[6.26] Accurate numerical derivatives with nDeriv() and Ridder's method

This tip is lengthy. It is divided into these sections:

- *Optimum results from nDeriv.* How to use the built-in *nDeriv()* function to get the most accurate results, with a function called *nder2().*
- *More accurate results with Ridders' method.* Shows a method (Ridders') that gives much better results than *nDeriv(),* with a program called *nder1().*
- *Diagnosing nder1() with nder1p().* Shows how to fix errors that might occur with the Ridders' method program.
- *General comments on numerical differentiation.* General concerns with any numerical differentiation method.
- *More comments on nder1() and Ridders' method*. Lengthy discussion of *nder1()* results, performance, and also some interesting behavior of the built-in *nDeriv()* function.

In general, the most accurate method to find a numerical derivative is to use the CAS to find the symbolic derivative, then evaluate the derivative at the point of interest. For example, to find the numeric derivative of tan(x), where x = π /2.01 = 1.562981..., find

$$\frac{d}{dx}\tan(x) = \frac{1}{(\cos(x))^2} = 16374.241898666$$

This method fails when the CAS cannot find a symbolic expression for the derivative, for example, for a complicated user function. The 89/92+ provide two built-in functions for finding numerical derivatives: *avgRC()* and *nDeriv(). avgRC()* uses the forward difference to approximate the derivative at *x*:

$$\text{avgRC}(f(x), x, h) = \frac{f(x+h)-f(x)}{h}$$

and *nDeriv()* uses the central difference to approximate the derivative:

$$\text{nDeriv}(f(x), x, h) = \frac{f(x+h)-f(x-h)}{2h}$$

The built-in functions *avgRC()* and *nDeriv()* are fast, but they may not be accurate enough for some applications. The *avgRC()* function can return, at best, an accuracy on the order of $e_m^{1/2}$, where e_m is the machine accuracy. Since the 89/92+ uses 14 decimal digits, e_m = 1E-14, so the best result we can expect is about 1E-7. For *nDeriv(),* the best accuracy will be about $e_m^{2/3}$, or about 5E-10. Neither of these accuracies reach the 12-digit accuracy of which the 89/92+ is capable. It is also quite possible that the actual error will be much worse. Our goal is to develop a routine which can calculate derivatives with an accuracy near the full displayed resolution of the 89/92+.

Since the title of this tip is *accurate* numeric derivatives, I won't further consider *avgRC().* Instead, I will show how to get the best accuracy from *nDeriv(),* and also present code for an alternative method that does even better.

Optimum results from nDeriv(): nder2()

nDeriv() finds the central difference over an interval *h*, where *h* defaults to 0.001. Since the limit of the *nDeriv()* formula *is* the derivative, it makes sense that the smaller we make *h*, the better the derivative result. Unfortunately, round-off errors dominate the result for too-small values of *h*. Therefore, there is an optimum value of *h* for a given function and evaluation point. An obvious strategy is to evaluate the

formula with increasingly smaller values of *h*, until the absolute value of successive differences in f'(x) begins to increase. This program, *nder2(),* shows this idea:

```
nder2(ff,xx)
func
©("f",x) find best f'(x) at x with central-difference formula
©6jun00 dburkett@infinet.com

local ff1,ff2,k,x,d1,d2,d3,h

©Build function expressions
ff&"(xx+h)"→ff1
ff&"(xx-h)"→ff2

©Find first two estimates
.01→h
(expr(ff1)-expr(ff2))/.02→d1
.001→h
(expr(ff1)-expr(ff2))/.002→d2

©Loop to find best estimate
for k,4,14
 10^-k→h
 (expr(ff1)-expr(ff2))/(2*h)→d3
 if abs(d3-d2)>abs(d2-d1) then
  return d2
 else
  d2→d1
  d3→d2
 endif
endfor

©Best not found; return last estimate
return d3

Endfunc
```

Call *nder2()* with the function name as a string. For example, to find the derivative of tan(x) at π/2.01, the call is

nder2("tan",π/2.01)

which returns 16374.242. The absolute error is about 1.01E-4, and the relative error is 6.19E-9.

The table below demonstrates the improvement in the derivative estimate as *h* decreases, for tan(x) evaluated at x = 1.

h	f'(x)	difference in f'(x)
1E-2	3.4264 6416 009	(none)
1E-3	3.4255 2827 135	9.359E-4
1E-4	3.4255 1891 5	9.356E-6
1E-5	3.4255 1882	9.5E-8
1E-6	3.4255 188	2E-8
1E-7	3.4255 185	3E-7

nder2() starts with h = 0.01, and divides h by 10 for each new estimate, so that the steps for h are 1E-2, 1E-3, ... 1E-14. Since the difference in f'(x) starts increasing at h = 1E-7, *nder2()* returns the value for h = 1E-6.

More accurate results with Ridders' method: nder1()

Ridders' method *(Advances in Engineering Software*, vol. 4, no. 2, 1982*)* is based on extrapolating the central difference formula to h=0. This method also has the important advantage that it returns an estimate of the error, as well. This program, *nder2(),* implements the algorithm:

```
nder1(ff,xx,hh)
func
©("f",x,"auto" or h), return {f'(x),err}
©Based on Ridders' algorithm
©6jun00/dburkett@infinet.com

local con,con2,big,ntab,safe,i,j,err,errt,fac,amat,dest,fphh,fmhh,ffun,h1,d3

© Initialize constants
1.4→con
con*con→con2
1E900→big
10→ntab
2→safe
newmat(ntab,ntab)→amat

© Build function strings
ff&"(xx+hh)"→fphh
ff&"(xx-hh)"→fmhh
ff&"(xx)"→ffun

© Find starting hh if hh="auto"
if hh="auto" then
 if xx=0 then: .01→h1
 else: xx/1000→h1
 endif
 expr(ff&"(xx+h1)")-2*expr(ffun)+expr(ff&"(xx-h1)")→d3

 if d3=0: .01→d3
 (√(abs(expr(ffun)/(d3/(h1^2)))))/10→hh
 if hh=0: .01→hh

endif

©Initialize for solution loop
(expr(fphh)-expr(fmhh))/(2*hh)→amat[1,1]
big→err

©Loop to estimate derivative
for i,2,ntab
 hh/con→hh
 (expr(fphh)-expr(fmhh))/(2*hh)→amat[1,i]

 con2→fac

 for j,2,i
  (amat[j-1,i]*fac-amat[j-1,i-1])/(fac-1)→amat[j,i]
  con2*fac→fac
  max(abs(amat[j,i]-amat[j-1,i]),abs(amat[j,i]-amat[j-1,i-1]))→errt
  if errt≤err then
   errt→err
   amat[j,i]→dest
  endif

 endfor
 if abs(amat[i,i]-amat[i-1,i-1])≥safe*err:exit
```

```
endfor

return {dest,err}

endfunc
```

nder1() is called with the function name as a string, the evaluation point, and the initial step size. If the step size is "auto", then *nder1()* tries to find a good step size based on the function and evaluation point. *nder1()* returns a list with two elements. The first element is the derivative estimate, and the second element is the error estimate.

nder1() is called as

```
nder1(fname,x,h)
```

where *fname* is the name of the function as a string in double quotes. *x* is the point at which to find the derivative. *h* is a parameter which specifies a starting interval. If *h* is "auto" instead of a number, then *nder1()* will try to automatically select a good value for *h*.

For example, to find the derivative of tan(x) at x = 1.0, with an automatic step size, use

```
nder1("tan",1,"auto")
```

which returns {3.42551882077, 3.7E-11}. The derivative estimate is 3.4255..., and the error estimate is 3.7E-11. To find the same derivative with a manual step size of 0.1, use

```
nder1("tan",1,.1)
```

which returns {3.42551882081,1.4E-12}.

The value *h* is an initial step size. It should not be too small, in fact, it should be an interval over which the function changes substantially. I discuss this more in the section below, *More comments on nder1() and Ridders' method.*

To use *nder1()* and return just the derivative estimate, use

```
nder1(f,x,h)[1]
```

where *[1]* specifies the first element of the returned list.

If *nder1()* returns a very large error, then the starting interval *h* is probably the wrong value. For example,

`nder1("tan",1.5707,"auto")` returns {-1.038873452988 E7, 3.42735731968 E8}

Note that the error is quite large, on the order of 3.43E8. We can manually set the starting interval to see if we can get a better estimate. First, using *nder1p()* (see below), we find that the starting interval with the "auto" setting is about 1.108E-4. If we try *nder1()* with a starting interval of about 1/10 of this value, we get

```
nder1("tan",1.5707,1E-5) = {1.07771965667E8, 1.62341}
```

Since the error estimate is much smaller, we can trust the derivative estimate.

Execution time will depend on the execution time of the function for which the derivative is being found. For simple functions, execution times of 5-20 seconds are not uncommon.

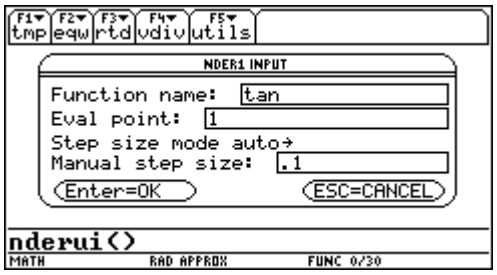
It can be convenient to have a user interface for *nder1().* This program provides an interface:

```
nderui()
Prgm
© User interface for nder1()
© 3jan00/dburkett@infinet.com
© Result also saved in nderout
local fn1,xx,steps,smode,reslist,ssz

1→xx
.1→steps

lbl lp

string(xx)→xx
string(steps)→steps
dialog
title "NDER1 INPUT"
request "Function name",fn1
request "Eval point",xx
dropdown "Step size mode",{"auto","manual"},smode
request "Manual step size",steps
enddlog
if ok=0:return

expr(xx)→xx
expr(steps)→steps

when(smode=2,steps,"auto")→ssz

nder\nder1(fn1,xx,ssz)→reslist

reslist[1]→nderout

dialog
 title "NDER1 RESULTS"
 text "Input x: "&string(xx)
 text "dy/dx: "&string(reslist[1])
 text "Error est: "&string(reslist[2])
 text "(Derivative → nderout)"
enddlog
if ok=0:return
goto lp

EndPrgm
```

To use *nderui(), nder1()* must be saved in a folder called *nder*. *nderui()* should be stored in the same folder. Run *nderui()* like this:

```
nder/nderui()
```

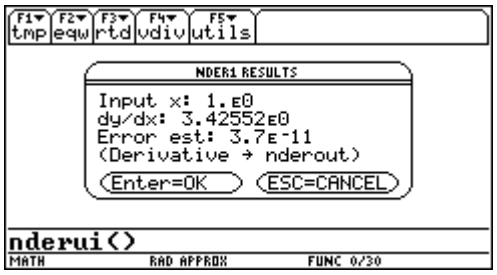
and this input dialog box is shown:



The Function Name is entered without quotes, as shown. *Eval point* is the point at which the derivative is found. *Step size mode* is a drop-down menu with these choices:

1: auto *nder1() finds the interval size*
2: manual *you specify the interval in Manual step size*

When all the input boxes are complete, press [ENTER] to find the derivative. This results screen is shown:



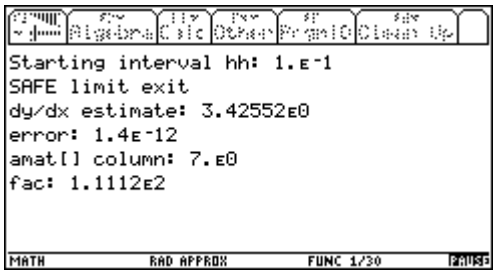
Input x is the point at which the derivative is estimated. *dy/dx* is the derivative estimate. *Error est* is the error estimate. The derivative estimate is saved in the global variable *nderout* in the current folder. Push [ENTER] to find another derivative, or press [ESC] to exit the program.

Diagnosing nder1() with nder1p()

nder1p() uses the same algorithm as *nder1(),* but is coded as a program instead of a function. This version can be used to debug situations in which *nder1()* returns results with unacceptably high errors.

nder1p() is called in the same way as *nder1():* `nder1p(f,x,h)`.

When *nder1p()* finishes, the results are shown on the program I/O screen. Push ENTER to return to the home screen. The results screen looks like this:



The 'Starting interval hh' shows the first value for *h*, which is especially helpful when the "auto" option is used. The string "SAFE limit exit" may or may not be shown, depending on how *nder1p()* terminates. The 'amat[] column' shows the number of main loops that executed before exit. 'fac' is a scaling factor.

nder1p() creates these global variables, which can all be deleted:

con, con2, big, ntab, safe, i, j, err, errt, fac, amat, dest, fphh, fmhh, ffun ,h1, d3

This is the code for *nder1p():*

```
nder1p(ff,xx,hh)
prgm
©("f",x,h) return f'(x), h is step size or "auto"
©6jun00/dburkett@infinet.com
©program version of nder1()

1.4→con
con*con→con2
1E900→big
10→ntab
2→safe

ff&"(xx+hh)"→fphh
ff&"(xx-hh)"→fmhh
ff&"(xx)"→ffun

if hh="auto" then
 if xx=0 then: .01→h1
 else: xx/1000→h1
 endif
 expr(ff&"(xx+h1)")-2*expr(ffun)+expr(ff&"(xx-h1)")→d3

 if d3=0: .01→d3

 (√(abs(expr(ffun)/(d3/(h1^2)))))/10→hh

 if hh=0: .01→hh

endif

clrio
disp "Starting interval hh: "&string(hh)

newmat(ntab,ntab)→amat
(expr(fphh)-expr(fmhh))/(2*hh)→amat[1,1]
big→err

for i,2,ntab
 hh/con→hh
 (expr(fphh)-expr(fmhh))/(2*hh)→amat[1,i]
```

```
 con2→fac

 for j,2,i
  (amat[j-1,i]*fac-amat[j-1,i-1])/(fac-1)→amat[j,i]
  con2*fac→fac
  max(abs(amat[j,i]-amat[j-1,i]),abs(amat[j,i]-amat[j-1,i-1]))→errt
  if errt≤err then
   errt→err
   amat[j,i]→dest
  endif

 endfor
 if abs(amat[i,i]-amat[i-1,i-1])≥safe*err then
 disp "SAFE limit exit"
 exit
 endif

endfor

disp "dy/dx estimate: "&string(dest)
disp "error: "&string(err)
disp "amat[] column: "&string(i)
disp "fac: "&string(fac)
pause
disphome

endprgm
```

General comments on numerical differentiation

Any numerical derivative routine is going to suffer accuracy problems where the function is changing rapidly. This includes asymptotes. For example, suppose that we try to find the derivative of tan(1.5707). This is very close to the asymptote of pi/2 = 1.57079632679. *nder2()* returns an answer of -1.009...E6, which is clearly wrong. The problem is that the starting interval brackets the asypmtote. *nder1()* returns an answer of -1.038..E7, but at least also returns an error of 3.427E8, so we know something is wrong.

The function must be continuous on the sample interval for both *nder1()* and *nder2().*

There are other methods for finding numeric derivatives. For example, you can fit a polynomial (either Lagrange or least-squares) through some sample points, then find the derivative of the polynomial. These methods might not have any speed advantage over *nder2(),* because an accurate, high-order polynomial requires considerable time to find the coefficients, although it is fast to find the derivative once you have them.

In my library of numerical methods books I find little reference to finding numerical derivatives. This is perhaps because it is relatively easy to find derivatives of most simple functions.

Numerical Recipes in Fortran, 2nd edition, William H. Press et al. Section 5.7, p181 describes various issues and problems in calculating numerical derivatives. The expressions for the errors of *avgRC()* and *nDeriv()* are also found here.

More comments on nder1() and Ridders' method

nder1() is a 'last resort' for finding numerical derivatives, to be used when the other alternatives have failed. The other alternatives are 1) finding a symbolic derivative, and 2) using the 89/92+ built-in numerical derivative functions *avgRC()* and *nDeriv().*

For example, there is no point in using *nder1()* for a simple function such as tan(x). The 89/92+ can easily find the symbolic derivative, which can then be evaluated numerically. However, you may have complex, programmed functions for which 89/92+ cannot find a symbolic derivative.

The error will increase dramatically near function asymptotes, where the function is changing very rapidly.

It is possible to adjust *h* in *nDeriv()* to get reduce the error at a given point, but that is not very valuable if you don't know what the answer should be! However, it is possible with a program like *nDer1(),* which returns an error estimate, to improve the answer by adjusting *h*. For example, suppose we want to find the derivative of tan(x) as accurately as possible at x = 1.4. The basic idea is to call *nder()* with different values of *h*, and try to locate the value of *h* with the smallest error. The table below shows some results.

Interval size *h*	**Reported error**	**Actual error**
0.13	1.77E-9	1.06E-9
0.10	7.54E-10	5.78E-10
0.08	1.03E-9	2.57E-10
0.06	4.58E-10	4.28E-10
0.04	4.06E-10	2.71E-10
0.02	3.83E-10	1.44E-9
0.01	9.06E-10	6.66E-10
0.005	3.05E-9	6.97E-9

This example shows that the error is not too sensitive to the interval size, and the actual error is usually less than the reported error bound.

nder1() uses Ridders' algorithm to estimate the derivative. The general idea is to extrapolate the central-difference equation to h=0:

$$f'(x) = \frac{f(x+h)-f(x-h)}{2 \cdot h}$$

If we could let h=0, then this equation would return the exact derivative. However, h=0 is obviously not allowed. Further, we can't just make *h* arbitrarily small, because the loss of precision degrades the answer. This can be seen by using *nDeriv()* with various values of h. The table below shows the results returned by *nDeriv()* for f(x) = tan(x), with various values of h, and with x = 1

h	**f'(x)**	**error**
1E-01	3.5230 0719 849	-9.749E-02
1E-02	3.4264 6416 008	-9.453E-04
1E-03	3.4255 2827 133	-9.451E-06
1E-04	3.4255 1891 538	-9.456E-08
1E-05	3.4255 1882 37	-2.886E-09
1E-06	3.4255 1880 37	1.712E-08
1E-07	3.4255 1882 081	-1.000E-13
1E-08	3.4255 1882 081	-1.000E-13
1E-09	3.4255 1882 081	-1.000E-13
1E-10	3.4255 1882 081	-1.000E-13
1E-11	3.4272 3158 023	-1.000E-13
1E-13	3.5967 9476 186	-1.713E-13
1E-14	1.7127 5941 041	1.712E+00

This seems to be very good performance - too good, in fact. Suppose that instead of including the tan(x) function in *nDeriv(),* we call the function indirectly, like this:

```
nDeriv(ftan(xx),xx,h)|xx=1
```

where *ftan()* is just a user function defined to return tan(x). This results in the following:

h	f'(x)	error
1E-01	3.5230071984915	-9.75E-02
1E-02	3.426464160085	-9.45E-04
1E-03	3.42552827135	-9.45E-06
1E-04	3.425518915	-9.42E-08
1E-05	3.42551882	8.15E-08
1E-06	3.4255188	2.08E-08
1E-07	3.4255185	3.21E-07
1E-08	3.42552	-1.18E-06
1E-09	3.4255	-1.88E-05
1E-10	3.4255	-1.88E-05
1E-11	3.425	5.19E-04
1E-12	3.4	2.55E-02
1E-13	3.5	-7.45E-02
1E-14	0	3.43E+00

This is more typical of the performance we would expect from the central difference formula. As *h* decreases, the number of digits in the results decreases, because of the increasingly limited resolution of f(x+h) - f(x-h). The accuracy gets better until h = 1E-6, then starts getting worse. At the best error, we only have 8 significant digits in the result.

This example seems to imply that the 89/92+ does not directly calculate the central difference formula to estimate the derivative of tan(x). Instead, the 89/92+ uses trigonometric identities to convert

$$\frac{\tan(x+h)-\tan(x-h)}{2\cdot h}$$

to

$$\frac{\sin(x+h)(\cos(x-h)-\cos(x+h)\sin(x-h))}{2\cdot h\cdot\cos(x+h)\cos(x-h)}$$

This is a clever trick, because it converts the tangent function, which has asymptotes, into a function of sines and cosines, which don't have asymptotes. Any numerical differentiator is going to have trouble wherever the function is changing rapidly, such as near asymptotes.

nDeriv() also transforms some other built-in functions:

sinh(), cosh(), tanh()	uses the exponential definitions of the hyperbolic functions
log()	converts expression to natural log
e^x	converts expression to use e^x form of sinh()
10^x	converts expression to use sinh()
$\tanh^{-1}()$	converts expression to use ln()

but *nDeriv()* directly evaluates these functions:

$\ln(), \sin^{-1}(), \cos^{-1}(), \tan^{-1}(), \sinh^{-1}(), \cosh^{-1}()$

nDeriv() also simplifies polynomials before calculating the central difference formula. For example, *nDeriv()* converts

$3x^2 + 2x + 1$

to

$6(x + 1/3) = 6x + 2$

Notice that in this case *h* drops out completely, because the central difference formula calculates derivatives of 2nd-order equations exactly.

While this is a laudable approach in terms of giving an accurate result, it is misleading if you expect *nDeriv()* to really return the actual result of the central difference formula. Further, you can't take advantage of this method with your own functions if those functions are not differentiable by the 89/92+. This example establishes the need for an improved numerical differentiation routine.

In Ridders' method the general principle is to extrapolate for h=0. *nder1()* implements this idea by using successively smaller values of the starting interval *h*. At each value of *h*, a new central difference estimate is calculated. This new estimate, along with the previous estimates, is used to find higher order estimates. In general, the error will get better as the starting value of *h* is increased, then suddenly get very large. The table below shows this effect using *nder1()* for f(x) = tan(x), where x=1.

hh	error
0.001	3.74E-10
0.005	1.48E-10
0.01	2.22E-11
0.1	4.70E-12
0.15	2.00E-12
0.2	1.64E-11
0.3	-1.10E-12
0.4	-4.60E-12
0.5	4.93E-01

Note that the error suddenly increases at hh = 0.5. Obviously, there is a best value for *hh* that reduces the error, but this best value is not too sensitive to the actual value of hh, as the error is on the order of E-12 from *hh* = 0.1 to 0.4.

One way to find the best *hh* would be to try different values of *hh* and see which returned the smallest error estimate. Since *nder1()* is so slow, I wanted a better method. In the reference below, the authors suggest that a value given by

$$h = \left[\frac{f(x)}{f''(x)} \right]^{\frac{1}{2}}$$

minimizes the error. However, note that this expression includes the second derivative of the function, f''(x). While we don't know this (we don't even know the first derivative!), we can estimate it with an expansion of the central difference formula, modified to find the second derivative instead of the first:

$$f''(x) \simeq \frac{d3}{h_1^2}$$

where h_1 is a small interval, and

$$d3 = f(x+h_1) - 2f(x) + f(x-h_1)$$

It might seem that we have just exchanged finding one interval, *hh*, for another interval, h_1. However, since we are just trying to find a crude estimate to f''(x), it turns out that we don't have to have a precise value for h_1. I arbitrarily chose

$h_1 = x/1000$ if $x \neq 0$, or
$h_1 = 0.1$ if $x=0$

If the function is fairly linear near *x*, d3 may equal 0. If this happens, it means that f'(x) is also near zero, so we can use any small value for d3; I chose d3 = 0.01, which avoids division by zero in the equation for hh above.

Next, if f(x) = 0, we'll get h = 0, which won't work. In this case, I set h = 0.01.

So, the final equation is

$$h = \frac{\sqrt{\frac{abs(f(x))}{f''(x)}}}{10}$$

where I have used the absolute value *abs()* to ensure that the root is real. I also divide the final result by 10 to ensure that *h* is small enough that *nder1()* doesn't terminate too early.

nder1() can terminate in one of two ways: if the error keeps decreasing, *nder1()* will run until the *amat[]* matrix is filled with values. However, if at some step the error increases, *nder1()* will terminate early. The variable *safe* specifies the magnitude of the error increase that causes termination. Refer to the code listing for details.

From my testing with a few functions, *nder1()* nearly always terminates by exceeding the *safe* limit.