Reprint from
The *Mathematica* Conference
June, 1992   Boston, MA

921.9.10.1992

The N functions of *Mathematica*

Jerry Keiper

1	Introduction	G.2
2	Numerical Quadrature	G.3
3	SequenceLimit[ ]	G.11
4	Numerical Summation and Multiplication	G.12
5	NLimit[ ] and ND[ ]	G.20
6	FindRoot[ ]	G.21
7	FindMinimum[ ]	G.28

NIntegrate, SequenceLimit, NSum, NProduct, EulerSum, NLimit, ND, FindRoot, and FindMinimum

1 Introduction

There are many mathematical operations that are inherently infinite in nature: limits, infinite sums, Riemann integrals, even derivatives are 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[1].

The main point of the mathematical concept of a limit is rigorously to 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)
```

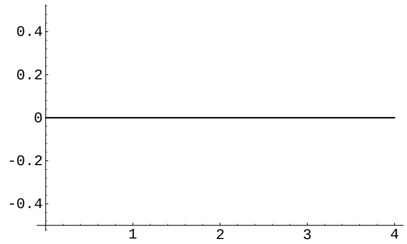
The polynomial `p` is certainly not identically 0.

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

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

[1] 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}]
```

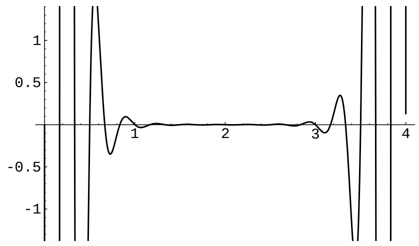


The problem here is similar to what can happen in any algorithm that only samples a few points: based on the sampled points, *Mathematica* incorrectly assumes that the behavior of the function is much simpler than what it actually is.

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}]
```



2 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[2]. Here we explain how *Mathematica* does quadrature and how you can make `NIntegrate[ ]` more effective by an appropriate choice of options.

■ 2.1 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

[2]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 algebra required to evaluate the integral.

2.2 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 a 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.*, stop when the estimated relative error is less than $10^{-(w-10)}$ where w is the `WorkingPrecision`[3].

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.

                                    -18
Out[10]= {0.2 Second, 0. 10    }
```

The `AccuracyGoal` is achieved quite quickly. Note that the result (which is all error) is not much bigger.

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

                                              -17
Out[11]= {0.0166667 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[4]. 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

[3]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.

[4]Quadrature refers to the process of numerically evaluating an integral. The term cubature is used to refer to the process of numerically evaluating a multi-dimensional integral when the multi-dimensional 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 for discontinuous integrands, they will take a *very* long time since they will not try to work only near the discontinuity[5].

`DoubleExponential` quadrature depends on the fact that the integrand is analytic on an open set in the complex plane containing the interval of integration[6]. 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 or is not periodic or the interval of quadrature is not an integral number of periods `Trapezoidal` quadrature will perform rather poorly.

---

[5] 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 2.3 below.

[6] 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]= {5.28333, 10.2333, 19.6833, 32.9167}
```

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]= {1.15, 0.333333, 0.416667, 0.516667}
```

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]= {0.633333, 0.783333, 0.7, 0.816667,
  0.883333, 0.983333}
```

Define a different function depending on SingularityDepth.

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

If there are not 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]= {0.766667, 0.783333, 0.85, 0.416667, 0.45,
  0.4}
```

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 2.3 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.

■ 2.3 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 this continuity is missing convergence will 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]
```

We don't get convergence because of discontinuities in the integrand or its derivatives.

```
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
```

Integrating it as several individual pieces solves the convergence problem.

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

Out[3]= 7.33333
```

2.4 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
```

2.5 Cauchy Principal Values

Cauchy principal values of divergent integrals can be evaluated using the package `CauchyPrincipalValue.m`. 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.

Get 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 1 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
```

2.6 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.

Read in the package.

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

Define some equally spaced data. The implied abscissas are $1, 2, 3, \ldots, 8$.

```
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 several cases. 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 $2 \leq x \leq 3$. 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 $0 \leq x \leq 1$ 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
```

3 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 applied to 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[a + (b0 + b1 n) lambda^n, {n, 8}];
```

The limit of the sequence is `a` *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]= a
```

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[ ]`.

4 Numerical Summation and Multiplication

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. Whatever we say below regarding `NSum[ ]` is also true for `NProduct[ ]`. Whenever this is not true we will explicitly point out the differences.

■ 4.1 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[ ], ]`.

■ 4.2 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

                   -10
Out[2]= -9.867 10
```

```
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 one 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 explicitly to 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 often can 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[7]. 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 option Method and 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 arbi-

[7] 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*.

trary 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

                     -11
Out[1]= -1.21208 10
```

The exact result would be 0.

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

                     -10
Out[2]= -3.18467 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 means to form a sequence of partial sums of the `NSumExtraTerms` terms and pass this sequence to `SequenceLimit[ ]`. The option `NSumExtraTerms` refers to the number of terms that will 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[ ]`.

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
```

■ 4.3 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 the accuracy of the result will be reduced 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`[8], 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
```

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
```

By requesting higher precision we get it provided the other options have values that make the `PrecisionGoal` possible to achieve.

```
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}];
```

[8] 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.

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

              -33
Out[5]= 0. 10

In[6]:= Accuracy[%]

Out[6]= 32
```

■ 4.4 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]
```

Finf 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]= 2.01294210825481695463443794395887350933148\
  37540963666326161
```

4.5 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

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

The error is simply unacceptable.

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

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

                    11
Out[1]= -3.77392 10
```

The error is 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 gained with a little thought is

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

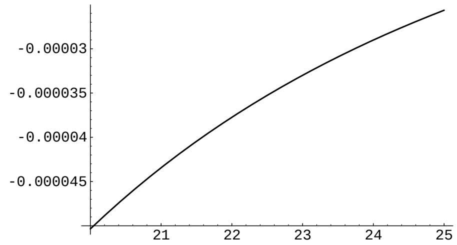
By rewriting this as

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

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.

This summand is quite well behaved.

```
In[1]:= Plot[Cos[(n - Sqrt[n(n-1)]) Pi]/n^2, {n, 20, 25}]
```

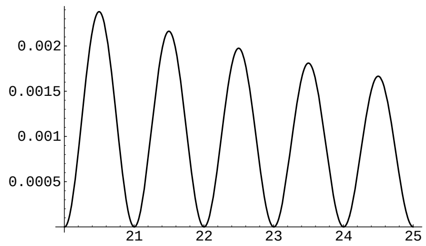


and the error is small.

```
In[2]:= NSum[Cos[(n - Sqrt[n(n-1)]) Pi]/n^2, {n, Infinity},
                Method -> NIntegrate,
                VerifyConvergence -> False] +
                1.1015103420583935729

                    -12
Out[2]= 8.25096 10
```

This is quite nasty. Notice that the imaginary part is positive. This would result in the imaginary part of the integral being nonzero and the error being quite large.

```
In[3]:= Plot[Im[Cos[Sqrt[n(n-1)] Pi] (-1)^n/n^2], {n, 20, 25}]
```



■ 4.6 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.m`.

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 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 $\{\lambda_i, deg_i + 1\}$. 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

Read in 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
```

5 NLimit[ ] and ND[ ]

The package `NumericalMath/NLimit.m` 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[ ]`. 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.

■ 5.1 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 is massive loss of precision when the function values are differenced.

5.2 Examples

Read in the package NLimit.m.

```
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
```

6 FindRoot[ ]

FindRoot[ ] is used numerically to 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.

6.1 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.

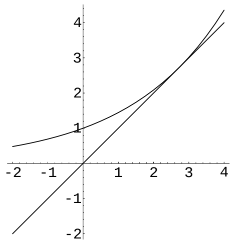
```
In[4]:= jac = Outer[D, func, {x, b}]

                    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]
```

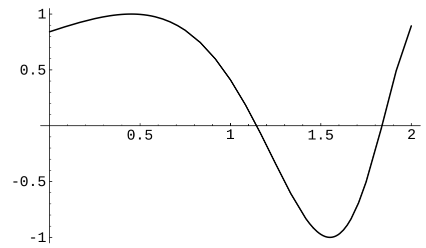


Brent's Method

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.

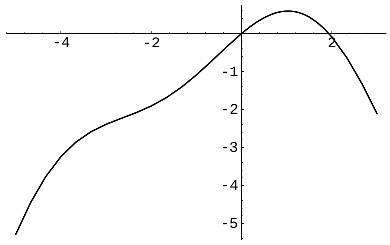
```
In[1]:= FindRoot[Evaluate[{b^x == x,
                           D[b^x, x] == 1}],
                           {x, 1, 2}, {b, 1, 2}]

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}
```

Starting at $x = -2$ and $x = -1$, find a root, stopping the search if it ever 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   }
```

Starting at $x = -3$ and $x = -2$, find a root, stopping the search when it 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.

■ 6.2 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 6.1). 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.

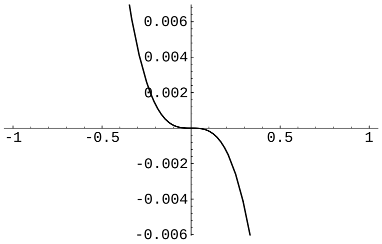
The form of Newton's method that `FindRoot[ ]` uses is

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

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}

LinearSolve::sing: Matrix {{0.}} is singular.

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

With the default `DampingFactor` this 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}
```

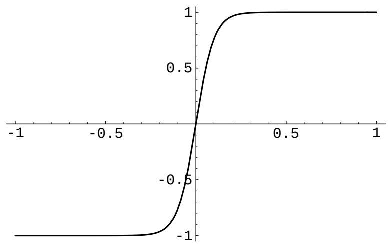
It converges quite nicely with the proper `DampingFactor`.

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

x
2.
0.
0.

Power::infy:
                                -(2/3)
   Infinite expression 0.             encountered.

FindRoot::fjnum:
   Jacobian {{ComplexInfinity}} is not a 1 by 1
     matrix of numbers at {x} = {0.}. Try giving two
     starting values for each variable.

                                 1/3
Out[5]= FindRoot[Print[x]; x    , {x, 2},

                        1
  DampingFactor -> -]
                        3
```

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]
```

Out[7]= $\{\frac{1}{64}, \frac{1}{32}, \frac{1}{16}, \frac{1}{8}, \frac{1}{4}, \frac{1}{2}, 1\}$

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 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 then 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 terminated.

■ 6.3 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 a local minimum that is not zero, so it is far from foolproof.

7 FindMinimum[ ]

`FindMinimum[ ]` is used to find numerically the minimum[9] 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.

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.

■ 7.1 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

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

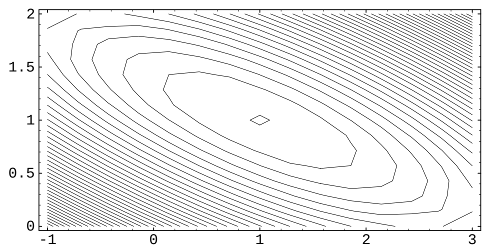
be used to specify the gradient of the objective function. If you leave it set to the default value of `Automatic` *Mathematica* will attempt to find the gradient symbolically from the expression for the objective function. The reasons you might want to specify the gradient yourself are the same reasons you might want to specify the Jacobian yourself when using Newton's method in `FindRoot[ ]` (See Section 6.1 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]= {1.58758 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]= {1.58758 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.

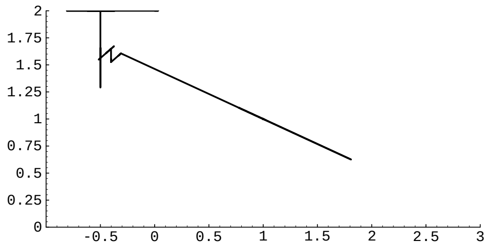
```
In[7]:= searchpath = {}

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}]

                      -27
Out[8]= {6.48478 10    , {x -> 1., y -> 1.}}
```

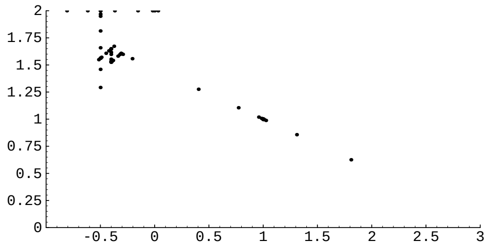
Plot the search path. Note that once it gets the direction chosen correctly it gets to the solution quite quickly. Of course most objective functions are not simple quadratic functions and the correct direction will never be found exactly, only in the limit as the search gets close to the solution.

```
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}]

                       -27
Out[11]= {4.76731 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.

■ 7.2 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.