N Function of MATHEMATICA
and Some Notices to Numerical Quadrature

Monika Kováčová
Dept. of Mathematics, Fac. of Mechanical Engineering,
Slovak Technical University, Bratislava
kovacova_v@dekan.sjf.stuba.sk

Abstract: *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. 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.

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 proof.

In fact such is not the case; each of the *MATHEMATICA* functions we will discuss here can give incorrect answers. It is also the case that by choosing the options to these functions appropriately you can always get the correct answer.

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. 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. In our paper we explain how *MATHEMATICA* does quadrature and how you can make **`NIntegrate[]`** more effective by an appropriate choice of options.

3. Integrate[] or Integrate[]//N (N[Integrate[]])

There are several ways to define integration in ***MATHEMATICA***:

```
Integrate[],
N[Integrate[], ],
NIntegrate[],
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 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 those points. If **`Integrate[]`** is unable to find the vale 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 sholud use **`NIntegrate[]`** directly rather than through **`N[Integrate[], ]`**.

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 paper can help you better understand how to use **`NIntegrate[]`**.

4. WorkingPrecision, AccuracyGoal, and PrecisionGoal

The there options WorkingPrecision, AccuracyGoal, and PrecisionGoal are used for several other numerical operations in addition to **`NIntegrate[]`**. All three deal with in some 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 *WorkingPresicion* 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 in as much as you can find a well-conditioned algorithm to evaluate it), but you will get a significant improvement in speed. The default for *WorkinPrecision* is to use machine precision. Using machine precision you get a machine precision answer.

In[1]:=

```
NIntegrate [Sin[x],{x,0,Pi}]-2
```

Out[1]=

```
               -16
4.44089 10
```

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

In[2]:=

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

Out[2]=

```
2.0000000000000000000
```

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

In[3]:=

```
Precision[y]
```

Out[3]=

```
20
```

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

In[4]:=

```
SetPrecision[y,40]-2
```

Out[4]=

```
         -39
-3. 10
```

AccuracyGoal and *PrecisonGoal* are used 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*. There is exception to this. Different machines have different precision for their machine arithmetic. To have some consistency between machines, when machine arithmetic is being used, the values of Automatic for *PrecisionGoal* means that the precision goal is to be six digits.

5. Compiled

The option Compiled is also common to many numerical functions in ***MATHEMATICA***. It can take only two values: True, which is 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.

6. Method

Currently there are four different methods that can be used for quadrature. Quadrature refers to the process of numerically evaluating an integral. The term cubature is used refer to the process of numerically evaluating a multi-dimensional integral when the multidimensional aspects are being emphasized.

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 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 *GaussKnonrod* for one dimensional quadrature and *MultiDimensional* for two or more dimensions. The methods *GaussKronrod, DoubleExponential* and *Trapezoidal* can also be used for cubature, but 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 type of singularity.

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

DoubleExponential quadrature converges as $exp(-cN/log N)$, where N is the number of the function evaluations. This means that for large N, the accuarcy 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 it 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 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}.

Trapeziodal quadrature can used to examine the trapezoidal method where the step size 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.

7. 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 GaussKnonrod quadrature there is only one dimension from which to choose. With both GaussKnonrod 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. 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 *MinRecusrion*.

With DoubleExponential and Trapezoidal quadrature, no recursive bisection of the interval of integration occurs, so *MinRecusrion* and *MaxRecursion* have a different meaning. Recall that with these methods the trapezoid rule is used and the step size 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.

8. Singularity Depth

Both GaussKronrod and Multidimensional quadrature are adaptive, that is bisection in 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 occured. To deal with these situations there is 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 asssumed singularity easier to integrate. Note that the singularity in only assumed to exist. If *SingularityDepth* is set too small the reparametrization will often occur when there is no singularity. Since the change of variable is expensive we want to avoid unnecessary reparametrizations. If *SingularityDepth* is set too large, too much time be spent trying to achieve convergence via recursive bisection while a change of variable would allow convergence much more quickly.

For example we define a function depending on *SingularityDepth*. Note the integrable singularities in the integrand at the endpoint of the interval of integration.

In[5]:=

```
g[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[6]:=

```
Table[First[Timing[g[n]]]/Second,{n,6}]
```

Out[6]=

```
{1.26, 0.55, 0.28, 0.27, 0.28, 0.33}
```

An another example:

In[7]:=

```
h[n_]:=NIntegrate[Sin[x],{x,0,13Pi},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[8]:=

```
Table[First[Timing[h[n]]]/Second,{n,10}]
```

Out[8]=

```
{1.87, 0.33, 0.27, 0.17, 0.16, 0.17, 0.16, 0.11, 0.17, 0.16}
```

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 result will be added together.

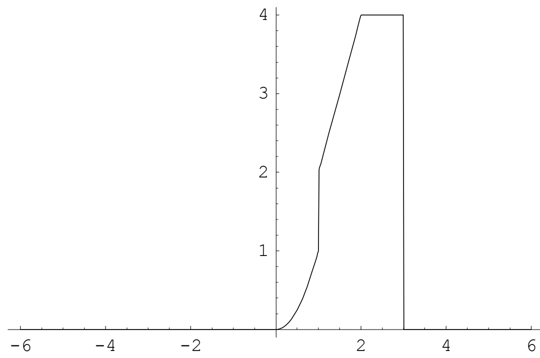
SingularityDepth is not used with either DoubleExponential or Trapezoidal quadrature. This is because the reparametrization inherent in DoubleExponential quadrature is already able to deal with integrable singularities at endpoints of the interval of integration and *Trapezoidal quadrature* is designed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

9. 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 **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. Some short example:

In[9]:=

```
f[x_]:= Which[x<0,0, x<1,x^2,x<2,2x,x<3,4,x>=3,0]
Plot[f[x],{x,-6,6}]
```



Out[10]=

```
-Graphics-
```

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

In[11]:=

```
NIntegrate[f[x],{x,-6,6}]
```

```
NIntegrate::slwcon:
   Numerical integration converging too slowly; suspect one of
the
    following: singularity, value of the integration being 0,
    oscillatory integrand, or insufficient WorkingPrecision. If
    your integrand is oscillatory try using the option
    Method->Oscillatory in NIntegrate.

NIntegrate::ncvb:
   NIntegrate failed to converge to prescribed accuracy after 7
     recursive bisections in x near x = 0.984375.
```

Out[11]=

```
7.32966
```

Integrating it as several individual pieces solves the problem of the convergence.

In[12]:=

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

Out[12]=

```
7.33333
```

10. `SequenceLimit[]` and Oscillatory Integrands

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

{3+1/2, 3+1/4, 3+1/8, 3+1/16, 3+1/32, 3+1/64}

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[]** used Wynn's algorithm and in order to get the correct value for the limit the option *WynnDegree* is the number of degrees of freedom in the model for the data. The algorithm for **SequenceLimit[]** works by transforming a sequence of length *n* into a sequence of length *n-2*. This transforming 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 sequence is too short to continue. This 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.

For example we define a sequence with total degree 3 and length 8

In[13]:=

```
post = 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[14]:=

```
SequenceLimit[post,WynnDegree->3]
```

Out[14]=

```
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.

Put post2 to a sequence of numbers known to convergence to *Pi/4*

In[15]:=

```
post2= FoldList[Plus,0,Table[(-1)^k/(2k+1),{k,0,16}]]//N
```

Out[15]=

```
{0, 1., 0.666667, 0.866667, 0.72381, 0.834921, 0.744012,

 0.820935, 0.754268, 0.813091, 0.76046, 0.808079, 0.764601,

 0.804601, 0.767564, 0.802046, 0.769788, 0.800091}
```

Use the default value for the option *WynnDegree* goves a prettz good approximation to the limit.

In[16]:=

```
SequenceLimit[N[post2]]- N[Pi/4]
```

Out[16]=

```
              -13
1.83852 10
```

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

In[17]:=

```
SequenceLimit [N[post2] , WynnDegree->Infinity] - N[Pi/4]
```

Out[17]=

```
               -13
-7.25864 10
```

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 zeroes 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 succesive zeroes 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

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

Let define a function that gives the integral between successive zeros.

In[18]:=

```
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 and form the sequence of the partial sum

In[19]:=

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

Out[19]=

```
{0.434001, -0.362003, 0.240436, -0.176644, 0.138989, -0.114386,

 0.0971148, -0.084344, 0.0745262, -0.0667473, 0.0604339}
```

In[20]:=

```
b= FoldList[Plus,0,a]
```

And now we find numerical limit of the sequence

In[21]:=

```
c1=SequenceLimit[b]
```

Out[21]=

```
0.212584
```

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

In[22]:=

```
c1-N[Pi/(2 E ^ 2)]
```

Out[22]=

```
            -8
4.79579 10
```

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

In[23]:=

```
c2=NSum[f[n],{n,0, Infinity}, Method->SequenceLimit,
VerifyConvergence->False]
```

Out[23]=

```
0.212584
```

In[14]:=

```
c2 - N[Pi/(2 E ^ 2)]
```

Out[24]=

```
            -15
6.77236 10
```

The error is quite small.

References:

[1] Halada L.: *Stabilita úloh a algoritmov vo výučbe numerickej matematiky na SJF STU,* Proceedings of the scientific conference with international participation, INFORMATICS AND ALGHORITHMS '98, Prešov 3. - 4. sept. 1998, pp.147-151

[2] Halada L.: *Použitie systému Dotest pri výčbe a skúšaní matematiky,* ***MATHEMATICA* 99**, Bratislava 29.6.-2.7. 1999

[3] Kolesárová A.: *Výučba Fourierových radov s podporou systému MATHEMATICA,* ***MATHEMATICA* 99**, Bratislava 29.6.-2.7. 1999

[4] Kováčová M., Halada L. : *Experimentálna výučba numerickej matematiky pomocou programového systému MATHEMATICA na Sjf STU,* Matematická štatistika a Numerická matematika, Kálnica 1. -5. júna 1998, pp.142-150

[5] Omachelová M.: *Zisťovanie priebehu funkcie 1 reálnej premennej s podporou pg. systému MATHEMATICA ,* ***MATHEMATICA* 99**, Bratislava 29.6.-2.7. 1999

[6] Záhonová V.: *Lineárne diferenciálne rovnice n-tého rádu s konštantnými koeficientami a program. systém MATHEMATICA,* ***MATHEMATICA* 99**, Bratislava 29.6.-2.7. 1999, pp.

[7] Záhonová, V.: *Výučba integrálneho počtu funkcie jednej reálnej premennej s podporou programového systému MATHEMATICA*, In.:25 VŠTEP-Z Matematika v inžinierskom vzdelávaní, Trnava, 7. - 10. September 1998, pp. 192 - 197

[8] Wolfram Research: *The MATHEMATICA Book 3rd ed.*, Wolfram Media/Cambridge University Press, 1996.