`FindRoot[ ]` has the attribute `HoldAll`, but the first argument *does* get evaluated in the process of finding the Jacobian) then there is no way to evaluate the Jacobian since evaluation of the Jacobian requires that the function be evaluated first; in such cases you must provide the Jacobian.

As an example, let us consider the problem of finding the base b and the point at which the function $y = b^x$ is tangent to the line $y = x$. We solve the problem in several different ways.

Since `FindRoot[ ]` is `HoldAll`, we need to evaluate the derivative explicitly in the function before `x` gets assigned a numerical value.

```
In[1]:= FindRoot[Evaluate[{b^x == x, D[b^x, x] == 1}],
                                      {x, 1}, {b, 2}]

Out[1]= {x -> 2.71828, b -> 1.44467}
```

Define the function whose root will be sought.

```
In[2]:= func = {b^x - x, D[b^x, x] - 1}

           x             x
Out[2]= {b  - x, -1 + b  Log[b]}
```

In this case the `func` does not need to be evaluated. It does, however, get evaluated in the process of finding the Jacobian.

```
In[3]:= FindRoot[func, {x, 1}, {b, 2}]

Out[3]= {x -> 2.71828, b -> 1.44467}
```

Find the Jacobian.

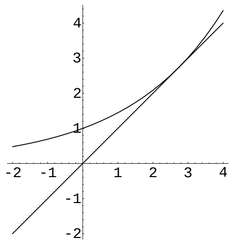
```
In[4]:= jac = Outer[D, func, {x, b}]

                  x             -1 + x
Out[4]= {{-1 + b  Log[b], b       x},

    x       2   -1 + x    -1 + x
  {b  Log[b] , b       + b       x Log[b]}}
```

Find the solution using the specified Jacobian.

```
In[5]:= FindRoot[func, {x, 1}, {b, 2}, Jacobian -> jac]

Out[5]= {x -> 2.71828, b -> 1.44467}
```

Demonstrate that the solution is correct.

```
In[6]:= Plot[Evaluate[{t, (b^t /. %)}], {t, -2, 4},
               AspectRatio -> Automatic]
```

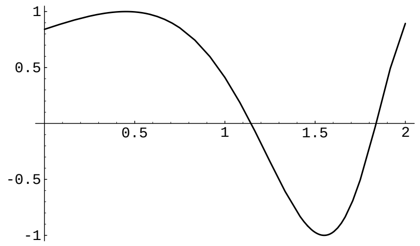


Brent's Method

Brent's method is a fairly standard and robust method for root finding in one unknown when the range of the function is a subset of the reals and a sign change can be found, *i.e.*, if $f(x_1)$ and $f(x_2)$ are real and of opposite sign and $f(x)$ is real for all $x_1 < x < x_2$. Since f is assumed to be continuous, by the intermediate value theorem there must be a zero of f between x_1 and x_2. In fact the algorithm is just a rather sophisticated proof of the intermediate value theorem. To use Brent's method, you must specify the two starting values x_1 and x_2. If the function values at these two points are not real or are not of opposite sign the secant method will be used instead of Brent's method.

Examine the function to choose two starting points such that the function values at those points are of opposite sign.

```
In[1]:= Plot[Sin[Exp[x]], {x, 0, 2}]
```



Find the unique root between 1 and 1.5.

```
In[2]:= FindRoot[Sin[Exp[x]], {x, {1, 1.5}}]

Out[2]= {x -> 1.14473}
```

Find the unique root between 1.5 and 2.0. The two starting values do not need to be given as a pair.

```
In[3]:= FindRoot[Sin[Exp[x]], {x, 1.5, 2.0}]

Out[3]= {x -> 1.83788}
```

Secant Method

Newton's method requires the Jacobian, but if the Jacobian cannot be found symbolically or if it is too expensive to evaluate you can use the secant method. This method can be thought of as being similar to Newton's method except that the Jacobian is approximately evaluated numerically from previous values of the function. An effort is made to keep the different sample points distributed in such a way that these function values contain the required information for numerical approximation to the Jacobian. For example if we allowed the sample points to become collinear then we could only extract information regarding partial derivatives in the direction along the line of collinearity.

Use the secant method to solve the problem discussed above.

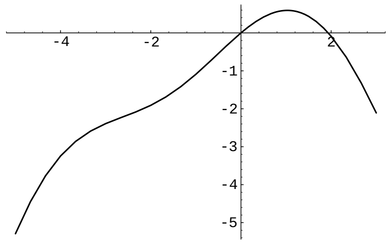
```
In[1]:= FindRoot[Evaluate[{b^x == x,
                    D[b^x, x] == 1}],
                    {x, 1, 2}, {b, 1, 2}]

Out[1]= {x -> 2.71828, b -> 1.44467}
```

Restricting the Search Region

One of the problems with root finding is that it is possible to wander around without ever converging to a solution. In fact it can wander quite far from the starting point. It is sometimes desirable to abort the search if the solution wanders too far. To do this, just specify bounds for each restricted variable. It should be emphasized that this does not alter the search process in any way; the only thing that happens is that if the search ever gets outside of search region it will be ended.

Plot the function whose zeros we want.

```
In[1]:= Plot[Sin[x] - x^2/4, {x, -5, 3}]
```



Starting at $x = 2$, find a root, stopping the search if it ever gets outside of the region $1 \le x \le 3$.

```
In[2]:= FindRoot[Sin[x] - x^2/4, {x, 2, 1, 3}]

Out[2]= {x -> 1.93375}
```

Use the starting values $x = -2$ and $x = -1$ to find a root, stopping the search if it gets outside of the region $-4 \le x \le 1$.

```
In[3]:= FindRoot[Sin[x] - x^2/4, {x, -2, -1, -4, 1}]

                       -8
Out[3]= {x -> 9.17319 10  }
```

A warning message is generated when the search gets outside of the region $-4 \le x \le 1$.

```
In[4]:= FindRoot[Sin[x] - x^2/4, {x, {-3, -2}, -4, 1}]

FindRoot::regex:
   Reached the point {1.96266} which is outside the region
    {{-4., 1.}}.

Out[4]= {x -> 1.96266}
```

When using `FindRoot[ ]` to solve several equations in several unknowns, any combination of restricted and unrestricted variables is allowed.

■ 7.3 Options

`FindRoot[ ]` has six options. `WorkingPrecision` and `Compiled` are used in the same way as they are used in other numerical functions. `Jacobian` has been explained above under Newton's method (see Section 7.2). If Newton's method is not used the option `Jacobian` is ignored. `MaxIterations` is related to a bound on the number of points at which the function is to be evaluated. It is not the actual bound itself because in the case of several unknowns with the secant method the function must be evaluated at several points before the iteration count begins. `AccuracyGoal` is the accuracy goal for the value of the function at the solution, or more precisely the 1-norm of that value; the accuracy of the coordinates of the solution is not considered. There is no `PrecisionGoal` for `FindRoot[ ]` since the value of the function is supposed to be 0 at the solution.

There is one more option, `DampingFactor`, that requires some explanation. It is not used in Brent's method, but it is used in the other two methods. Newton's method for a single function in a single unknown is

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

and it converges quadratically if f has a continuous first derivative that does not vanish at the solution. However, if f has a multiple root at the solution, the derivative vanishes and the convergence is not quadratic. The secant method likewise has problems. An example of a different problem occurs with $f(x) = x^{1/3}$ where things get even worse: the derivative is infinite at the solution and Newton's method diverges.

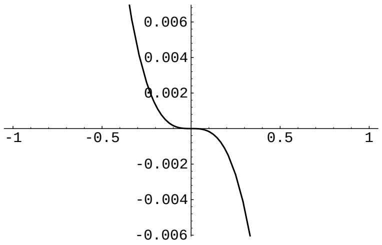
The form of Newton's method that `FindRoot[ ]` uses is

$$x_{n+1} = x_n - d\frac{f(x_n)}{f'(x_n)}$$

where d is the `DampingFactor`. (The default value of `DampingFactor` is 1.) This allows some control over the convergence of the algorithm. In particular, if the function has a zero of multiplicity m then setting `DampingFactor` to m will restore quadratic behavior to the algorithm. Likewise, if the root is a fractional zero, setting `DampingFactor` to its multiplicity will give quadratic convergence. Appropriate choices for `DampingFactor` with the secant method will likewise give good convergence rates.

Here is a function with a triple zero at $x = 0$.

```
In[1]:= Plot[Sin[x] - x, {x, -1, 1}]
```



Using Newton's method we count the number of function evaluations used. (The first function evaluation is to get the Jacobian.)

```
In[2]:= n = 0; {FindRoot[n++; Sin[x] - x, {x, 2},
                         AccuracyGoal -> 15], n}

FindRoot::cvnwt:
   Newton's method failed to converge to the prescribed accuracy
     after 15 iterations.

Out[2]= {{x -> 0.00402227}, 16}
```

With `DampingFactor` chosen correctly the convergence is much faster.

```
In[3]:= n = 0; {FindRoot[n++; Sin[x] - x, {x, 2},
                              AccuracyGoal -> 15,
                              DampingFactor -> 3], n}

                             -11
Out[3]= {{x -> -3.37854 10      }, 7}
```

With the default `DampingFactor` it doesn't converge. Note the first evaluation of the function before `x` has been set to a value. This occurs in the evaluation of the `Jacobian`.

```
In[4]:= FindRoot[Print[x]; x^(1/3), {x, 2}]

x
2.
-4.
-4.
-1.
-1.
0.5
-1.
-0.25
0.125
-0.25
-0.0625
0.03125
-0.0625
-0.015625
0.0078125
-0.015625
-0.00390625
0.00195312

FindRoot::cvnwt:
   Newton's method failed to converge to the prescribed accuracy
     after 15 iterations.

Out[4]= {x -> -0.00390625}
```

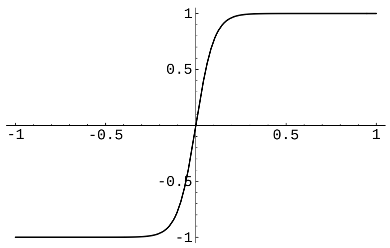
It converges quite nicely with the proper `DampingFactor`.

```
In[5]:= FindRoot[Print[x]; x^(1/3), {x, 2},
                              DampingFactor -> 1/3]

x
2.
           -16
-4.44089 10
           -16
-4.44089 10
          -31                -31
8.87469 10    - 1.64346 10    I
          -31                -31
8.87469 10    - 1.64346 10    I
           -45                -46
-2.97776 10    + 5.25487 10    I

                          -45                -46
Out[5]= {x -> -2.97776 10    + 5.25487 10    I}
```

Another use of `DampingFactor` is to control the size of the steps taken until quadratic convergence sets in. Newton's method or the secant method simply will not converge to a root if the starting value is too far away.

Unless the starting value is between approximately $-.1$ and $.1$ Newton's method will not converge.

```
In[6]:= Plot[Tanh[10x], {x, -1, 1}]
```



We could specify a small value for the `DampingFactor` but that would destroy the quadratic convergence. The alternative is to specify a sequence of `DampingFactor` values. Each iteration uses a successive value in the sequence of values and the last value is used repeatedly until the algorithm stops.

Here is a sequence of damping factors to use. The 1 at the end will be used repeatedly.

```
In[7]:= 2^Range[-6,0]

          1   1   1   1  1  1
Out[7]= {--, --, --, -, -, -, 1}
         64  32  16  8  4  2
```

With this sequence of damping factors the algorithm converges nicely.

```
In[8]:= FindRoot[Print[x]; Tanh[10x], {x, 1},
                              DampingFactor -> %]

x
1.
0.710297
0.214818
0.100102
0.0773861
0.049336
0.0204657
-0.00057627
          -8
1.27582 10
          -8
1.27582 10
           -18
-3.96948 10

                          -18
Out[8]= {x -> -3.96948 10    }
```

A particularly important use of a sequence of damping factors occurs when you want to do an exhaustive search of a region for all of the roots. If you simply loop through a lattice of mesh points as starting values you are likely to find each of the roots many times (assuming the mesh is fine enough). One way to limit the number of times each root is found is to restrict the search region with each call to `FindRoot[ ]`. But if you do that, you could inadvertently end searches that would eventually find a root within the specified search region but which step outside that region just once or twice at the beginning of the search. By using a sequence of damping factors to reduce such wanderings you can get restricted searches while reducing the chances that good searches terminate.

7.4 Choosing Starting Values

You are pretty much on your own when it comes to choosing starting values for `FindRoot[ ]`, but there are a few tricks that sometimes can be helpful. For a single function in a single real unknown, you can plot the function and usually get a pretty good idea of where to start. If the domain of the unknown is complex you can plot the absolute value of the function over the complex region in question; a zero will usually show up as a sharp dip in the function values if the `PlotPoints` option to `Plot3D` is set sufficiently high. Likewise, if you are working with two real unknowns you can plot the sum of the absolute values of the two functions over the region in $\mathbf{R}^2$. An alternative to `Plot3D` is `ContourPlot`. In higher dimensions things get more difficult. You cannot plot some related function, but sometimes you can use `FindMinimum[ ]` on the sum of squares of the functions of which you are trying to find zeros, *i.e.*, the 2-norm of the vector-valued function. If complex numbers are involved you need a complex conjugate to make the norm real. This can be slow and it can result in local minima that are not zero, so it is far from foolproof.

7.5 Zeros of Bessel Functions

For certain applications the zeros of Bessel functions are needed. While it is possible to find them individually using `FindRoot[ ]`, to do so requires good starting values. The package ``NumericalMath`BesselZeros` `` can be used to find lists of zeros of Bessel functions in sorted order. The package is documented more completely in the *Guide to Standard Mathematica Packages*. Here we just give a few brief examples.

Read in the package.

```
In[1]:= << NumericalMath`BesselZeros`
```

Find the first 3 (nontrivial) zeros of `BesselJ[1, #]&`.

```
In[2]:= BesselJZeros[1, 3]

Out[2]= {3.83171, 7.01559, 10.1735}
```

Find the 6th through the 8th zeros of `BesselY[2, #]&`.

```
In[3]:= BesselYZeros[2, {6, 8}]

Out[3]= {19.539, 22.694, 25.8456}
```

8 Minimization

8.1 FindMinimum[ ]

`FindMinimum[ ]` is used to find numerically the minimum[11] value of a real valued continuous function, called the objective function. To be effective a certain amount of continuity is assumed in the first derivative of the objective function as well, so `FindMinimum[ ]` should not be applied to linear programming problems; you should use `ConstrainedMin[ ]` or `LinearProgramming[]` for such problems. `FindMinimum[ ]` is not very effective for nonlinear programming either, but currently `ConstrainedMin[ ]` does not support nonlinear constraints or nonlinear objective functions. (You will have to use "penalty" functions to achieve the constraints.) If the objective function is a function of several variables the "derivative" is really the gradient of the function.

You might think that numerical minimization could be accomplished by using `FindRoot[ ]` to find where the gradient of the of the objective function became zero. There are several problems with this approach. First of all you might end up at a maximum rather than a minimum or you might get trapped at a saddle point. Another problem is that when you get into the case of a function of several variables, numerical root finding is inherently more difficult than minimization finding. It is in fact sometimes easier to use `FindMinimum[ ]` to find a minimum of some norm rather than finding a zero of a vector-valued function of several variables. Of course there is no guarantee that the minimum will be zero, so you still might not find the zero for which you are looking.

[11] If you want the maximum of a function, just find the minimum of the negative of the function whose maximum is desired.

The basic method for finding the minimum has two parts. One part is a one-dimensional minimization algorithm that is based on one of Brent's two minimization algorithms. Which of these algorithms is used depends upon whether information about the derivative of the objective function is to be used. If the objective function is a function of a single variable, this part is all that is needed. If there are several variables involved, another part of the algorithm is needed. This part of the algorithm attempts to choose the direction to go from the current point to the minimum. If this direction can be chosen correctly, then searching in that direction with the appropriate algorithm of Brent will find the desired minimum. There are also two direction choosing algorithms: one using derivative information and the other not using it.

You should be aware that it is possible for `FindMinimum[ ]` to get stuck at a saddle point. This is unlikely to happen except in cases where the curve of steepest ascent from the saddle point is in fact a straight line and the starting point is on that line. In such cases the algorithm follows the line of steepest descent from the starting point and when it hits the saddle point it cannot decide whether to turn to the left or right to continue its descent. Sometimes you will be warned that a saddle point is possible, but there are cases where it does not recognize this possibility. If you start at a point that is not special in some way it is very unlikely that this problem will arise, but restarting from a perturbation of the "solution" will normally circumvent the problem if it does arise.

■ 8.2 Syntax

The syntax for `FindMinimum[ ]` is similar to that for `FindRoot[ ]`. The first argument specifies the objective function and if the objective function is a function of n variables, arguments 2 through $n+1$ correspond to the n variables. Also, like `FindRoot[ ]`, `FindMinimum[ ]` uses several different algorithms and the choice of algorithm is determined by the form of arguments 2 through $n+1$.

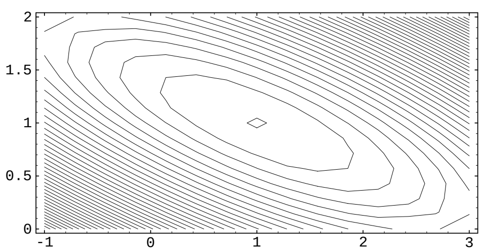
In particular, if you give each variable a single starting value the method used is a conjugate gradient method that uses the gradient of the objective function. When using this method the option `Gradient` can be used to specify the gradient of the objective function. If you leave it set to the default value of `Automatic` *Mathematica* will attempt to find the gradient symbolically from the expression for the objective function. The reasons you might want to specify the gradient yourself are the same reasons you might want to specify the Jacobian yourself when using Newton's method in `FindRoot[ ]` (See Section 7.2 above).

Let `func` be the objective function that is to be minimized.

```
In[1]:= func = (x+2y-3)^2 - (x-1)(y-1)

                                            2
Out[1]= -((-1 + x) (-1 + y)) + (-3 + x + 2 y)
```

Plot the level curves of the objective function.

```
In[2]:= ContourPlot[func, {x, -1, 3}, {y, 0, 2},
                        Contours -> 40,
                        ContourShading -> False,
                        AspectRatio -> Automatic,
                        ContourSmoothing -> Automatic]
```



Find the minimum using the conjugate gradient method and letting *Mathematica* find the gradient. Note that `func` gets evaluated before the gradient is found.

```
In[3]:= FindMinimum[func, {x, 1}, {y, 0}]

                    -29
Out[3]= {1.97215 10    , {x -> 1., y -> 1.}}
```

Find the gradient.

```
In[4]:= grad = Outer[D, {func}, {x, y}][[1]]

Out[4]= {1 - y + 2 (-3 + x + 2 y), 1 - x + 4 (-3 + x + 2 y)}
```

Find the minimum as before, but specify the gradient rather than let *Mathematica* find it.

```
In[5]:= FindMinimum[func, {x, 1}, {y, 0},
                           Gradient -> grad]

                  -29
Out[5]= {1.97215 10    , {x -> 1., y -> 1.}}
```

Giving the wrong gradient breaks the algorithm.

```
In[6]:= FindMinimum[func, {x, 1}, {y, 0},
                        Gradient -> Reverse[grad]]

FindMinimum::fmcv:
   FindMinimum failed to converge to the prescribed accuracy
     within 30 iterations.

Out[6]= {0.299766, {x -> 1.8275, y -> 0.696484}}
```

The other method used by *Mathematica* for finding minima is a direction set algorithm. In this method the gradient is not needed, but it keeps a record of recent function evaluations and tries to choose the direction to the minimum based on this information. This method is chosen by specifying two starting points for each variable.

We want to see the points in the search path. Start with an empty list.

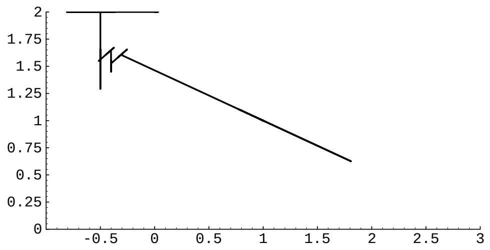
```
In[7]:= searchpath = {}

Out[7]= {}
```

Find the minimum, keeping a record of the points at which the function gets evaluated.

```
In[8]:= FindMinimum[AppendTo[searchpath, {x, y}];
                    func, {x, 0, 1}, {y, 2, 1}]

                  -24
Out[8]= {5.30829 10    , {x -> 1., y -> 1.}}
```

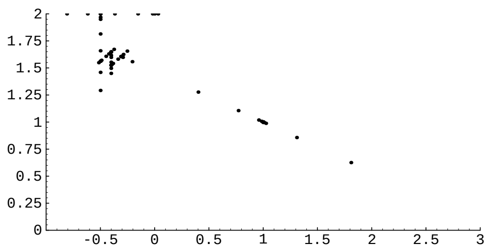
Plot the search path. Note that once it gets the direction chosen correctly it gets to the solution quite quickly. Of course most objective functions are not simple quadratic functions and the correct direction will never be found exactly, only in the limit as the search gets close to the solution.

```
In[9]:= ListPlot[searchpath,
                 AspectRatio -> Automatic,
                 PlotJoined -> True,
                 AxesOrigin -> {-1, 0},
                 PlotRange -> {{-1,3}, {0, 2}}]
```



Plot the individual points where the function was evaluated.

```
In[10]:= ListPlot[searchpath,
                  AspectRatio -> Automatic,
                  AxesOrigin -> {-1, 0},
                  PlotRange -> {{-1,3}, {0, 2}}]
```



As with `FindRoot[ ]`, you can restrict the variables used in `FindMinimum[ ]` to certain regions. If the search ever gets outside of the specified region the search will be terminated.

Here the search is restricted to $-2 \leq x \leq 1.5$, with y unrestricted.

```
In[11]:= FindMinimum[func, {x, 0, 1, -2, 1.5}, {y, 2, 1}]

                     -28
Out[11]= {8.81158 10    , {x -> 1., y -> 1.}}
```

As with `FindRoot[ ]`, you can have any combination of restricted and unrestricted variables with either one or two starting values. There is one difference between restricting variables in the two algorithms. In `FindRoot[ ]` the only effect that restricting the variables has on the algorithm is that it terminates if the search ever strays from the specified region. In `FindMinimum[ ]` the restrictions also slightly affect the points that get sampled in that the size of the region (if it is restricted) is used to set the scale for the initial step sizes. This does not amount to a practical control mechanism on the algorithm, however.

■ 8.3 Options

`FindMinimum[ ]` has six options. `WorkingPrecision` and `Compiled` are used in the same way as they are used in other numerical functions. `Gradient` is similar to `Jacobian` of `FindRoot[ ]` and has been explained above. It is only used if the method is the conjugate gradient method; otherwise it is ignored. `MaxIterations` refers to the number of times the basic one-dimensional search is allowed to be used rather than the total number of function evaluations and also to the number of function evaluations allowed in *each* one-dimensional search. Unlike `FindRoot[ ]`, `FindMinimum[ ]` has both `AccuracyGoal` and `PrecisionGoal` as options. As with `AccuracyGoal` in `FindRoot[ ]`, they do not refer to the accuracy or precision of the coordinates of the solution, but rather to the value of the objective function at the solution, *i.e.*, the minimum value itself.

■ 8.4 Linear Programming

The function `FindMinimum[ ]` is useful for finding local minima of unconstrained, smooth functions of one or several variables. For other types of problems it will not be very effective, although constraints can sometimes be simulated through the use of penalty functions. Linear programming is one way to solve problems in which the function to be minimized (or maximized) is piecewise linear and for which the region to be considered is bounded by linear constraints. *Mathematica* provides three functions for solving linear programming problems: `ConstrainedMin[ ]`, `ConstrainedMax[ ]`, and `LinearProgramming[ ]`.

`ConstrainedMax[`f`, {`*inequalities*`}, {`x`, `y`, ... }]`	find the global maximum of f in the region specified by the inequalities
`ConstrainedMin[`f`, {`*inequalities*`}, {`x`, `y`, ... }]`	find the global minimum of f in the region specified by the inequalities
`LinearProgramming[`c`, `m`, `b`]`	find the vector $\mathbf{x}$ which minimizes $\mathbf{c}.\mathbf{x}$ subject to the constraints $\mathbf{m}.\mathbf{x} \geq \mathbf{b}$ and $\mathbf{x} \geq 0$

Solving linear programming problems.

Consider the problem of finding the minimum value of the function

$$f(x, y) = \max(x, y, 2 - (x + y)).$$

Mathematica cannot solve this problem in its present form. There are two difficulties with it. First of all the "objective function" is not linear, but rather the maximum of three linear functions. Second, there are no constraints to the region over which the problem is to be solved. The solution to these deficiencies is to add a third variable. Let the objective function be

$$g(x, y, z) = z$$

and let the feasible region be the points

$$\{(x, y, z) \mid z \geq x, z \geq y, z \geq 2 - (x + y)\}.$$

The problem[12] is now in exactly the form required by `ConstrainedMin[ ]`.

Solve the constrained minimization problem.

```
In[1]:= ConstrainedMin[z, {z > x, z > y, z > 2-(x+y)},
                {x, y, z}]

         2        2       2       2
Out[1]= {-, {x -> -, y -> -, z -> -}}
         3        3       3       3
```

9 Solving Differential Equations

9.1 Introduction

Despite the capability of `DSolve[ ]`, only a few classes of ordinary differential equations have analytic solutions. When `DSolve[ ]` cannot find the answer, numerical techniques are available to obtain approximate solutions. `NDSolve[ ]` is the *Mathematica* function that handles ordinary differential equations numerically.

`NDSolve[{`eqn_1, eqn_2, … `}, `y`, {`x`, `$xmin$`, `$xmax$`}]`	find a numerical solution for the function y with x in the range *xmin* to *xmax*
`NDSolve[{`eqn_1, eqn_2, … `}, {`y_1, y_2, … `}, {`x`, `$xmin$`,`$xmax$`}]`	find numerical solutions for several functions y_i

Finding numerical solutions to differential equations.

When a differential equation is solved symbolically, the answer is often given in the form of a general solution containing arbitrary constants, without considering boundary conditions. The boundary conditions are substituted into the general solution to arrive at a particular solution by eliminating the arbitrary constants. As long as the number of boundary conditions is equal to the number of arbitrary constants, the particular solution can be represented numerically. The nature of the boundary conditions does not play a significant role in evaluating the particular solution.

In numerical solutions of ordinary differential equations, boundary conditions dictate the solution methods employed. If the boundary conditions involve only one point in the independent variable `x`, the conditions are called initial conditions and the problem becomes an initial value problem. If more than one point is involved, the problem is termed a boundary value problem. For example, a differential equation with boundary conditions `y[0] == 1`, `y'[0] == 2` is an initial value problem, while one with `y[0] == 1`, `y[1] == 2` is a boundary value problem. Both classes of problems are important. The current version of `NDSolve[ ]` supports only initial value problems, but we give an example below of how to work around this restriction.

Input to `NDSolve[ ]` must include not only the differential equations to be solved, but also a sufficient number of initial conditions to completely specify the solution. In general, a single differential equation with derivatives up to order n requires initial conditions for up to $(n-1)$ derivatives. For a system of differential equations, the number of initial conditions needed is equal to the sum of the orders of the individual equations.

[12] We have actually altered the problem slightly. In the original statement of the problem the point $(1, 1, 17)$ is above the set of points $\{(x, y, f(x, y))\}$, but is included as a feasible point in the new problem. This sort of altering of problems is not uncommon in linear programming.

The third argument to `DSolve[ ]` gives the independent variable and the range of the independent variable over which the equation is to be solved.

A finite range is needed for `x`.

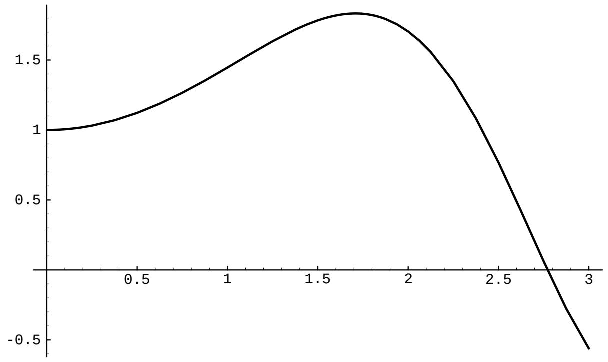
```
In[2]:= NDSolve[{y'[x] == -y[x], y[0] == 1}, y[x], {x, 0, 5}]

Out[2]= {{y[x] -> InterpolatingFunction[{0., 5.}, <>][x]}}
```

Three initial conditions are needed for a third order differential equation.

```
In[3]:= NDSolve[{y'''[x] + x y''[x] + x^2 y'[x] + x^3 y[x] == 0,
                     y''[0] == 1, y'[0] == 0, y[0] == 1},
                 y[x], {x, 0, 3}]

Out[3]= {{y[x] -> InterpolatingFunction[{0., 3.}, <>][x]}}
```

Here is a plot of the result.

```
In[4]:= Plot[ y[x] /. %, {x, 0, 3}]
```



Because `NDSolve[ ]` does not hold its arguments unevaluated (it can't since it needs to send the list of equations to `Solve[ ]`) you can sometimes get into trouble. The fix is usually quite easy, but the problem can be quite frustrating if you don't know what to do about it.

Define a function that will be used as the right-hand side of the ODE.

```
In[5]:= f[x_] := If[x < 0, x, x^2, foo]
```

Attempt to solve the ODE.

```
In[6]:= NDSolve[{y'[x] == f[x], y[-1] == 2},
            y[x], {x, -1, 1}]

NDSolve::ndnum:
   Differential equation does not evaluate to a number at x =
    -1..

Out[6]= {{y[x] -> InterpolatingFunction[{-1., -1.}, <>][x]}}
```

Define a different function that will be used as the right-hand side of the ODE. But this time insist that it only evaluate if the argument is a real number.

```
In[7]:= f1[x_Real] := If[x < 0, x, x^2, foo]
```

Solve the ODE.

```
In[8]:= NDSolve[{y'[x] == f1[x], y[-1] == 2},
            y[x], {x, -1, 1}]

Out[8]= {{y[x] -> InterpolatingFunction[{-1., 1.}, <>][x]}}
```

■ 9.2 InterpolatingFunction

`NDSolve[ ]` returns solutions as `InterpolatingFunction` objects, which are stored internally as piecewise polynomials. (`InterpolatingFunctions` can also be generated by the *Mathematica* function `Interpolation[ ]`.) An `InterpolatingFunction` object consists of two elements. The first element specifies the range of values of the independent variable within which the approximate solution can be found. The second element is a table of coefficients storing the necessary interpolation information. In standard output (`OutputForm`), this table is printed as `<>`, hiding the lengthy lists of numbers. `InputForm` or `FullForm` can be used to display this table if you want to see it.

Note that the range specification in an `InterpolatingFunction` returned by `NDSolve[ ]` will not always match the range requested in the input, since `NDSolve[ ]` may not have been able to find the solution for the entire range.

`NDSolve[ ]` cannot find the solution across the singularity[13] in the middle of the range. The range that was found is indicated in the first argument of the `InterpolatingFunction` object.

```
In[9]:= NDSolve[{y'[x] == 1/(2-x)^2, y[0] == 1/2}, y[x],{x,0,4}]

NDSolve::mxst: Maximum number of steps reached at the point 2..

Out[9]= {{y[x] -> InterpolatingFunction[{0., 2.}, <>][x]}}
```

If the desired solution is specified as, say, `y`, rather than `y[x]`, `NDSolve[ ]` will return the result as a pure `InterpolatingFunction` object, rather than as an expression in `x`.

The solution is given as a pair of `InterpolatingFunction` objects.

```
In[10]:= NDSolve[{y'[x] == z[x], z'[x] == -y[x],
                 y[0] == 0, z[0] == 1}, {y,z},
                 {x, 0, Pi}]

Out[10]= {{y -> InterpolatingFunction[{0., 3.14159}, <>],
   z -> InterpolatingFunction[{0., 3.14159}, <>]}}
```

This selects the solution for `y`.

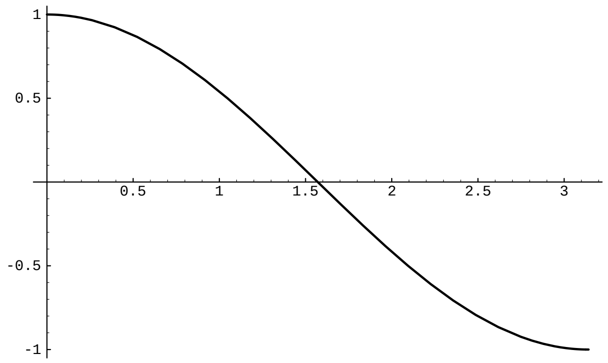
```
In[11]:= sol = y /. %[[1]]

Out[11]= InterpolatingFunction[{0., 3.14159}, <>]
```

The solution can be evaluated at any numerical point in the range of the `InterpolatingFunction` object.

```
In[12]:= sol[2.4]

Out[12]= 0.675457
```

Here is a plot of the derivative of the result. The derivative of the exact solution is $\cos(x)$.

```
In[13]:= Plot[Evaluate[sol'[x]],{x,0,3.1415}]
```



9.3 Other Features in NDSolve

`NDSolve[ ]` uses "multistep" methods to compute solutions. Multistep methods are generalizations of the well-known Euler method from elementary calculus. The basic idea of the Euler method is to start from the given solution value y_0 at the initial point x_0, and proceed a small step to the next value, y_1, using the derivative y'_0 computed from the differential equation. The computation is done using the equation:

$$y_1 = y_0 + \Delta x \cdot y'_0$$

The step size Δx is constantly revised to achieve maximum efficiency. This process is repeated until the end of the requested range, or the algorithm is unable to continue.

By specifying the initial conditions for the differential equation, the starting point x_0 for the independent variable is implicitly specified as well. If the third argument to `NDSolve[ ]` is given as {*x*, *xmax*} rather than {*x*, *xmin*, *xmax*}, *xmin* is taken to be equal to x_0. Also, if both *xmin* and *xmax* are given, neither of them need to be equal to the value x_0 used in the initial conditions. `NDSolve[ ]` will do the numeric integration automatically in both directions from x_0 .

Here *xmin* is taken to be 1.

```
In[14]:= NDSolve[{y'[x] == x y[x], y[1] == 1}, y, {x, 5}]

Out[14]= {{y -> InterpolatingFunction[{1., 5.}, <>]}}
```

Here `NDSolve[ ]` goes in both directions from $x = 1$.

```
In[15]:= NDSolve[{y'[x] == y[x] + Cos[x], y[1] == 1}, y, {x, -5,
         5}]

Out[15]= {{y -> InterpolatingFunction[{-5., 5.}, <>]}}
```

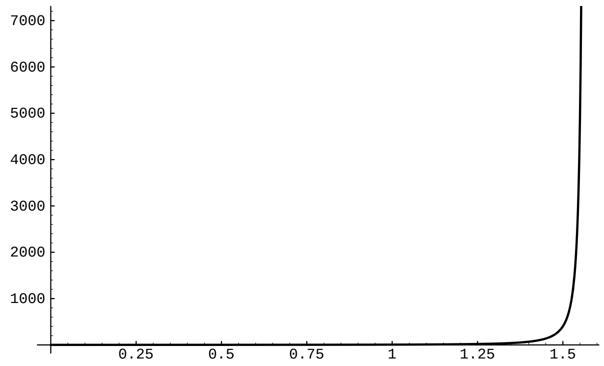
It is not uncommon for differential equations to possess singularities. `NDSolve[ ]` detects singularities by monitoring the size of a step taken during the computation. If the step size gets too small, it is an indication of a singularity and `NDSolve[ ]` stops.

The solution to this equation has a singularity at the first zero of `Cos[x]`.

```
In[16]:= NDSolve[{y'[x] == Cos[x] y[x]^2, y[0] == 1}, y, {x, 5}]

NDSolve::ndsz:
   At x = 1.56645, step size is effectively zero; singularity
     suspected.

Out[16]= {{y -> InterpolatingFunction[{0., 1.56645}, <>]}}
```

A plot of the solution shows the singularity.

```
In[17]:= Plot[Evaluate[y[x] /. %], {x, 0, 1.5664}]
```



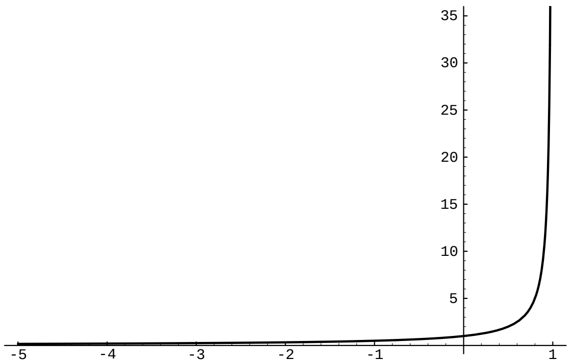
A singularity can consume an infinite number of steps. Sometimes, it takes far too many steps before the singularity can be identified. As second defense against singularities, `NDSolve[ ]` limits the number of steps that can be taken. This feature can save a lot of unnecessary work after the solution profile becomes sufficiently clear. The default limitation of 500 steps can be changed using the `MaxSteps` option. A value of `MaxSteps -> Infinity` effectively removes the limit.

A warning message is generated when the maximum number of steps is exceeded.

```
In[18]:= NDSolve[{y'[x] == y[x]^2, y[0] == 1}, y, {x, -5, 5}]

NDSolve::mxst:
   Maximum number of steps reached at the point 0.999981.

Out[18]= {{y -> InterpolatingFunction[{-5., 0.999981}, <>]}}
```

This shows the discontinuity.

```
In[19]:= Plot[Evaluate[y[x] /. %], {x, -5, 0.9999}]
```



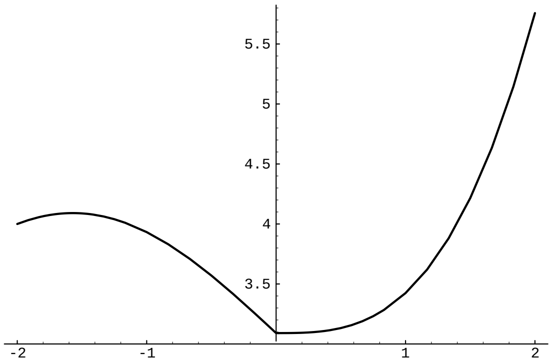
Another type of singularity occurs when there is a discontinuity in the derivatives computed from the differential equation. `NDSolve[ ]` responds to the discontinuity by reducing the step size until the solution becomes reasonably smooth again.

The derivative `y'[x]` is discontinuous at `x = 0`.

```
In[20]:= NDSolve[{y'[x] == If[ x > 0, x^2, -Cos[x]],
                          y[-2] == 4}, y, {x, 2}]

Out[20]= {{y -> InterpolatingFunction[{-2., 2.}, <>]}}
```

The solution has a kink at the discontinuity.

```
In[21]:= Plot[Evaluate[y[x] /. %], {x, -2, 2}]
```



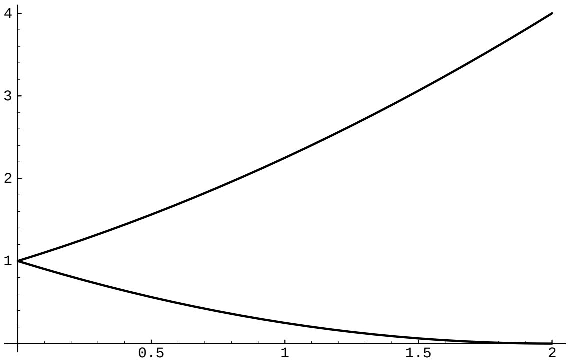
Some differential equations have more than one solution. When this happens, `NDSolve[ ]` will often generate all of them.

This equation has two solutions.

```
In[22]:= NDSolve[{y'[x]^2 == y[x], y[0] == 1}, y, {x, 2}]

Out[22]= {{y -> InterpolatingFunction[{0., 2.}, <>]},
  {y -> InterpolatingFunction[{0., 2.}, <>]}}
```

A plot shows how the solutions diverge from the initial conditions.

```
In[23]:= Plot[Evaluate[y[x] /. %], {x, 0, 2}]
```



In dealing with a large number of equations, it is useful to have a systematic naming convention for functions. One convenient method is to use an index for each function, as in `y[`*i*`]`.

Construct a system of six equations.

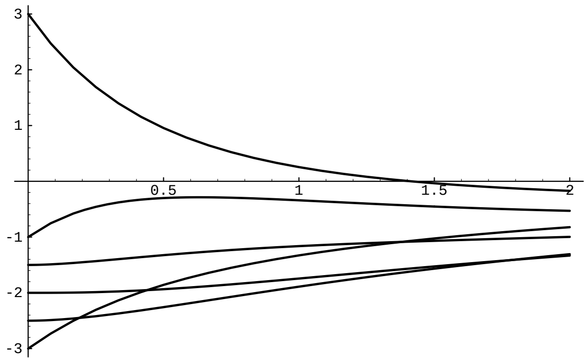
```
In[24]:= n = 6 ; eqns = Join[{y[1]'[x] == -2 y[1][x] + y[2][x]},
            Table[y[i]'[x] == y[i-1][x] - 2 y[i][x] +
                          y[i+1][x], {i, 2, n-1}],
            {y[n]'[x] == y[n-1][x] - 2 y[n][x],
                  y[1][0] == 3, y[n][0] == -3},
            Table[y[i][0] == -i/2., {i, 2, n-1}]]

Out[24]= {(y[1])'[x] == -2 y[1][x] + y[2][x],
  (y[2])'[x] == y[1][x] - 2 y[2][x] + y[3][x],
  (y[3])'[x] == y[2][x] - 2 y[3][x] + y[4][x],
  (y[4])'[x] == y[3][x] - 2 y[4][x] + y[5][x],
  (y[5])'[x] == y[4][x] - 2 y[5][x] + y[6][x],
  (y[6])'[x] == y[5][x] - 2 y[6][x], y[1][0] == 3,
  y[6][0] == -3, y[2][0] == -1., y[3][0] == -1.5,
  y[4][0] == -2., y[5][0] == -2.5}
```

The solution is a list of six `InterpolatingFunction` objects.

```
In[25]:= NDSolve[eqns, Table[y[i], {i,n}], {x, 2}]

Out[25]= {{y[1] -> InterpolatingFunction[{0., 2.}, <>],
   y[2] -> InterpolatingFunction[{0., 2.}, <>],
   y[3] -> InterpolatingFunction[{0., 2.}, <>],
   y[4] -> InterpolatingFunction[{0., 2.}, <>],
   y[5] -> InterpolatingFunction[{0., 2.}, <>],
   y[6] -> InterpolatingFunction[{0., 2.}, <>]}}
```

This shows all six solutions on the same plot.

```
In[26]:= Plot[Evaluate[Table[y[i][x],{i,1,n}] /. %[[1]]],
            {x, 0, 2}, PlotRange->All]
```



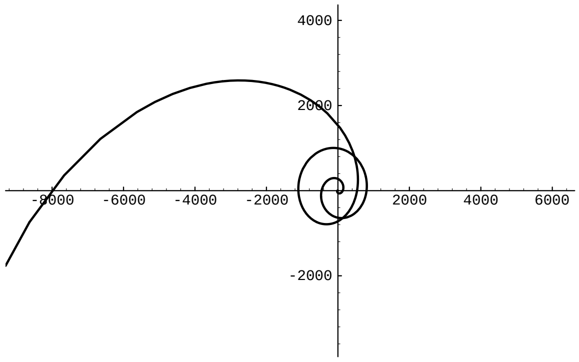
`NDSolve[ ]` can also solve differential equations with complex coefficients. Differential equations with complex coefficients are common in electrical engineering.

Here is a system of differential equations with complex coefficients.

```
In[27]:= NDSolve[{y1'[x] == (2+4I) y1[x] +
                  (-2-2I) y2[x] + (4+2I) y3[x],
            y2'[x] == (3/4+11I/4) y1[x] +
                  (-1-I) y2[x] + (7/2+2I) y3[x],
            y3'[x] == (-1/2+3I/2) y1[x] -
                  2I y2[x] + (2+3I) y3[x],
            y1[0] == 1, y2[0] == 1+I, y3[0] == 1+2I},
            {y1[x], y2[x], y3[x]}, {x, 10}]

Out[27]= {{y1[x] -> InterpolatingFunction[{0., 10.}, <>][x],
   y2[x] -> InterpolatingFunction[{0., 10.}, <>][x],
   y3[x] -> InterpolatingFunction[{0., 10.}, <>][x]}}
```

The real and imaginary parts of the solution can be shown together.

```
In[28]:= ParametricPlot[Evaluate[{Re[y1[x]], Im[y1[x]]}
                   /. %[[1]]], {x,0,10}]
```



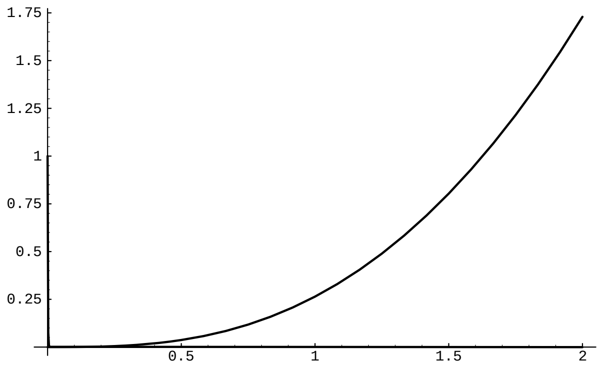
Initial value problems can be classified as either "stiff" or "non-stiff". Stiff problems are distinguished from non-stiff problems in that stiff problems have interacting components that vary on widely different scales. `NDSolve[ ]` uses the Adams predictor-corrector method to handle non-stiff problems, and backward differentiation formulae (*i.e.*, Gear's method) for stiff problems. `NDSolve[ ]` automatically detects the existence of stiffness and chooses the correct method. The whole process is transparent to the user so that a novice does not even have to understand what stiffness is or recognize the presence of it.

Here is a stiff system of differential equations. The function `y[x]` varies much more rapidly than `z[x]`.

```
In[29]:= NDSolve[{y'[x] == -1000 y[x] + Cos[x],
                 z'[x] == -z[x] + x^2, y[0] == 1, z[0] == 0},
                {y[x], z[x]}, {x, 2}]

Out[29]= {{y[x] -> InterpolatingFunction[{0., 2.}, <>][x],
   z[x] -> InterpolatingFunction[{0., 2.}, <>][x]}}
```

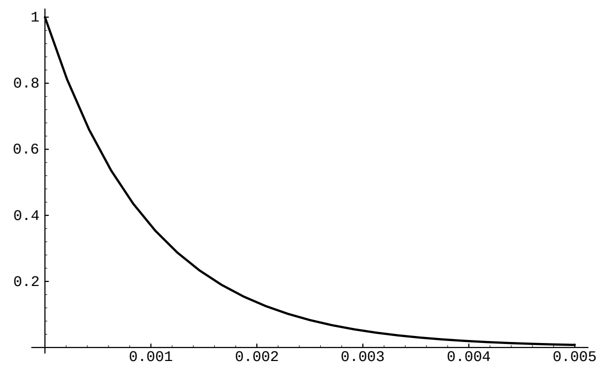
Both components can be plotted together, but only one is visible due to the large difference in scale.

```
In[30]:= Plot[Evaluate[{y[x], z[x]} /. %[[1]]], {x, 0, 2},
              PlotRange -> All]
```



Here is the rapidly varying component, `y[x]`. We need a different scale to display it.

```
In[31]:= Plot[Evaluate[y[x] /. %%[[1]]], {x, 0, 0.005},
                    PlotRange -> All]
```



■ 9.4 Options to NDSolve[ ]

The options `AccuracyGoal` and `PrecisionGoal` to `NDSolve[ ]` control the desired accuracy and precision in the computed solution. Greater accuracy and precision can be requested by setting these options to larger values. `AccuracyGoal` and `PrecisionGoal` are in turn related to the `WorkingPrecision` option. With the default setting `WorkingPrecision -> Automatic`, `NDSolve[ ]` performs numeric computation using machine-precision arithmetic. Bignum arithmetic can be requested by making `WorkingPrecision` greater than machine precision. The default values of `AccuracyGoal` and `PrecisionGoal` are both `Automatic`, which causes their values to be $w - 10$ where w is the `WorkingPrecision`[14].

Leaving the options unspecified causes `NDSolve[ ]` to use machine-precision arithmetic.

```
In[32]:= NDSolve[{y'[x] == y[x], y[0] == 1}, y, {x,2}]

Out[32]= {{y -> InterpolatingFunction[{0., 2.}, <>]}}
```

Comparing this answer with the exact result `Exp[x]` shows that the solution has about six digits of accuracy.

```
In[33]:= Exp[.51] - y[.51] /. %[[1]]

                   -6
Out[33]= -4.85657 10
```

Now repeat the calculation using larger values for `PrecisionGoal` and `AccuracyGoal`.

```
In[34]:= NDSolve[{y'[x] == y[x], y[0] == 1}, y, {x,2},
                 AccuracyGoal -> 10, PrecisionGoal -> 10]

Out[34]= {{y -> InterpolatingFunction[{0., 2.}, <>]}}
```

The solution now has about nine digits of accuracy.

```
In[35]:= Exp[.51] - y[.51] /. %[[1]]

                   -9
Out[35]= -1.36239 10
```

The value of the `WorkingPrecision` option is not automatically increased if `AccuracyGoal` or `PrecisionGoal` are increased, even though more digits are often needed to achieve these goals. As a result, `NDSolve[ ]` may not achieve the requested accuracy and precision if either `AccuracyGoal` or `PrecisionGoal` is larger than `WorkingPrecision`. If the value for `AccuracyGoal` or `PrecisionGoal` is increased, the value of `WorkingPrecision` should be increased as well. It is recommended that `WorkingPrecision` be greater than `AccuracyGoal` and `PrecisionGoal` by at least a few digits.

The value of `WorkingPrecision` must be increased to achieve these goals.

```
In[36]:= NDSolve[{y'[x] == y[x], y[0] == 1}, y, {x, 5},
                 AccuracyGoal -> 14, PrecisionGoal -> 14]

NDSolve::ndtol: Requested tolerances are too small.

Out[36]= {{y -> InterpolatingFunction[{0., 0.}, <>]}}
```

[14] There is an exception to this. Different machines have different precisions for their machine arithmetic. To have some consistency between machines, when machine arithmetic is being used, the value `Automatic` means 6.

In this example, WorkingPrecision -> 20 is sufficient.

```
In[37]:= NDSolve[{y'[x] == y[x], y[0] == 1}, y, {x, 5},
              AccuracyGoal -> 14, PrecisionGoal -> 14,
         WorkingPrecision -> 20]

Out[37]= {{y -> InterpolatingFunction[{0., 5.}, <>]}}
```

The result has about twelve digits of accuracy.

```
In[38]:= Exp[3.5] - y[3.5] /. %[[1]]

                       -12
Out[38]= -4.01457 10
```

It is important to understand that AccuracyGoal and PrecisionGoal are just goals. The actual accuracy and precision achieved may vary from problem to problem.

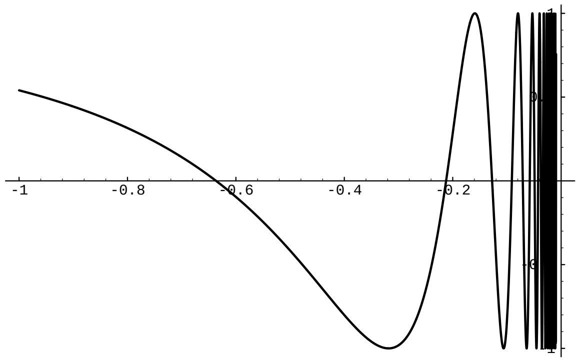
As mentioned before, the option MaxSteps limits the number of steps taken for each calculation. This limit is commonly used to prevent arbitrarily small, never-ending reductions in the step size near singularities.

Without limiting the number of steps, the algorithm would never terminate for this problem.

```
In[39]:= NDSolve[{y'[x] == Sin[1/x] / x^2, y[-1] == Cos[1]},
              y, {x, 0}]

NDSolve::mxst:
   Maximum number of steps reached at the point -0.00927779.

Out[39]= {{y -> InterpolatingFunction[{-1., -0.00927779}, <>]}}
```

This shows the result as far as it was computed.

```
In[40]:= Plot[Evaluate[y[x] /. %[[1]]], {x, -1, -0.0093},
              PlotPoints -> 500]
```



Some calculations require more than the default 500 steps. These calculations can be done by increasing the value of MaxSteps.

Here we attempt to solve the van der Pol equation.

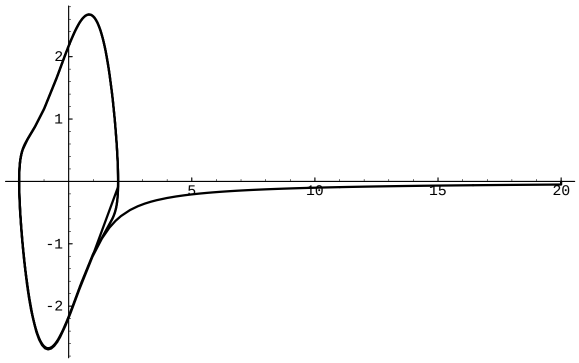
```
In[41]:= NDSolve[{y'[x] == z[x], z'[x] == (1-y[x]^2) z[x] - y[x],
              y[0] == 20, z[0] == 0}, {y[x],z[x]}, {x,0,220}]

NDSolve::mxst:
   Maximum number of steps reached at the point 209.664.

Out[41]= {{y[x] -> InterpolatingFunction[{0., 209.664}, <>][x],
   z[x] -> InterpolatingFunction[{0., 209.664}, <>][x]}}
```

Increasing MaxSteps allows NDSolve to finish the problem.

```
In[42]:= NDSolve[{y'[x] == z[x],
              z'[x] == (1-y[x]^2) z[x] - y[x],
              y[0] == 20, z[0] == 0}, {y[x], z[x]},
         {x, 0, 220}, MaxSteps -> 1000]

Out[42]= {{y[x] -> InterpolatingFunction[{0., 220.}, <>][x],
   z[x] -> InterpolatingFunction[{0., 220.}, <>][x]}}
```

Plot the solution.

```
In[43]:= ParametricPlot[Evaluate[{y[x], z[x]} /. %[[1]]],
             {x, 0, 220}, PlotPoints -> 150,
                    PlotRange -> All]
```



■ 9.5 Indefinite Integration

You could use `F[x_] := NIntegrate[f[t], {t, 0, x}]` to do indefinite integration. But if you want to evaluate the integral at more than a few values of the upper limit of integration you will be doing a lot of redundant work: `NIntegrate[ ]` will integrate from 0 to x for each value of x. A better approach is to use `NDSolve[ ]` for indefinite integration as in `NDSolve[{y[0] == 0, y'[t] == f[t]}, {y}, {t, 0, 5}]`. Because `NDSolve[ ]` accumulates the value of the indefinite integral rather than going back to the lower limit of integration for each new value, it is usually faster to use `NDSolve[ ]` than `NIntegrate[ ]` in doing indefinite numeric integration.

Here is a table of values for a function defined by a numerical integral. A separate integration is done for each point.

```
In[44]:= Timing[Table[NIntegrate[Exp[Sin[t]], {t, 0, x}],
              {x, 0, 20}]]

Out[44]= {4.93333 Second,
  {0, 1.63187, 4.23653, 6.05667, 6.79647, 7.18912, 7.70809,
   8.98387, 11.4548, 13.6363, 14.604, 15.0384, 15.4747,
   16.4506, 18.6425, 21.1067, 22.3721, 22.8878, 23.2812,
   24.0271, 25.859}}
```

The same table can be obtained much faster using `NDSolve`.

```
In[45]:= Timing[sol = NDSolve[{y'[t] == Exp[Sin[t]], y[0] == 0},
                         y[t], {t, 0, 20}];
           Table[y[t] /. sol, {t, 0, 20}]]

Out[45]= {0.466667 Second,
                    -323
  {{5.92879 10    }, {1.63187}, {4.23653}, {6.05668},
   {6.79648}, {7.18913}, {7.70808}, {8.98383}, {11.4548},
   {13.6363}, {14.604}, {15.0384}, {15.4747}, {16.4507},
   {18.6426}, {21.1068}, {22.3722}, {22.8879}, {23.2813},
   {24.0272}, {25.8591}}}
```

■ 9.6 Boundary Value Problems

We mentioned in the introduction that `NDSolve[ ]` cannot solve boundary value problems directly. But we can use it to solve them indirectly. The idea is to define a function that solves an initial value problem using its argument as an initial parameter. By varying that parameter we then try to find what value causes the other boundary condition to be satisfied. The function `FindRoot[ ]` is quite convenient for this application.

As an example consider the following boundary value problem: $y'' = -y$ with $y(0) = 1$ and $y(5) = 2$.

Define the function `f[ ]` to solve an initial value problem, have a side effect of setting `sol` to the solution, and return the value of the solution at 5.

```
In[1]:= f[yp_] := First[sol = y /. NDSolve[{y[0] == 1,
                        y'[0] == yp, y''[t] == -y[t]},
                        {y}, {t, 0, 5}]][5]
```

The solution given by setting $y'(0) = -1$ has $y(5)$ too small.

```
In[2]:= f[-1]

Out[2]= 1.24257
```

But setting $y'(0) = -2$ makes $y(5)$ too large.

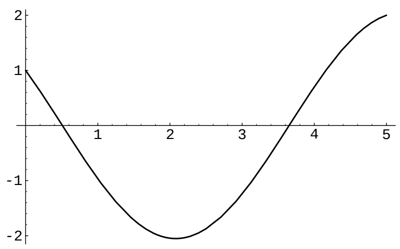
```
In[3]:= f[-2]

Out[3]= 2.2015
```

Find the solution that has $y(5) = 2$.

```
In[4]:= FindRoot[f[yp] == 2, {yp, -1, -2}]

Out[4]= {yp -> -1.78987}
```

Plot the solution on the interval.

```
In[5]:= Plot[First[sol][t], {t, 0, 5}, PlotRange -> All]
```



It should be noted that not all boundary value problems have a solution. For example, if the above problem had been $y'' = -y$ with $y(0) = 1$ and $y(2\pi) = 2$ there would have been no solution since any solution must be periodic with period 2π.

10 Curve Fitting

There are basically two types of curve fitting. These types are distinguished by the kind of data for the problem. If the data consists of a slow algorithm for evaluating a function over a domain and you want to come up with a fast algorithm for evaluating the same function to some prescribed accuracy you should use one of the functions defined in the package `NumericalMath`Approximations`` and discussed in the *Guide to Standard Mathematica Packages*. To adequately cover the use of these functions would require an entire course and we will not go into it here.

The other type of curve fitting involves some data given at specific points and the problem is to find the values of certain parameters in a model function such that the model best approximates the data. Without going into statistics and the theory of how errors are distributed, suffice it to say that we want to find the values of the parameters that are most likely to be the true values assuming that the model is correct and that the errors are independent and normally distributed. If these assumptions are incorrect the results you get may have little or no meaning.

■ 10.1 Fit[ ]

The model for the function `Fit[ ]` is a linear combination of the basis functions specified as the second argument. Thus if you specify the basis functions to be $\{f_1(x), f_2(x), f_3(x)\}$ the implied model is $c_1 f_1(x) + c_2 f_2(x) + c_3 f_3(x)$ and the parameters c_1, c_2, c_3 are the unknowns. The problem is to find the values of the parameters c_i such that the model best fits the data.

Define a randomizer.

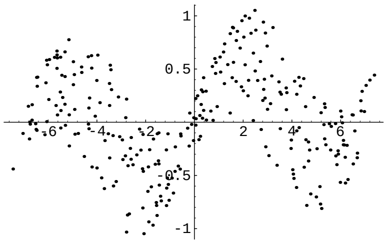
```
In[1]:= r := Random[ ] - 1/2
```

Form some data with some random error.

```
In[2]:= data = Table[{x,BesselJ[1,x]}+r, {x,-7.,7.,.05}];
```

Examine the data.

```
In[3]:= ListPlot[data]
```



Form a set of basis functions for the problem. The 0.44780184487 is to make the zeros of the sin function match those of the Bessel function.

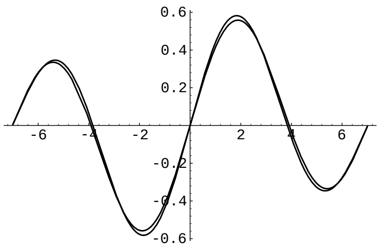
```
In[4]:= basis = Table[Sin[k 0.44780184487 x], {k, 2}]

Out[4]= {Sin[0.44780184487 x], Sin[0.895604 x]}
```

Fit the model to the data.

```
In[5]:= Fit[data, basis, x]

Out[5]= 0.158027 Sin[0.44780184487 x] + 0.442962 Sin[0.895604 x]
```

Plot the model together with the original function.

```
In[6]:= Plot[{%, BesselJ[1, x]}, {x, -7, 7}]
```



■ 10.2 PolynomialFit[ ]

It is quite common to want to fit a polynomial model to some data. Because this case is common and because using simple powers of the independent variable is numerically unstable for many problems, there is the package `NumericalMath`PolynomialFit``. This works much like `Fit[ ]` itself, in fact the former uses the latter, but you don't need to specify an independent variable and the result is returned as a pure function rather than an expression.

Read in the package.

```
In[1]:= << NumericalMath`PolynomialFit`
```

Make a table of data with some random error.

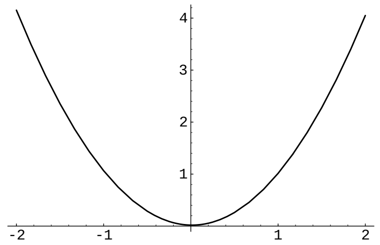
```
In[2]:= data = Table[{x, Random[Real, {-1,1}] + x^2},
                     {x, -2., 2., .05}];
```

Find the best quadratic fit to the data.

```
In[3]:= p = PolynomialFit[data, 2]

Out[3]= FittingPolynomial[<>, 2]
```

Plot the best fit.

```
In[4]:= Plot[p[x], {x, -2, 2}]
```



You can also see the result as an expression in a variable. Note, however, that this is generally a bad thing to do because of numerical instabilities.

```
In[5]:= Expand[p[x]]

                                             2
Out[5]= 0.0194031 - 0.0255713 x + 1.02052 x
```

■ 10.3 NonlinearFit[ ]

If the model used to fit the data is not a simple linear combination of basis functions, the situation gets more complicated. With nonlinear models you still have parameters that determine the function that best fits the data, but those parameters interact in nonlinear ways. The package `Statistics`NonlinearFit`` was written to do nonlinear curve fitting.

`NonlinearFit[`*data*`,` *model*`,` *variables*`,` *parameters*`]`	fit the data to the model with the named variables and parameters, returning the values of the parameters at the fit.

Fitting a nonlinear model to data.

Read in the package.

```
In[1]:= << Statistics`NonlinearFit`
```

Define a randomizer.

```
In[2]:= r := Random[Real, {-.1, .1}]
```

Generate a sample data set to which to fit to a model.

```
In[3]:= data = Table[{x, BesselJ[1,x]+r}, {x,-4.,4.,.25}];
```

Fit the data with paramaters a and b.

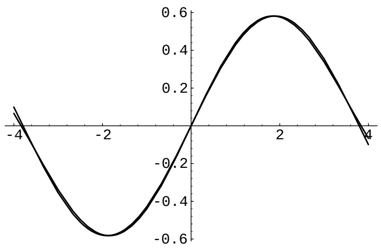
```
In[4]:= NonlinearFit[data, b Sin[a x], x, {a, b}]

Out[4]= {a -> 0.828354, b -> 0.581763}
```

This gives the fitted model.

```
In[5]:= m = b Sin[a x] /. %

Out[5]= 0.581763 Sin[0.828354 x]
```

Show a plot of the model together with that of the original function.

```
In[6]:= Plot[{BesselJ[1,x], m}, {x, -4, 4}]
```



For a more complete description of the package `Statistics`NonlinearFit`` see the *Guide to Standard Mathematica Packages*.

11 Linear Algebra

There is much of linear algebra that can be done symbolically or with matrices and vectors with either symbolic entries or in the context of modular arithmetic. Ignoring these aspects is a grave injustice to both mathematics and *Mathematica*, but this is a course on numerical computation in *Mathematica*. We apologize for this injustice, but realize that it is not possible to cover adequately even the numerical aspects of linear algebra in this short course.

■ 11.1 Vectors and Matrices

Vectors and matrices are represented in *Mathematica* using lists. A vector is represented as a list of elements. A matrix is represented as a list of lists, where each sublist represents a row in the matrix. No distinction is made between row vectors and column vectors.

Any function that returns lists can be used to construct vectors and matrices. The functions `Table` and `Array` are particularly useful for this purpose.

`Table[`*f*`, {`*imax*`}]`	construct a vector consisting of *imax* copies of *f*
`Table[`*f*`, {`*i*`, `*imax*`}]`	construct a length-*n* vector by evaluating *f* for *i* `= 1, 2,` … , *i* `=` *n*
`Table[`*f*`, {`*i*`, `*imin*`, `*imax*`}]`	construct a vector starting with *i* `=` *imin*
`Table[`*f*`, {`*i*`, `*m*`}, {`*j*`, `*n*`}]`	construct an $m \times n$ matrix where *f* is a function of *i* and *j* that gives the value of the i, j^{th} matrix element
`Array[`*f*`, {`*n*`}]`	construct a length-*n* vector with the i^{th} element given by *f*`[`*i*`]`
`Array[`*f*`, {`*m*`, `*n*`}]`	construct an $m \times n$ matrix whose i, j^{th} element is *f*`[`*i*`, `*j*`]`

Functions for constructing general vectors and matrices.

`Range[`*n*`]`	construct the vector `{1, 2, 3, … ,` *n*`}`
`Range[`n_1`, `n_2`]`	construct a vector `{`n_1`, `n_1` + 1, … , `n_2`}`
`Range[`n_1`, `n_2`, `*dn*`]`	construct a vector `{`n_1`, `n_1` + `*dn*`, … , `n_2`}`
`DiagonalMatrix[`*list*`]`	construct a diagonal matrix with the elements of *list* on the diagonal
`IdentityMatrix[`*n*`]`	construct an $n \times n$ identity matrix

Special vectors and matrices.

Construct a vector with five elements.

```
In[1]:= Table[N[Sqrt[n]], {n, 5}]

Out[1]= {1., 1.41421, 1.73205, 2., 2.23607}
```

The same vector can be constructed using `Array`.

```
In[2]:= Array[N[Sqrt[#]] &, {5}]

Out[2]= {1., 1.41421, 1.73205, 2., 2.23607}
```

Here is a 2×2 matrix.

```
In[3]:= Table[1/(m+n), {m, 2}, {n, 2}]

          1  1     1  1
Out[3]= {{-, -}, {-, -}}
          2  3     3  4
```

Here is a vector with three elements.

```
In[4]:= Range[10, 30, 10]

Out[4]= {10, 20, 30}
```

Constuct a 3×3 diagonal matrix from these elements.

```
In[5]:= DiagonalMatrix[%]

Out[5]= {{10, 0, 0}, {0, 20, 0}, {0, 0, 30}}
```

`ColumnForm[`*v*`]`	display the elements of the vector *v* in a column
`MatrixForm[`*m*`]`	display the elements of the matrix *m* in a rectangular array

Displaying vectors and matrices.

The functions `ColumnForm` and `MatrixForm` do not affect the matrices themselves, but only the way in which they are displayed.

Format the vector `Array[f, {3}]` in a column.

```
In[6]:= ColumnForm[Array[f, {3}]]

Out[6]= f[1]
        f[2]
        f[3]
```

Format the matrix `Array[f, {3, 3}]`, allowing the same amount of space for each matrix element.

```
In[7]:= MatrixForm[Array[f, {3, 3}]]

Out[7]//MatrixForm= f[1, 1]   f[1, 2]   f[1, 3]

                    f[2, 1]   f[2, 2]   f[2, 3]

                    f[3, 1]   f[3, 2]   f[3, 3]
```

Many functions are available for working with matrices. The basic function `Part` is used to get individually specified elements. The functions `Drop` and `Take` are used to get elements with indices in specified ranges. The functions `Select` and `Cases` can be used to select elements based on particular criteria. Other useful functions for manipulating matrices include `Transpose`, `Map`, `Apply`, `Join`, `Append`, and `Prepend`.

The package `LinearAlgebra`MatrixManipulation`` contains a collection of functions for working with matrices, and for constructing special matrices. All of the functions in this package are defined in terms of simple combinations of basic operations such as `Part` and `Transpose`. For a description of the functions contained in this package, see the *Guide to Standard Mathematica Packages*. The function definitions in this package also provide instructive examples for constructing your own specialized matrix operations.

Here is a 3×3 matrix.

```
In[8]:= m = Array[a, {3, 3}]; MatrixForm[m]

Out[8]//MatrixForm= a[1, 1]   a[1, 2]   a[1, 3]

                    a[2, 1]   a[2, 2]   a[2, 3]

                    a[3, 1]   a[3, 2]   a[3, 3]
```

This gives the third element in the second row of `m`. Note that `Part[m, 2, 3]` can also be entered as `m[[2, 3]]`.

```
In[9]:= Part[m, 2, 3]

Out[9]= a[2, 3]
```

This adds a fourth row to the matrix.

```
In[10]:= Append[m, {1, 2, 3}] //MatrixForm

Out[10]//MatrixForm= a[1, 1]   a[1, 2]   a[1, 3]

                     a[2, 1]   a[2, 2]   a[2, 3]

                     a[3, 1]   a[3, 2]   a[3, 3]

                     1         2         3
```

This is the definition of TakeColumns from the package LinearAlgebra`MatrixManipulation`. TakeColumns[m, n] gives the first n columns from a matrix m.

```
In[11]:= TakeColumns[mat_?MatrixQ, part_] :=
             Take[#, part]& /@ mat
```

This gives the first two columns of m.

```
In[12]:= TakeColumns[m, 2] //MatrixForm

Out[12]//MatrixForm= a[1, 1]   a[1, 2]
                     a[2, 1]   a[2, 2]
                     a[3, 1]   a[3, 2]
```

Here is another definition from LinearAlgebra`MatrixManipulation`. AppendRows[m_1, m_2, ...] joins the rows in matrices m_1, m_2, ...

```
In[13]:= AppendRows[l__?MatrixQ] :=
             Apply[Join, Transpose[{l}], {1}]
```

This joins the rows in matrix {{1}, {2}, {3}} with the rows in matrix m.

```
In[14]:= AppendRows[{{1}, {2}, {3}}, m] //MatrixForm

Out[14]//MatrixForm= 1         a[1, 1]   a[1, 2]   a[1, 3]
                     2         a[2, 1]   a[2, 2]   a[2, 3]
                     3         a[3, 1]   a[3, 2]   a[3, 3]
```

■ 11.2 Matrix Arithmetic

Addition of matrices, vectors, and scalars is handled automatically by the Listable attribute of the function Plus. The attribute Listable causes Plus to be threaded over arguments that are lists. For example, the input {a, b, c} + x will return {a + x, b + x, c + x}, corresponding to addition of a scalar x to each element of a vector. The input {a + b + c} + {x + y + z} gives {a + b + c + x + y + z}, corresponding to addition of two vectors.

The function Times also has attribute Listable, and can be used for multiplication of a vector or a matrix by a scalar. For example, x {a, b, c} gives {a x, b x, c x}.

The function Dot is used for products of vectors and matrices. The Listable attribute of Times can also be used to multiply matrices, but this will give elementwise multiplication, rather than the conventional matrix or "dot" product. In the product of a vector and a matrix, if the vector appears on the right it is treated as a column vector, and if the vector appears on the left it is treated as a row vector. For convenience, the input A . B . C is equivalent to Dot[A, B, C].

Construct a 2×2 matrix.

```
In[15]:= m = Array[a, {2, 2}]; MatrixForm[m]

Out[15]//MatrixForm= a[1, 1]   a[1, 2]
                     a[2, 1]   a[2, 2]
```

This gives elementwise multiplication of the matrix m and the vector {x, y}.

```
In[16]:= m {x, y} //MatrixForm

Out[16]//MatrixForm= x a[1, 1]   x a[1, 2]
                     y a[2, 1]   y a[2, 2]
```

Here is the dot product of m and {x, y}.

```
In[17]:= m . {x, y}

Out[17]= {x a[1, 1] + y a[1, 2], x a[2, 1] + y a[2, 2]}
```

■ 11.3 Matrix Operations

Matrix Structural Functions

The length of any expression is the number of arguments it has. If the expression is a list and you think of it as a vector then `Length[ ]` gives the dimension of the vector space from which it comes. Note that this is a completely different concept from the norm[15].

The length of this vector is 4.

```
In[1]:= Length[{1, 2, 3, 4}]
Out[1]= 4
```

The length of this expression is also 4.

```
In[2]:= Length[{{1, 2, 3}, {4, 5}, {6}, 7}]
Out[2]= 4
```

The function `Dimensions[ ]` will give you a list of the sizes at each level of any vector, matrix or tensor, down to the depth to which uniformity is preserved. The first dimension is always the length of the outermost list. The second dimension is the length of each of the lists that are also elements of the outermost list. If any of the elements of the outermost list are not themselves lists or if they are not all of the same length then there will be no second dimension.

The first dimension, *i.e.* the length, is 2. There is no second dimension.

```
In[3]:= Dimensions[{1, 2}]
Out[3]= {2}
```

This matrix is 2×3.

```
In[4]:= Dimensions[{{1, 2, 3}, {4, 5, 6}}]
Out[4]= {2, 3}
```

This tensor has dimensions $4 \times 2 \times 3$.

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

Since the second dimension is ambiguous only the first dimension is given.

```
In[6]:= Dimensions[{{1,2}, {3,4,5}}]
Out[6]= {2}
```

The `Depth` of any expression is the number of levels in the tree structure of that expression. It can also be thought of as 1 more than the maximum number of indices needed to access any leaf of the tree. The `TensorRank` of any expression is very similar, except that instead of giving the depth of the deepest leaf it gives the depth to which the expression is a uniformly structured expression with the same head at each level.

The deepest leaves in this expression are `x` and 2.

```
In[7]:= Depth[d = f[f[g[2, 3], g[4, Sin[x^2]]]]]
Out[7]= 6
```

The tensor rank is only 2 because the third level has different heads from the second level.

```
In[8]:= TensorRank[d]
Out[8]= 2
```

Here the third level has the same head as the levels above it, but it is not uniform in size.

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

`TensorRank[`*expr*`]` is the same as `Length[Dimensions[`*expr*`]]`.

The two propositional functions `VectorQ[ ]` and `MatrixQ[ ]` can be used to test whether an expression is a vector or a matrix, respectively. To be a vector an expression must have a head of `List` and none of its elements can be lists. To be a matrix an expression must have a head of `List`, each of its elements must be lists of the same length, and none of their elements can be lists.

[15] For example `Sqrt[v . v]` will give you the Euclidean norm or geometric length of a real vector.

{a, b, c} is a vector.

```
In[10]:= VectorQ[{a, b, c}]

Out[10]= True
```

{{a, b, c}, {1, 2, 3}} is a matrix.

```
In[11]:= MatrixQ[{{a, b, c}, {1, 2, 3}}]

Out[11]= True
```

The functions VectorQ[ ] and MatrixQ[ ] allow a second argument, which can be used to test the individual elements of the vector or matrix for certain properties.

{a, b, c} is not a vector of numbers.

```
In[12]:= VectorQ[{a, b, c}, NumberQ]

Out[12]= False
```

{{a, b, c}, {1, 2, 3}} is a matrix of numbers and symbols.

```
In[13]:= MatrixQ[{{a, b, c}, {1, 2, 3}},
           (NumberQ[#] || Head[#] === Symbol)&]

Out[13]= True
```

There is not a more general test for tensors of various ranks, but we can make such a test quite easily. The features of the test serve to clarify what VectorQ[ ] and MatrixQ[ ] actually do.

Specify a default for the optional third argument.

```
In[14]:= Default[TensorQ, 3] := True&
```

Define the function TensorQ[ ] with an optional test.

```
In[15]:= TensorQ[m_, n_, test_.] :=
             Module[{flatm},
          If[n == 0, Return[Head[m] =!= List && TrueQ[test[m]]]];
                 If[Head[m] =!= List || TensorRank[m] != n,
         Return[False]];
                 flatm = Flatten[m, n-1];
                 !MemberQ[flatm, _List] && (And @@
         (TrueQ[test[#]]& /@ flatm))
                 ]
```

This $3 \times 2 \times 1$ array is a rank 3 tensor.

```
In[16]:= TensorQ[{{{1},{2}}, {{3},{4}}, {{5},{6}}}, 3]

Out[16]= True
```

This 1×3 tensor is not made up of numbers only.

```
In[17]:= TensorQ[{{1, 2, a}}, 2, NumberQ]

Out[17]= False
```

Determinants and Minors

The determinant of a square matrix can be evaluated with Det[ ].

Find the determinant of a numerical matrix.

```
In[1]:= Det[{{1,0,0}, {0,2,0}, {0,0,3}}]

Out[1]= 6
```

Det[ ] also works with symbolic matrices.

```
In[2]:= Det[{{a, b}, {c, d}}]

Out[2]= -(b c) + a d
```

The minors of any given size of any matrix may be found by using the function Minors[ ].

The minors are given as a matrix, the ordering of whose elements is lexicographic. That is, in this example the column ordering is given by $(1,2)$, $(1,3)$, $(1,4)$, $(2,3)$, $(2,4)$, $(3,4)$ and the row ordering is given by $(1,2)$, $(1,3)$, $(2,3)$.

```
In[3]:= Minors[{{1, 2, 3, 0},
         {4, 5, 6, 0},
         {7, 8, 4, 0}}, 2] // MatrixForm

Out[3]//MatrixForm= -3    -6    0    -3    0    0
                    -6    -17   0    -16   0    0
                    -3    -26   0    -28   0    0
```

Inner and Outer Products

The standard inner product for n-tuples of real values is provided by the function `Dot[ ]`. The infix notation `.` is an alternative to the input form.

Here is the dot product of two vectors.

```
In[1]:= {a, b, c} . {1, 2, 3}

Out[1]= a + 2 b + 3 c
```

`Dot[ ]` can also be used for matrix multiplication and tensor multiplication.

```
In[2]:= {{1,2}, {3,4}} . {a, b}

Out[2]= {a + 2 b, 3 a + 4 b}
```

You should be aware that for vectors `Dot[u, v]` is effectively nothing more than `Plus @@ (u v)`. In particular `{1,I} . {1,I}` gives 0. `Dot[ ]` is only an inner product for real vectors. For complex vectors a complex conjugate is needed.

Because *Mathematica* does not distinguish between column and row vectors, `Dot[ ]` can be used equally well to pre-multiply or post-multiply a vector by a matrix. Unfortunately this can lead to minor problems. For example, consider the following matrix product:

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} 5 \\ 6 \end{pmatrix} \begin{pmatrix} 5 & 6 \end{pmatrix} \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} = \begin{pmatrix} 391 & 578 \\ 897 & 1326 \end{pmatrix}$$

This result does not even make any sense.

```
In[3]:= {{1,2},{3,4}} . {5,6} .
        {5,6} . {{1,2},{3,4}}

Out[3]= 319 . {{1, 2}, {3, 4}}
```

One way to get around this difficulty is to explicitly make it clear which vectors are column vectors and which are row vectors.

Now it gets the right result.

```
In[4]:= {{1,2},{3,4}} . {{5},{6}} .
        {{5,6}} . {{1,2},{3,4}}

Out[4]= {{391, 578}, {897, 1326}}
```

One way to explain the difficulty in the previous example is to note that matrix multiplication of a row vector and a column vector is an inner product while matrix multiplication of a column vector and a row vector is an outer product. Given a matrix and a vector there is (up to commutativity) only one way to multiply them so `Dot[ ]` can always do the right thing. But given two vectors there is an outer product and (if the vectors are the same length) an inner product. Representing a column vector as an $n \times 1$ matrix and an row vector as a $1 \times n$ matrix solves the difficulty.

Another way to solve the previous problem is to actually use the generalized outer product provided by *Mathematica*.

The same problem solved using `Outer[ ]` for the outer product.

```
In[5]:= {{1,2},{3,4}} . Outer[Times, {5,6}, {5,6}] .
        {{1,2},{3,4}}

Out[5]= {{391, 578}, {897, 1326}}
```

The function `Outer[ ]` is a generalization of the simple outer product for vectors. In the above example we used `Times[ ]` as the elementary operation. Another useful elementary operation is `D[ ]`, which gives you the Jacobian of a function.

Define a function from $\mathbf{R}^3$ to $\mathbf{R}^2$.

```
In[6]:= f[x_, y_, z_] := {x^2 + y^2 + z^2, x y z}
```

Find the Jacobian of `f[ ]`.

```
In[7]:= Outer[D, f[x,y,z], {x,y,z}] // MatrixForm

Out[7]//MatrixForm= 2 x   2 y   2 z
                    y z   x z   x y
```

If the elementary operation is a function of n variables you need to provide n lists of arguments for `Outer[ ]` to feed to it. The result will be a rank n tensor.

Mathematica also provides a generalized inner product in the function `Inner[ ]`. `Inner[ ]` takes four arguments: two lists of the same length and two elementary operations that are analogous to multiplication and addition in the ordinary inner product.

This gives a generalized inner product.

```
In[8]:= Inner[g, {a, b, c}, {1, 2, 3}, h]

Out[8]= h[g[a, 1], g[b, 2], g[c, 3]]
```

If the "multiplication" operation `g[ ]` is listable then `Inner[g, list1, list2, h]` is equivalent to `h @@ g[list1, list2]`.

Systems of Linear Equations

A common application of linear algebra is in solving systems of linear equations. A system of linear equations can be written in the form $A\boldsymbol{x} = \boldsymbol{b}$, where A is a matrix of coefficients, $\boldsymbol{b}$ is a vector containing the constant term from each equation, and $\boldsymbol{x}$ is the unknown solution vector.

One conceptually straightforward way to solve $A\boldsymbol{x} = \boldsymbol{b}$ is to compute the inverse A^{-1} of A, in which case the answer is given by $\boldsymbol{x} = A^{-1}\boldsymbol{b}$. Although the inverse of a matrix can be computed in *Mathematica* using the function `Inverse`, there are several reasons you should not solve a system by finding the inverse of the coefficient matrix.

If the inverse of the matrix is not needed for some other reason, a more efficient way to solve systems of linear equations is to use the function `LinearSolve`. Internally, `LinearSolve` uses LU decomposition, described below under matrix decompositions, and never constructs an explicit inverse for the coefficient matrix. `LinearSolve` uses the LINPACK subroutine `dgesl` if the coefficients are real numbers, and `cgesl` if the coefficients are complex numbers.

In solving systems of linear equations, a large fraction of the work is spent computing intermediate results that are independent of the right-hand sides (*i.e.* the $\boldsymbol{b}$s) of the equations. If you have several equations of the form $A\boldsymbol{x} = \boldsymbol{b}$ with the same value of A and different values of $\boldsymbol{b}$, a significant amount of time can be saved by keeping track of these intermediate results. In the latest version of *Mathematica* (version 2.1), `LinearSolve` has been extended to accept a matrix as the second argument, so it can handle more than one system of equations at the same time.

Construct a matrix of random real coefficients.

```
In[9]:= m = Table[Random[], {3}, {3}]

Out[9]= {{0.0560708, 0.6303, 0.359894},
  {0.871377, 0.858645, 0.584579},
  {0.742906, 0.391461, 0.357224}}
```

Construct a vector representing the constant terms in the equations.

```
In[10]:= b = Table[Random[], {3}]

Out[10]= {0.380514, 0.928922, 0.175248}
```

This result represents the solution `x` of the equation `m . x == b`.

```
In[11]:= Inverse[m] . b

Out[11]= {1.53472, 5.36885, -8.58454}
```

Here is the same result computed using `LinearSolve`.

```
In[12]:= LinearSolve[m, b]

Out[12]= {1.53472, 5.36885, -8.58454}
```

Construct a vector representing the right-hand side of a second system of equations, and solve the equations.

```
In[13]:= c = {1, 1, 1}; LinearSolve[m, c]

Out[13]= {-2.12984, -6.28264, 14.1135}
```

It is more efficient to solve the equations together when the coefficient matrix `m` is the same for both systems of equations.

```
In[14]:= ((* LinearSolve[m, Transpose[{b,c}]] this is what it
         should be *)
         Transpose[LinearSolve[m, {b, c}]])

Out[14]= {{1.53472, -2.12984}, {5.36885, -6.28264},
  {-8.58454, 14.1135}}
```

This shows that it is indeed the solution.

```
In[15]:= Chop[m . % - Transpose[{b, c}]]

Out[15]= {{0, 0}, {0, 0}, {0, 0}}
```

Note that if you have several vectors that each represent a right-hand side and you wish to combine them to form a matrix for the right-hand side you need to take the transpose since simply forming a list of them results in the vectors becoming rows of the matrix.

RowReduce

With any matrix the columns or rows can be viewed as vectors and we can think of the vector space spanned by those vectors. Clearly there are many sets of vectors that will span the same vector space. Row reduction of a matrix is one way to find the "simplest" set of vectors that span the same vector space as the span of the rows of the given matrix. Clearly, if the matrix is square and the rows are linearly independent the row-reduced form of the matrix is simply the identity matrix. More generally, in the row-reduced form of a matrix, the first non-zero entry in each row is 1 and all entries above and below it are 0. The function `RowReduce[ ]` gives the row-reduced form of any matrix.

Since this matrix is singular the last row reduces to 0 and the last column cannot be zeroed.

```
In[1]:= RowReduce[{{1,2,3},
                   {4,5,6},
                   {7,8,9}}] // MatrixForm

Out[1]//MatrixForm= 1    0    -1
                    0    1    2
                    0    0    0
```

The matrix need not be square for row reduction.

```
In[2]:= RowReduce[{{1,1,1,1,1},
                   {1,2,3,4,5},
                   {2,3,4,6,8}}] // MatrixForm

Out[2]//MatrixForm= 1    0    -1    0    1
                    0    1    2     0    -2
                    0    0    0     1    2
```

The number of non-zero rows is the rank of the matrix. Clearly, it can never exceed the number of columns.

```
In[3]:= RowReduce[{{0,2,3},
                   {4,5,6},
                   {1,1,1},
                   {6,5,4}}] // MatrixForm

Out[3]//MatrixForm= 1   0   0
                    0   1   0
                    0   0   1
                    0   0   0
```

Start with a "nearly" singular matrix.

```
In[4]:= m = {{Pi, 8043944202}, {E, 6960070819}}

Out[4]= {{Pi, 8043944202}, {E, 6960070819}}
```

Roundoff effects can pose a significant problem for numerical linear algebra.

```
In[5]:= RowReduce[N[m]] // MatrixForm

                                         9
Out[5]//MatrixForm= 1           2.56047 10

                    0           0.
```

With sufficiently high precision it is able to see that the matrix is non-singular.

```
In[6]:= RowReduce[N[m, 40]] // MatrixForm

Out[6]//MatrixForm= 1   0
                    0   1
```

NullSpace

The *null space* of a matrix A is the set of vectors $\boldsymbol{x}$ each having the property that $A\boldsymbol{x} = \mathbf{o}$. The function `NullSpace` is used to compute the null space of a matrix. Internally, `NullSpace` uses singular value decomposition, described in the section below on matrix decompositions. `NullSpace` has the option `Tolerance`, which is passed through to the internal code used by `SingularValues` to control which singular values are included.

This matrix has a non-trivial null space since the second column is a multiple of the first.

```
In[7]:= m = {{1.2, 3.6}, {2.4, 7.2}};
```

There is one vector in the null space of `m`.

```
In[8]:= NullSpace[m]

Out[8]= {{-0.948683, 0.316228}}
```

The components of this matrix-vector product are nearly zero. The non-zero result is caused by numerical error.

```
In[9]:= m . %[[1]]

                    -16              -16
Out[9]= {1.36826 10    , 2.73653 10    }
```

Eigensystem, Eigenvectors, and Eigenvalues

The eigenvalues of a matrix A are those values λ_i such that $A - \lambda_i I$ is singular. For each eigenvalue λ_i we have $A\boldsymbol{x}_i = \lambda_i \boldsymbol{x}_i$ where $\boldsymbol{x}_i$ is an eigenvector corresponding to λ_i.

`Eigenvalues[`*m*`]`	a list of the eigenvalues of *m*
`Eigenvectors[`*m*`]`	a list of the eigenvectors of *m*
`Eigensystem[`*m*`]`	a list of the form {*eigenvalues*, *eigenvectors*}

Eigenvalues and eigenvectors.

Start with a simple 2×2 matrix.

```
In[10]:= m = {{1.2, 3.4}, {5.6, 7.8}};
```

Compute the eigenvalues and eigenvectors of the matrix.

```
In[11]:= {vals, vecs} = Eigensystem[m]

Out[11]= {{-0.970832, 9.97083},
  {{-0.842853, 0.538144}, {-0.368675, -0.951055}}}
```

A way to check the result is to verify that `m . vecs[[`*i*`]] == vals[[`*i*`]] vecs[[`*i*`]]`.

```
In[12]:= m . vecs[[1]] == vals[[1]] vecs[[1]]

Out[12]= True
```

Eigenvalues and eigenvectors are computed for matrices of real numbers using the EISPACK subroutine `hqr2`, and for matrices of complex numbers using the EISPACK subroutine `comqr2`. These subroutines use reduction to upper Hessenberg form, followed by a shifted QR decomposition algorithm.

Gram-Schmidt Orthogonalization

In any vector space with an inner product the concept of orthogonality makes sense. Orthogonality in numerical linear algebra is important because it tends to minimize the effects of roundoff errors. The classical method of orthogonalizing a set of vectors is known as Gram-Schmidt orthogonalization. The algorithm for this is implemented in the package `LinearAlgebra`Orthogonalization``.

Read in the package.

```
In[1]:= << LinearAlgebra`Orthogonalization`
```

Find an orthonormal basis for the 3-dimensional subspace of $\mathbf{R}^5$ spanned by some random vectors.

```
In[2]:= GramSchmidt[Table[Random[ ], {3}, {5}]]

Out[2]= {{0.0393878, 0.442764, 0.252813, 0.612112, 0.603169},
  {0.768305, 0.482413, 0.229662, -0.270191, -0.226357},
  {0.369007, -0.642298, 0.450018, 0.455413, -0.203397}}
```

option name	*default value*	
`InnerProduct`	`Dot`	function to use as the inner product for orthogonalization
`Normalized`	True	whether to normalize the vectors in the result

Options for `GramSchmidt`.

`GramSchmidt[ ]` is quite general and can be used to find orthogonal polynomials and orthogonalize vectors in any vector space as well as in $\mathbf{R}^n$. See the *Guide to Standard Mathematica Packages* for further details.

MatrixPower and MatrixExp

`MatrixPower[`m`,` n`]`	n^{th} matrix power
`MatrixExp[`m`]`	matrix exponential

Powers and exponentials of matrices.

Powers and exponentials of matrices occur in a number of applications, such as solving differential equations. Matrix exponentiation is is a notorious problem in numerical analysis, since naive methods for computing the exponential of a matrix can be extremely unstable. `MatrixExp` uses truncated Taylor series of a normalized Schur decomposition, which is among the best known methods. `MatrixPower` uses repeated squaring of the matrix for integer powers, and calls `Eigensystem` for non-integer powers. Negative powers are computed as positive powers of the inverse of the matrix.

This computes the fifth power of a simple matrix.

```
In[3]:= m = {{1.2, 3.4}, {4.5, 6.7}}; MatrixPower[m, 5]

Out[3]= {{10780.6, 18043.8}, {23881.5, 39969.2}}
```

The same result can be computed using repeated dot products, but `MatrixPower` is more efficient, especially for large powers.

```
In[4]:= m . m . m . m . m

Out[4]= {{10780.6, 18043.8}, {23881.5, 39969.2}}
```

■ 11.4 Matrix Decompositions

Various canonical representations of matrices are useful for different applications. *Mathematica* provides a number of standard decompositions for these canonical representations.

LUDecomposition and LUBackSubstitution

The most effective way to solve the matrix equation $A\boldsymbol{x} = \boldsymbol{b}$ is to use Gaussian elimination with partial pivoting followed by back substitution. It can be shown that Gaussian elimination without partial pivoting is equivalent to factoring the matrix A into a lower and an upper triangular matrix. Note that a triangular system of equations, *e.g.*,

$$\begin{aligned} a_{11}x + a_{12}y + a_{13}z &= b_1 \\ a_{22}y + a_{23}z &= b_2 \\ a_{33}z &= b_3 \end{aligned}$$

where the diagonal coefficients a_{ii} are all non-zero, is straightforward to solve. Because of this it is equally easy to solve the system $LU\boldsymbol{x} = \boldsymbol{b}$ where L and U are lower and upper triangular matrices: first solve the equation $L\boldsymbol{y} = \boldsymbol{b}$ for $\boldsymbol{y}$ and then solve $U\boldsymbol{x} = \boldsymbol{y}$. In the real world things become a bit more difficult because of issues of numerical stability. It is often necessary to permute the rows of the matrix and the right-hand side to avoid dividing by 0 or numbers close to 0, so you don't actually get lower and upper triangular matrices. In fact, to save space it is standard practice to store the lower and upper triangular factors in the same array with the zeros being implicit. In spite of this the canonical representation is usually referred to as LU decomposition. If the matrix A is not square or is singular, its LU decomposition may not exist. But if the LU decomposition does exist it can be calculated with a finite number of simple arithmetic operations. The package `LinearAlgebra`GaussianElimination``, described in the *Guide to Standard Mathematica Packages* provides functionality very similar to that of `LUDecomposition[ ]` and `LUBackSubstitution[ ]`, but is more pedagogical in nature and allows for the use of either interval arithmetic or computer arithmetic for its elementary operations.

`LUDecomposition[`*m*`]`	find the LU decomposition of the square, numerical matrix *m*
`LUBackSubstitution[`*lu*`, `*b*`]`	find the solution `x` to the linear system `A.x == b`, where *lu* is the result of `LUDecomposition[A]`

Functions for solving linear systems.

For large systems nearly all of the cost of solving a system is in the decomposition part. Because the two parts are separate, if you have many linear systems with the same matrix you can do the expensive part just once and do the (inexpensive) back substitution for each separate right-hand side.

Start with a matrix.

```
In[1]:= a = Table[Random[], {4},{4}];
```

Find the LU decomposition of `a`. Note that there are two parts. The first part is an array containing both the lower and upper triangular factors and the second part is a record of the row permutations.

```
In[2]:= lu = LUDecomposition[a]

Out[2]= {{{0.871377, -0.723338, -0.413018, -0.0643473},
   {0.391461, 0.833456, -0.697368, -0.361651},
   {0.175248, 0.345947, 0.61529, -0.209071},
   {0.649997, 0.816591, -0.56003, 0.245778}}, {4, 4, 3, 4}}
```

Get a right-hand side for the system.

```
In[3]:= b = Table[Random[], {4}]

Out[3]= {0.200244, 0.17906, 0.853949, 0.902815}
```

Solve the system `a . x == b` for `x`.

```
In[4]:= x = LUBackSubstitution[lu,b]

Out[4]= {-1.97929, 5.58668, 0.0943118, -3.72284}
```

This shows that the solution satisfies the equation.

```
In[5]:= a . x == b

Out[5]= True
```

SingularValues

Another important decomposition is singular value decomposition or SVD. The SVD of a real matrix A is the triple (U, W, V) such that $A = U^{\mathrm{T}}WV$, where U and V are row-orthonormal matrices and W is a diagonal matrix. If complex numbers are involved this becomes $A = U^{\mathrm{H}}WV$ where U^{H} is the conjugate transpose of U. The algorithm for SVD is based on QR iteration using Givens rotations.

Note that most of the literature on singular value decomposition uses a different notation, namely $A = UWV^{T}$. The reason for the change is that the orthogonal vectors that comprise the matrices U and V are sometimes useful individually. By making U and V row-orthonormal rather than column-orthonormal these vectors become the elements of the lists of lists that are U and V.

`SingularValues[`*m*`]`	find the singular value decomposition of the matrix *m*

Singular value decomposition.

Start with a matrix `m`.

```
In[1]:= m = Table[Random[], {3},{5}]

Out[1]= {{0.0560708, 0.6303, 0.359894, 0.871377, 0.858645},
  {0.584579, 0.742906, 0.391461, 0.357224, 0.380514},
  {0.928922, 0.175248, 0.858417, 0.894181, 0.277896}}
```

Find the singular value decomposition of `m`.

```
In[2]:= {u,w,v} = SingularValues[m]

Out[2]= {{{-0.572595, -0.480425, -0.664324},
   {-0.731516, -0.0664674, 0.678577},
   {-0.370162, 0.874513, -0.31338}},
  {2.19733, 0.872537, 0.513323},
  {{-0.423267, -0.379661, -0.438901, -0.575513, -0.390965},
   {0.630888, -0.44873, 0.336048, -0.0623456, -0.532734},
   {0.388374, 0.704137, -0.116675, -0.565671, -0.140573}}}
```

The decomposition is correct.

```
In[3]:= m - Transpose[u] . DiagonalMatrix[w] . v // Chop

Out[3]= {{0, 0, 0, 0, 0}, {0, 0, 0, 0, 0}, {0, 0, 0, 0, 0}}
```

In many applications of singular value decomposition it is desirable to throw away those singular values (*i.e.*, elements of `w`) that are are insignificant, together with the corresponding rows in `u` and `v`. The option `Tolerance` is used to control what is meant by "insignificant".

option name	*default value*	
`Tolerance`	`Automatic`	relative size of the singular values that are kept or rejected

The `Tolerance` option for `SingularValues`.

The value `Automatic` for `Tolerance` means that the tolerance is about 10^{-p+4} where p is the precision of the matrix.

Form a matrix with less than full rank.

```
In[4]:= AppendTo[m, m[[3]]]

Out[4]= {{0.0560708, 0.6303, 0.359894, 0.871377, 0.858645},
  {0.584579, 0.742906, 0.391461, 0.357224, 0.380514},
  {0.928922, 0.175248, 0.858417, 0.894181, 0.277896},
  {0.928922, 0.175248, 0.858417, 0.894181, 0.277896}}
```

Find the singular value decomposition of `m` keeping all non-zero singular values.

```
In[5]:= {u,w,v} = SingularValues[m, Tolerance -> 0]

Out[5]= {{{-0.435408, -0.382793, -0.576146, -0.576146},
   {-0.800028, -0.250459, 0.385503, 0.385503},
   {-0.412765, 0.889235, -0.139437, -0.139437},
                   -17
   {0., 6.7085 10    , 0.707107, -0.707107}},
                                          -18
  {2.66328, 0.995976, 0.524709, 8.8472 10   },
  {{-0.495095, -0.285645, -0.486504, -0.580677, -0.315302},
   {0.527055, -0.557451, 0.27699, -0.097569, -0.570279},
   {0.452883, 0.670048, -0.0759294, -0.55532, -0.178289},
   {-0.0694015, -0.232491, 0.559085, -0.585268, 0.534806}}}
```

The decomposition is correct.

```
In[6]:= m - Transpose[u] . DiagonalMatrix[w] . v // Chop

Out[6]= {{0, 0, 0, 0, 0}, {0, 0, 0, 0, 0}, {0, 0, 0, 0, 0},
  {0, 0, 0, 0, 0}}
```

Find the singular value decomposition of `m` keeping only those singular values that are at least 0.1 times as large as the largest singular value.

```
In[7]:= {u,w,v} = SingularValues[m, Tolerance -> 0.1]

Out[7]= {{{-0.435408, -0.382793, -0.576146, -0.576146},
   {-0.800028, -0.250459, 0.385503, 0.385503},
   {-0.412765, 0.889235, -0.139437, -0.139437}},
  {2.66328, 0.995976, 0.524709},
  {{-0.495095, -0.285645, -0.486504, -0.580677, -0.315302},
   {0.527055, -0.557451, 0.27699, -0.097569, -0.570279},
   {0.452883, 0.670048, -0.0759294, -0.55532, -0.178289}}}
```

The decomposition is still correct even though it is quite different.

```
In[8]:= m - Transpose[u] . DiagonalMatrix[w] . v // Chop

Out[8]= {{0, 0, 0, 0, 0}, {0, 0, 0, 0, 0}, {0, 0, 0, 0, 0},
  {0, 0, 0, 0, 0}}
```

Note that all matrices have an SVD, but calculating it is like solving for the roots of a polynomial: you can do it exactly only for very small or very special matrices. In general, while the SVD of a large, exact matrix exists it cannot be found exactly. The algorithm that *Mathematica* uses is an iterative one involving Givens rotations that converges to the mathematical SVD.

PseudoInverse

The pseudo-inverse (or Moore-Penrose inverse) is a generalization of inverse and applies to all matrices. The pseudo-inverse X of any matrix A satisfies the four Moore-Penrose conditions

$$AXA = A \qquad XAX = X \qquad (AX)^{\mathrm{T}} = AX \qquad (XA)^{\mathrm{T}} = XA$$

For complex matrices the transpose is replaced by the conjugate transpose.

One important use of singular value decomposition is in the problem of linear, least squares curve fitting. If A has more rows than columns then in general the system $A\boldsymbol{x} = \boldsymbol{b}$ is overdetermined and has no solution. By multiplying both sides of the overdetermined system by the pseudo-inverse of A we get the "best" approximate solution $\boldsymbol{x} = A^{(-1)}\boldsymbol{b}$ where "best" means that the Euclidean length of the residual $A\boldsymbol{x} - \boldsymbol{b}$ is as short as possible.

Mathematica provides the function `PseudoInverse` for approximately solving overdetermined systems. For linear, least squares curve fitting, however, you should use the built-in function `Fit[ ]`, or `PolynomialFit[ ]` (defined in the package `NumericalMath`PolynomialFit``) for polynomial fitting.

The way `PseudoInverse` works for inexact matrices is to first find the singular value decomposition $A = U^{\mathrm{T}}WV$ and then return $A^{(-1)} = V^{\mathrm{T}}W^{-1}U$. The inverse of W is known to exist because the small singular values of W have been discarded. Like `SingularValues`, `PseudoInverse` has the option `Tolerance`, which is simply passed through to the algorithm for `SingularValues` to control which singular values are included.

option name	*default value*	
`Tolerance`	`Automatic`	relative size of the singular values that are kept or rejected

The `Tolerance` option for `PseudoInverse`.

Start with a 3×4 matrix of rank 2.

```
In[1]:= a = {{1,2,3,4},{3,1,3,2},{3,1,3,2}};
```

Find the pseudo-inverse.

```
In[2]:= (x = PseudoInverse[a]) // MatrixForm
```

`Out[2]//MatrixForm=`
$$\begin{matrix} -(\frac{43}{206}) & \frac{17}{103} & \frac{17}{103} \\ \frac{12}{103} & -(\frac{7}{206}) & -(\frac{7}{206}) \\ \frac{3}{206} & \frac{6}{103} & \frac{6}{103} \\ \frac{24}{103} & -(\frac{7}{103}) & -(\frac{7}{103}) \end{matrix}$$

The matrix `x` satisfies the 4 Moore-Penrose conditions.

```
In[3]:= {a.x.a == a, x.a.x == x,
         Transpose[a.x] == a.x,
         Transpose[x.a] == x.a}

Out[3]= {True, True, True, True}
```

QRDecomposition

QR decomposition decomposes a matrix into the product of a column-orthonormal matrix and an upper triangular matrix. In particular we get the matrices Q and R such that $A = Q^{\mathrm{T}} R$ for real matrices or $A = Q^{\mathrm{H}} R$ for complex matrices. As with singular value decomposition this notation is different from most of the literature on QR decomposition. For inexact matrices the algorithm uses Householder transformations to get the decomposition.

Start with a matrix.

```
In[4]:= a = Table[Random[], {5},{3}];
```

Find the QR decomposition of `a`.

```
In[5]:= {q,r} = QRDecomposition[a];
```

The decomposition is correct.

```
In[6]:= a == Transpose[q] . r

Out[6]= True
```

If `a` is full rank the elements of `q` form an orthonormal basis for the range of `a`. However, if `a` is rank deficient then the QR factorization may not give a basis. The way to avoid this problem is to allow the columns of `a` to be permuted. This is done by setting the option `Pivoting` to `True`. The default value is `False`, but when it is set to true `QRDecomposition` will give three matrices: `{q, r, p}` such that `a` is `Conjugate[Transpose[q]].r.Inverse[p]`.

option name	*default value*	
`Pivoting`	`False`	whether to include a permutation matrix in the result for numerical stability

The `Pivoting` option for `QRDecomposition`.

Start with a 4 × 3 matrix of rank 2.

```
In[7]:= a = N[{{1,1,1},{0,0,1},{0,0,1},{0,0,1}}];
```

Without pivoting we do not get a basis for the range of `a`.

```
In[8]:= QRDecomposition[a][[1]]

Out[8]= {{-1., 0., 0., 0.}, {0., 1., 0., 0.},
  {0., 0., -0.707107, -0.707107}}
```

With pivoting it gives a basis.

```
In[9]:= {q,r,p} = QRDecomposition[a, Pivoting -> True]; q

Out[9]= {{-0.5, -0.5, -0.5, -0.5},
  {0.866025, -0.288675, -0.288675, -0.288675}}
```

The decomposition is correct.

```
In[10]:= Chop[Conjugate[Transpose[q]] . r . Inverse[p]]

Out[10]= {{1., 1., 1.}, {0, 0, 1.}, {0, 0, 1.}, {0, 0, 1.}}
```

SchurDecomposition

The Schur decomposition of a square matrix is a factorization of the matrix with the eigenvalues on the diagonal of one of the factors. Since there is no closed form for the roots of a general polynomial of degree 5 or more it is not possible to give this decomposition for large, exact matrices except in special cases. The algorithm for Schur decomposition uses QR iteration and converges to the result for inexact matrices.

The Schur decomposition of a square matrix A is the factorization $A = QTQ^{\mathrm{T}}$ where T is upper triangular with the eigenvalues of A on its diagonal. If A is a complex matrix the transpose is replaced by the conjugate of the transpose.

Start with a square matrix.

```
In[11]:= a = Table[Random[], {5},{5}];
```

Find the Schur decomposition of `a`.

```
In[12]:= {q,t} = SchurDecomposition[a];
```

The decomposition is correct.

```
In[13]:= a == q . t. Transpose[q]

Out[13]= True
```

Like `QRDecomposition`, `SchurDecomposition` has the option `Pivoting`. In `SchurDecomposition` the purpose is numerical stability. With pivoting `SchurDecomposition` gives the list `{q,t,d}` where `a` is `d.q.t.Transpose[q].Inverse[d]`.

option name	*default value*	
`Pivoting`	`False`	whether to balance the matrix before proceeding and exchange columns as needed for numerical stability

The `Pivoting` option for `SchurDecomposition`.

Start with a very unbalanced matrix.

```
In[14]:= m = {{10.^20, 5. 10.^20}, {1,2}};
```

Find the Schur decomposition of `m`.

```
In[15]:= {q,t} = SchurDecomposition[m];
```

The second row is completely wrong due to numerical instability.

```
In[16]:= (q . t . Transpose[q])/m - 1

                             -16
Out[16]= {{0., -1.11022 10      }, {-1., 0.}}
```

Find the Schur decomposition using `Pivoting -> True`.

```
In[17]:= {q,t,d} = SchurDecomposition[m, Pivoting -> True];
```

Now the decomposition is correct.

```
In[18]:= m == d . q . t . Transpose[q] . Inverse[d]

Out[18]= True
```

JordanDecomposition

The Jordan canonical form is another useful canonical decomposition of square matrices. Among things for which it can be used are evaluating `MatrixExp[ ]` and other power series expansions of functions with matrix arguments. In particular `JordanDecomposition[A]` gives `{S, J}` such that $A = SJS^{-1}$. The eigenvalues of A appear down the diagonal of J and J is bi-diagonal with only 1s and 0s appearing in the super-diagonal. The matrix S is not orthogonal as in Schur decomposition, but represents rather a similarity transformation.

The important property of Jordan decomposition for power series expansions is that if

$$f(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots$$

then

$$f(A) = f(SJS^{-1}) = S(a_0 + a_1 J + a_2 J^2 + a_3 J^3 + \cdots)S^{-1}$$

and the power series expansion at J can be easily evaluated in terms of f and its derivatives at the various eigenvalues.

Find the Jordan decomposition of a matrix.

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

Out[1]= {{{1, -((-4 (-48 + 12 (-13 + Sqrt[297])) +
             6 (-32 + (-13 + Sqrt[297]) (-5 + Sqrt[297]))) /
           ((-13 + Sqrt[297])
             (-32 + (-13 + Sqrt[297]) (-5 + Sqrt[297])))),
        -((-4 (-48 + 12 (-13 - Sqrt[297])) +
             6 (-32 + (-13 - Sqrt[297]) (-5 - Sqrt[297]))) /
           ((-13 - Sqrt[297])
             (-32 + (-13 - Sqrt[297]) (-5 - Sqrt[297]))))},
                      -48 + 12 (-13 + Sqrt[297])
       {-2, -(-----------------------------------------),
              -32 + (-13 + Sqrt[297]) (-5 + Sqrt[297])
                -48 + 12 (-13 - Sqrt[297])
        -(-----------------------------------------)}, {1, 1, 1}},
          -32 + (-13 - Sqrt[297]) (-5 - Sqrt[297])
                        15 - Sqrt[297]               15 + Sqrt[297]
      {{0, 0, 0}, {0, --------------, 0}, {0, 0, --------------}}}
                              2                            2
```

12 Compiled Evaluation

Many of the numerical functions discussed in this course make use of compiled evaluation. This appendix discusses what it means to “compile” a function in *Mathematica* and how you can make the best use of it.

12.1 Using Compile and CompiledFunction

The system for working with compiled expressions consists of a compiler for translating expressions into "pseudocode", and a pseudocode interpreter. The "compilation" is not true compilation to machine level instructions (which would make compiled code nontransportable), but rather to operation codes for a virtual computer that *Mathematica* runs. This virtual computer is the same on all actual computers that run *Mathematica*. The compiler can be accessed using the `Compile` function, which returns a `CompiledFunction` object containing a pseudocode represention of the input. The `CompiledFunction` object is used by applying it to an appropriate list of arguments.

The functions `FindMinimum`, `FindRoot`, `NDSolve`, `NIntegrate`, `NProduct`, and `NSum` all include the option `Compiled`, with default value `Compiled -> True`. The first argument in these functions is normally compiled unless the `Compiled` option is set to `Compiled -> False`. Compilation is handled by an internal call to `Compile`. Numerical values are obtained with the corresponding `CompiledFunction` object rather than the original input.

`Compile[{`x_1`, `x_2`, ... }, `*expr*`]`	construct a `CompiledFunction` object corresponding to *expr*, with arguments x_i assumed to be of type `_Real` (real numbers)
`Compile[{{`x_1`, `t_1`}, {`x_2`, `t_2`}, ... }, `*expr*`]`	compile *expr* assuming that x_i is of type t_i, where t_i is `True|False` (*i.e.*, Boolean), `_Integer`, `_Real`, or `_Complex`
`Compile[`*vars*`, `*expr*`, {{`p_1`, `pt_1`}, ... }]`	compile *expr*, assuming subexpressions that match p_i are of type t_i

Creating `CompiledFunction` objects.

`CompiledFunction[`*args*`, `*regs*`, `*code*`, `*func*`]`	compiled function with argument list *args*, pseudocode instructions *code* using registers *regs*, and with pure function *func* corresponding to the original input

The structure of a `CompiledFunction` object.

The main purpose of function compilation is to improve the speed of function evaluation. This speedup is achieved in part by taking advantage of commonly available hardware and software optimizations for working with certain types of objects, such as integers and real numbers. The four types of objects supported by the *Mathematica* compiler are Boolean expressions (type `True|False`), machine-size integers (type `_Integer`, represented in C using `int`), machine-size real numbers (type `_Real`, represented in C using `double`), and machine-size complex numbers (type `_Complex`, with real and imaginary parts represented using `double`).

Use `Compile` to construct a `CompiledFunction` representation of the expression `Log[Exp[x] + 1]` with argument `x`.

```
In[1]:= f = Compile[{x}, Log[Exp[x] + 1]]

Out[1]= CompiledFunction[{x}, Log[Exp[x] + 1], -CompiledCode-]
```

Evaluate the function `f` with a machine-size real number as an argument.

```
In[2]:= f[2.7]

Out[2]= 2.76504
```

If the argument is not a machine-size real number, it is converted to a machine-size number whenever possible.

```
In[3]:= f[Pi]

Out[3]= 3.1839
```

This shows the information contained in a `CompiledFunction` object.

```
In[4]:= InputForm[f]

Out[4]//InputForm=
  CompiledFunction[{_Real}, {0, 1, 5, 0},
   {{1, 17}, {4, 1, 0}, {47, 0, 1}, {12, 1, 0}, {20, 0, 2},
    {33, 1, 2, 3}, {48, 3, 4}, {8, 4}},
   Function[{x}, Log[Exp[x] + 1]]]
```

Here is a function with one real argument and one complex argument.

```
In[5]:= g = Compile[{{x, _Real}, {z, _Complex}}, x + Re[z]]

Out[5]= CompiledFunction[{x, z}, x + Re[z], -CompiledCode-]
```

Note that although the input arguments were exact, the result is an inexact real number.

```
In[6]:= g[2/3, 4 + 5 I]

Out[6]= 4.66667
```

The third argument in `Compile` is used to give a list of pairs {*pat*, *type*} indicating the type to assume for subexpressions matching the pattern *pat*. The type can be `True|False`, `_Integer`, `_Real`, `_Complex`, or `Null`. Notice that `_Integer`, `_Real`, and `_Complex` are patterns while `True|False` and `Null` values. This feature is useful primarily for subexpressions that will be evaluated outside the pseudocode interpreter, since the compiler is not always able to determine the correct type at compile time.

Since the `BesselJ[ ]` cannot be compiled, the compiler doesn't know what type its value will be when it gets evaluated. Here we tell it to expect a `Real` value.

```
In[7]:= f = Compile[x, E^BesselJ[2, x], {{BesselJ[_,_], _Real}}]

                                  BesselJ[2, x]
Out[7]= CompiledFunction[{x}, E              , -CompiledCode-]
```

The Bessel function is evaluated by standard evaluation, but the exponential function is evaluated by the pseudocode interpreter.

```
In[8]:= f[2.3]

Out[8]= 1.51273
```

The third argument can also include specifications of the form {pat_1, *type*, pat_2}, where pat_2 is a pattern used to indicate subexpressions that should not be compiled. In many cases, even when the compiler encounters a function that will be evaluated by standard evaluation, it still tries to compile arguments to that function. The second pattern pat_2 can be used in cases when this compilation is undesirable.

In this example, the arguments in `Print` are compiled, even though the `Print` function itself is not. Since no type is given, the compiler assumes that `var` will evaluate to a real number.

```
In[9]:= f = Compile[x, Print[var, Sin[Cos[x]]]]

Out[9]= CompiledFunction[{x}, Print[var, Sin[Cos[x]]],
  -CompiledCode-]
```

If `var` is a real number, compiled evaluation works as expected.

```
In[10]:= var = 12345; f[7]

12345.0.684489
```

An error occurs if `var` does not evaluate to a real number.

```
In[11]:= var = "string  "; f[7]

CompiledFunction::ccr:
   Expression string   should be a machine-size real number.

CompiledFunction::cfex:
   External evaluation error at instruction 3
    ; proceeding with uncompiled evaluation.
string  Sin[Cos[7]]
```

This instructs the compiler not to compile functions arguments for subexpressions that match `Print[_,_]`.

```
In[12]:= f = Compile[x, Print[var, Sin[Cos[x]]],
          {{Print[___], Null, Print[_,_]}}]

Out[12]= CompiledFunction[{x}, Print[var, Sin[Cos[x]]],
  -CompiledCode-]
```

Compiled evaluation now works without error. Note that `Sin[Cos[x]]` is not compiled, and gives an exact answer computed using standard evaluation.

```
In[13]:= var = "string "; f[7]

string Sin[Cos[7]]
```

12.2 Errors

The current *Mathematica* compiler is intended primarily for computations involving numerical computations and procedural programming. If the compiler encounters an expression it is not specifically designed to handle, it will still return a valid `CompiledFunction` object, but the corresponding pseudocode will include instructions that do standard evaluation. Although this situation is not strictly an error, it is normally more efficient in such cases to avoid the compiler and use standard evaluation in the first place.

`CompiledFunction` objects make implicit assumptions about the type for each subexpression and each function argument. These types can be specified in the arguments to `Compile`, or are determined automatically based on simple heuristics, with the default being that most subexpressions and arguments are assumed to have type `_Real`.

In general, if an argument or subexpression in a `CompiledFunction` object does not match the expected type, a result is computed using uncompiled evaluation. When this happens, the pure function that appears as the last element in the `CompiledFunction` object is applied to the input arguments, and the expression is evaluated normally.

If compiled evaluation fails for any reason, the action taken depends on whether, when the pseudocode was compiled, the compiler was invoked from a *Mathematica* program or the keyboard, or was called internally by a function having the `Compiled -> True` option. If the compiler was called from a *Mathematica* program or explicitly from the keyboard, a warning message explaining the problem is generated, and evaluation proceeds using uncompiled evaluation. If the compiler was called internally, the message is normally suppressed, and if compiled evaluation fails twice in a row the remainder of the computation is completed using uncompiled evaluation.

There are several common situations in which compiled evaluation can fail. If the original compilation fails, or a `CompiledFunction` object is constructed incorrectly, compiled evaluation will almost certainly fail. Most other failures are due to type conflicts, such as a real number being used in place of an integer. Even if the number is nominally of the correct type, an error will occur if the range of machine-size numbers is exceeded, either on input or as an intermediate result. Another class of type conflicts comes up when a value lies outside some implicit range. For example, if x is a real number, the compiler assumes that $\sqrt{x}$ is also real, and compiled evaluation will fail for $x < 0$.

The compiler cannot handle expressions with a head of `List`.

```
In[14]:= f = Compile[{x}, {1+x, 2+x}]

Compile::cph:
   Expression {1 + x, 2 + x} with head List cannot be compiled.

Out[14]= CompiledFunction[{x}, {1 + x, 2 + x}, -CompiledCode-]
```

Evaluation still gives the expected result, but the calculation is not very efficient.

```
In[15]:= f[3]

CompiledFunction::cfn:
   Numerical error encountered at instruction 3
    ; proceeding with uncompiled evaluation.

Out[15]= {4, 5}
```

Here is a function with one Boolean argument and one integer argument.

```
In[16]:= g = Compile[{{b, True|False}, {t, _Integer}},
                   If[b, t, t-16, t^2]]

                                                     2
Out[16]= CompiledFunction[{b, t}, If[b, t, t - 16, t ],
  -CompiledCode-]
```

Evaluate the function g with two arguments.

```
In[17]:= g[False, 9]

Out[17]= -7
```

If the types of the arguments do not match the types in the `CompiledFunction` object, a message is generated and the uncompiled expression is evaluated.

```
In[18]:= g[b, 5]

CompiledFunction::cftf:
   Argument b at position 1 should be True or False.

Out[18]= 25
```

Here is a function that multiplies two integers.

```
In[19]:= f = Compile[{{x, _Integer}, {y, _Integer}}, x y]

Out[19]= CompiledFunction[{x, y}, x y, -CompiledCode-]
```

On this computer, the result of the multiplication is too large to be represented as a machine-size integer. A warning message is generated, and the result is computed using uncompiled evaluation.

```
In[20]:= f[2^30, 2^30]

CompiledFunction::cfn:
   Numerical error encountered at instruction 4
    ; proceeding with uncompiled evaluation.

Out[20]= 1152921504606846976
```

In this example, since no types are specified, the compiler assumes that both `x` and `ArcSin[x]` are machine-size real numbers.

```
In[21]:= g = Compile[{x}, ArcSin[x]]

Out[21]= CompiledFunction[{x}, ArcSin[x], -CompiledCode-]
```

An error occurs if the argument lies outside the range in which `ArcSin[x]` is a real-valued function.

```
In[22]:= g[2.3]

CompiledFunction::cfn:
   Numerical error encountered at instruction 3
    ; proceeding with uncompiled evaluation.

Out[22]= 1.5708 - 1.47504 I
```

In this example, the compiler treats `y` as a global variable, and assumes that it will evaluate to a real number when the `CompiledFunction` object is used.

```
In[23]:= g = Compile[{x}, x + y]

Out[23]= CompiledFunction[{x}, x + y, -CompiledCode-]
```

Compiled evaluation fails if `y` does not evaluate to a real number.

```
In[24]:= g[2.7]

CompiledFunction::ccr:
   Expression y should be a machine-size real number.

CompiledFunction::cfex:
   External evaluation error at instruction 3
    ; proceeding with uncompiled evaluation.

Out[24]= 2.7 + y
```

Now compiled evaluation proceeds normally.

```
In[25]:= y = 2; g[2.7]

Out[25]= 4.7
```

■ 12.3 Performance Considerations When Using Compile

The compiler is specifically designed to handle about a hundred *Mathematica* functions, and typically uses standard evaluation for functions it doesn't recognize. The speed improvements for compiled evaluation over uncompiled evaluation are greatest when the evaluation is handled entirely within the pseudocode interpreter and there are few if any operations that require standard evaluation.

Here is a function that might be used when making a picture of the Mandelbrot set.

```
In[26]:= iterationCount[c_, limit_] :=
                Module[{z=0, n=0},
                        While[Abs[z]<2 && n<limit,
                                z = z^2 + c;
                                n++];
                         n]
```

Use `Timing` to measure the efficiency of this function.

```
In[27]:= Table[iterationCount[x, 1000],
              {x, .3655 +.1 I, .366 + .1 I,
              .00001}] // Timing

Out[27]= {86.5333 Second,
  {290, 417, 541, 1000, 713, 607, 520, 368, 327, 522, 271, 205,
   567, 316, 160, 197, 154, 228, 142, 147, 168, 200, 1000,
   1000, 794, 686, 349, 231, 324, 182, 480, 194, 170, 156, 418,
   307, 480, 625, 680, 306, 255, 938, 179, 240, 130, 127, 123,
   119, 116, 115, 115}}
```

Here is the same program expressed as a compiled function.

```
In[28]:= compiledIterationCount =
        Compile[{{c, _Complex}, {limit, _Integer}},
              Module[{z=I, n=0}, z=0;
                    While[Abs[z]<2 && n<limit,
                          z = z^2 + c;
                          n++];
                    n]]

Out[28]= CompiledFunction[{c, limit},
  Module[{z = I, n = 0}, z = 0;

                                              2
    While[Abs[z] < 2 && n < limit, z = z  + c; n++]; n],
  -CompiledCode-]
```

In this example, compiled evaluation is over twenty times faster than uncompiled evaluation.

```
In[29]:= Table[compiledIterationCount[x, 1000],
              {x, .3655 +.1 I, .366 + .1 I,
              .00001}]; // Timing

Out[29]= {2.1 Second, Null}
```

The speedup in this example is achieved because the program consists entirely of operations that the compiler is designed to handle. The boxes below summarize the set of *Mathematica* functions that can be handled without resorting to standard evaluation.

Plus	Times	Power
Minus	Subtract	Divide
Sqrt	Exp	Log
Sin	Cos	Tan
ArcSin	ArcCos	ArcTan
Csc	Sec	Cot
ArcCsc	ArcSec	ArcCot
Sinh	Cosh	Tanh
ArcSinh	ArcCosh	ArcTanh
ArcCsch	ArcSech	ArcCoth
Round	Floor	Ceiling
N	Sign	Abs
Max	Min	Re
Im	Conjugate	Arg
Mod	Quotient	

Numerical functions.

If	Switch	Which
Equal	Unequal	Less
LessEqual	Greater	GreaterEqual
SameQ	EvenQ	OddQ
And	Or	Not
Module	Block	With
Set	AddTo	SubtractFrom
TimesBy	DivideBy	Increment
PreIncrement	Decrement	PreDecrement
Sum	Product	Do
For	While	CompoundExpression
Break	Continue	Return
Goto	Label	
Throw	Catch	

Control structures, predicates, and other functions.

If compiled evaluation fails, and evaluation continues by evaluating the uncompiled expression, the overhead associated with compiled evaluation and with the original compilation is wasted. This overhead is typically less than a factor of two except in contrived examples. Still, in functions that include the `Compiled` option, the compiler should be turned off using `Compiled -> False` if it is known that compiled evaluation will fail.

Even when compiled evaluation succeeds, the benefits of compilation are lost if standard evaluation is used to compute intermediate results. Although the ability of the pseudocode interpreter to access the full functionality of *Mathematica* through standard evaluation is a great asset, this feature should for efficiency reasons be used sparingly.

The most effective way to determine whether or not compiled evaluation will use standard evaluation is to look at the pseudocode contained in the `CompiledFunction` object. Referring to the list of pseudocode instructions in the next section, instructions numbers 22 through 31 are seen to correspond to standard evaluation. A crude way to predict the efficiency of compiled evaluation is to check for these instructions in the `CompiledFunction` object.

A rudimentary understanding of the pseudocode generated by the *Mathematica* compiler is very helpful both in deciding when to use the compiler, and in improving the efficiency of compiled evaluation. For this purpose, it is instructive to work through some simple examples.

This function adds two arguments of type `_Real`.

```
In[30]:= InputForm[Compile[{x, y}, x + y]]

Out[30]//InputForm=
  CompiledFunction[{_Real, _Real}, {0, 0, 3, 0},
   {{1, 17}, {4, 1, 0}, {4, 2, 1}, {33, 0, 1, 2}, {8, 2}},
   Function[{x, y}, x + y]]
```

In the above example the register specification `{0, 0, 3, 0}` indicates that compiled evaluation uses three real registers. The first instruction, `{1, 17}`, checks the compiler version. The second instruction, `{4, 1, 0}`, loads the first argument into register `0`, and the third instruction, `{4, 2, 1}`, loads the second argument into register `1`. The instruction `{33, 0, 1, 2}` adds the contents of register `0` to the contents of register `1`, and puts the result in register `2`. The last instruction, `{8, 2}`, returns the contents of register `2`.

```
In[31]:= InputForm[Compile[{x}, temp = 25.0 * x; x + temp]]

Out[31]//InputForm=
  CompiledFunction[{_Real}, {0, 0, 5, 0},
   {{1, 17}, {4, 1, 0}, {13, 25., 1}, {36, 1, 0, 2},
    {26, Function[{x}, temp = 25.*x], 17},
    {24, Function[{x}, temp], 3}, {33, 0, 3, 4}, {8, 4}},
   Function[{x}, temp = 25.*x; x + temp]]
```

In this example the first four instructions check the version number, load the argument `x`, load the constant `25.0`, and put the result of `25.0 * x` in register 2. The next two instructions involve standard evaluation. The intstruction `{26, Function[{x}, temp = 25.*x], 17}` evaluates the assignment `temp = 25.*x`, and the instruction `{24, Function[{x}, temp], 3}` puts the result of evaluating `Function[{x}, temp]` in register 3. Standard evaluation is needed here because `temp` is a global variable, and the compiler has no way to determine whether additional rules will be invoked by this evaluation.

```
In[32]:= InputForm[Compile[{x}, Block[{temp = 25.0 * x}, x +
        temp]]]

Out[32]//InputForm=
  CompiledFunction[{_Real}, {0, 0, 4, 0},
   {{1, 17}, {4, 1, 0}, {13, 25., 1}, {36, 1, 0, 2},
    {33, 0, 2, 3}, {8, 3}},
   Function[{x}, Block[{temp = 25.*x}, x + temp]]]
```

Here is the same program written with `temp` as a local variable. This result is more efficient than the previous one, since there are no instructions corresponding to standard evaluation. Rules associated with local variables in a `Block` are temporarily cleared, so evaluation of `temp` outside the compiler is not needed. The value of `temp` is saved in one of the registers used by the pseudocode interpreter.

■ 12.4 Listing of Pseudocode Instructions

The pseudocode instructions described below correspond to the compiler included with *Mathematica* Version 2.0 and Version 2.1.

`{0}`	do nothing (any arguments are ignored)

The null operation.

`{1, 17}`	specify instruction set 17 (this instruction set)

Specifying the instruction set.

`{2,` *n*`,` *boolreg*`}`	load argument number *n* into boolean register *boolreg*
`{3,` *n*`,` *intreg*`}`	load argument number *n* into integer register *intreg*
`{4,` *n*`,` *realreg*`}`	load argument number *n* into real register *realreg*
`{5,` *n*`,` *cxreg*`}`	load argument number *n* into complex register *cxreg*

Loading function arguments into registers.

`{6,` *boolreg*`}`	return the contents of Boolean register *boolreg*
`{7,` *intreg*`}`	return the contents of integer register *intreg*
`{8,` *realreg*`}`	return the contents of real register *realreg*
`{9,` *cxreg*`}`	return the contents of complex register *cxreg*
`{10}`	return *Null*

Returning results.

`{11,` *bool*`,` *boolreg*`}`	load logical constant *bool* into Boolean register *boolreg*
`{12,` *int*`,` *intreg*`}`	load integer constant *int* into integer register *intreg*
`{13,` *real*`,` *realreg*`}`	load real constant *real* into real register *realreg*
`{14,` *cx*`,` *cxreg*`}`	load complex constant *cx* into complex register *cxreg*

Loading registers with constants.

`{15,` $boolreg_1$`,` $boolreg_2$`}`	copy $boolreg_1$ into $boolreg_2$
`{16,` $intreg_1$`,` $intreg_2$`}`	copy $intreg_1$ into $intreg_2$
`{17,` $realreg_1$`,` $realreg_2$`}`	copy $realreg_1$ into $realreg_2$
`{18,` $cxreg_1$`,` $cxreg_2$`}`	copy $cxreg_1$ into $cxreg_2$

Copying registers.

`{19,` *boolreg*`,` *intreg*`}`	convert *boolreg* into an integer (0 or 1) and place in *intreg*
`{20,` *intreg*`,` *realreg*`}`	convert *intreg* into a real number and place in *realreg*
`{21,` $realreg_1$`,` $realreg_2$`,` *cxreg*`}`	put the complex number ($realreg_1$ `+ I` $realreg_2$) into *cxreg*

Converting from one type to another.

`{22,` *expr*`,` *boolreg*`}`	evaluate *expr*`[`arg_1`, … ]` to get a Boolean variable
`{23,` *expr*`,` *intreg*`}`	evaluate *expr*`[`arg_1`, … ]` to get an integer
`{24,` *expr*`,` *realreg*`}`	evaluate *expr*`[`arg_1`, … ]` to get a real number
`{25,` *expr*`,` *cxreg*`}`	evaluate *expr*`[`arg_1`, … ]` to get a complex number
`{26,` *expr*`}`	evaluate *expr*`[`arg_1`, … ]` and ignore the result

Evaluating noncompilable expressions.

`{27,` $expr$`,` $type_1$`,` reg_1`, … ,` $boolreg$`}`	
	evaluate $expr$`[`reg_1`, … ]` to get a Boolean variable
`{28,` $expr$`,` $type_1$`,` reg_1`, … ,` $intreg$`}`	
	evaluate $expr$`[`reg_1`, … ]` to get an integer
`{29,` $expr$`,` $type_1$`,` reg_1`, … ,` $realreg$`}`	
	evaluate $expr$`[`reg_1`, … ]` to get a real number
`{30,` $expr$`,` $type_1$`,` reg_1`, … ,` $cxreg$`}`	
	evaluate $expr$`[`reg_1`, … ]` to get a complex number
`{31,` $expr$`,` $type_1$`,` reg_1`, … }`	evaluate $expr$`[`reg_1`, … ]` and ignore the result

Evaluating noncompilable functions after computing their arguments.

`{32,` $intreg_1$`,` $intreg_2$`, … ,` $intreg$`}`	
	add integers
`{33,` $realreg_1$`,` $realreg_2$`, … ,` $realreg$`}`	
	add real numbers
`{34,` $cxreg_1$`,` $cxreg_2$`, … ,` $cxreg$`}`	
	add complex numbers

Addition.

`{35,` $intreg_1$`,` $intreg_2$`, … ,` $intreg$`}`	
	multiply integers
`{36,` $realreg_1$`,` $realreg_2$`, … ,` $realreg$`}`	
	multiply real numbers
`{37,` $cxreg_1$`,` $cxreg_2$`, … ,` $cxreg$`}`	
	multiply complex numbers

Multiplication.

`{38,` $intreg_1$`,` $intreg$`}`	negate an integer
`{39,` $realreg_1$`,` $realreg$`}`	negate a real number
`{40,` $cxreg_1$`,` $cxreg$`}`	negate a complex number

Negation.

`{41,` $realreg_1$`,` $realreg$`}`	reciprocal of a real number
`{42,` $cxreg_1$`,` $cxreg$`}`	reciprocal of a complex number

Reciprocals.

{43, $realreg_1$, $intreg_1$, $realreg$}	real number to an integer power
{44, $realreg_1$, $realreg_2$, $realreg$}	real number to a real power
{45, $cxreg_1$, $cxreg_2$, $cxreg$}	complex number to a complex power
{46, $realreg_1$, $realreg$}	square root of a real number

Powers.

{47, $realreg_1$, $realreg$}	`Exp[`x`]`
{48, $realreg_1$, $realreg$}	`Log[`x`]`
{49, $realreg_1$, $realreg$}	`Sin[`x`]`
{50, $realreg_1$, $realreg$}	`Cos[`x`]`
{51, $realreg_1$, $realreg$}	`Tan[`x`]`
{52, $realreg_1$, $realreg$}	`ArcSin[`x`]`
{53, $realreg_1$, $realreg$}	`ArcCos[`x`]`
{54, $realreg_1$, $realreg$}	`ArcTan[`x`]`

Transcendental functions of real variables.

{55, $cxreg_1$, $cxreg$}	`Exp[`x`]`
{56, $cxreg_1$, $cxreg$}	`Log[`x`]`
{57, $cxreg_1$, $cxreg$}	`Sin[`x`]`
{58, $cxreg_1$, $cxreg$}	`Cos[`x`]`
{59, $cxreg_1$, $cxreg$}	`Tan[`x`]`
{60, $cxreg_1$, $cxreg$}	`ArcSin[`x`]`
{61, $cxreg_1$, $cxreg$}	`ArcCos[`x`]`
{62, $cxreg_1$, $cxreg$}	`ArcTan[`x`]`

Transcendental functions of complex variables.

{63, $realreg_1$, $intreg$}	`Floor[`x`]`
{64, $cxreg_1$, $realreg$}	`Re[`z`]`
{65, $cxreg_1$, $realreg$}	`Im[`z`]`
{66, $cxreg_1$, $cxreg$}	`Conjugate[`z`]`
{67, $cxreg_1$, $realreg$}	`Abs[`z`]`
{68, $intreg_1$, $intreg_2$, $intreg$}	`Mod[`i, j`]`

Miscellaneous functions.

`{69,` $logreg$`,` $offset$`}`	jump if false
`{70,` $offset$`}`	jump always
`{71,` $intreg$`,` int_1`,` $offset_1$`,` $\ldots$ `}`	switch

Branching.

`{72,` $intreg_1$`,` $intreg_2$`,` $logreg$`}`	i `==` j
`{73,` $realreg_1$`,` $realreg_2$`,` $logreg$`}`	x `==` y
`{74,` $intreg_1$`,` $intreg_2$`,` $logreg$`}`	i `<` j
`{75,` $realreg_1$`,` $realreg_2$`,` $logreg$`}`	x `<` y
`{76,` $intreg_1$`,` $intreg_2$`,` $logreg$`}`	i `<=` j
`{77,` $realreg_1$`,` $realreg_2$`,` $logreg$`}`	x `<=` y

Comparisons.

`{78,` $logreg_1$`,` $logreg_2$`,` $logreg$`}`	p `&&` q
`{79,` $logreg_1$`,` $logreg_2$`,` $logreg$`}`	p `||` q
`{80,` $logreg_1$`,` $logreg$`}`	`!`p

Boolean operations.

`{81}`	force an error to occur

Forcing an error.