Numerical Differentiation

We shall now look at the problems related to the calculation of derivatives via numerical methods. Now at first thought, it would seem that a numerical calculation of a derivative would be rather straightforward. For the very definition of the derivative

\[ f'(x) = \lim_{h \to 0} \frac{f(x + h) - f(x)}{h} \]

lends itself immediately to a natural numerical approximation for a derivative:

\[ f'(x) \approx \frac{f(x + h) - f(x)}{h}, \quad h << 1. \]

It would thus seem that, if we wanted to get an extremely accurate value for the derivative of a function, all we’d have to do is pick a small enough value for \( h \) and calculate

\[ \frac{f(x + h) - f(x)}{h} \]

Let’s see if this really works. Let \( f \) be the function \( f(x) = \sin(x) \), so that \( f'(x) = \cos(x) \). We will calculate \( f'(1) \) using the formula above using successively small values of \( h \).

```plaintext
f := x -> sin(x);
f1 := x -> cos(x);  # df/dx
x0 := 1.0;            # sample point
fexact := f1(x0);    # the exact result for df/dx at x=1.0
lprint('fexact =',fexact);  # print value to screen
lprint('');           # print a blank line
h := 0.5;            # initial value of h
n := 15;             # number of iterations
for i from 1 to n do
   Deltaf := f(x0 + h) - f(x0);
f1approx := Deltaf/h;
error := fexact - f1approx;
lprint(i,'h =',h,f1approx,'error =',error):
   h := h/10;
end:
```

The output of this program is

\[
\begin{align*}
\text{f1exact} & = 0.5403023059 \\
1 & \quad h = 0.5 \quad \text{f1approx} = 0.3120480036 \quad \text{error} = 0.2282543023 \\
2 & \quad h = 0.0000000000e-1 \quad \text{f1approx} = 0.5190448160 \quad \text{error} = 0.2125748999e-1 \\
3 & \quad h = 0.0000000000e-2 \quad \text{f1approx} = 0.5381963800 \quad \text{error} = 0.2105925989e-2 \\
4 & \quad h = 0.0000000000e-3 \quad \text{f1approx} = 0.5403023000 \quad \text{error} = 0.21030598e-3
\end{align*}
\]
Here's one simple way to do that. Suppose we take the difference of the following two second order Taylor formulae

\begin{align*}
5 & \quad h = .50000000000e-4 \quad f_1 \approx = .5402820000 \quad \text{error} = .203059e-4 \\
6 & \quad h = .50000000000e-5 \quad f_1 \approx = .5403000000 \quad \text{error} = .23059e-5 \\
7 & \quad h = .50000000000e-6 \quad f_1 \approx = .5404000000 \quad \text{error} = -.976941e-4 \\
8 & \quad h = .50000000000e-7 \quad f_1 \approx = .5400000000 \quad \text{error} = .3023059e-3 \\
9 & \quad h = .50000000000e-8 \quad f_1 \approx = .5400000000 \quad \text{error} = .3023059e-3 \\
10 & \quad h = .50000000000e-9 \quad f_1 \approx = 1.000000000 \quad \text{error} = -.4596976941 \\
11 & \quad h = .50000000000e-10 \quad f_1 \approx = 0 \quad \text{error} = .5403023059 \\
12 & \quad h = .50000000000e-11 \quad f_1 \approx = 0 \quad \text{error} = .5403023059 \\
13 & \quad h = .50000000000e-12 \quad f_1 \approx = 0 \quad \text{error} = .5403023059 \\
14 & \quad h = .50000000000e-13 \quad f_1 \approx = 0 \quad \text{error} = .5403023059 \\
15 & \quad h = .50000000000e-14 \quad f_1 \approx = 0 \quad \text{error} = .5403023059 \\
\end{align*}

Notice that our closest estimate does not occur for the smallest value of $h$: in fact, once $h$ is smaller than $5 \times 10^{-6}$ our estimates for $f'(1.0)$ get progressively worse. Indeed, even as we approach the optimal value of $h$ we have a problem; for we start losing significant digits at $i = 3$.

The loss of significant digits, of course, can be traced to the *subtraction error* that occurs when we try to compute the difference between two floating point numbers of about the same size: e.g.,

\[ 1.1234567 \times 10^7 - 1.1234566 \times 10^7 = 0.000001 \times 10^7 \]

The complete failure of this algorithm for very small values of $h$ ($i > 10$) has to do with the fact that there is only a discrete set of machine numbers; for once $h$ gets small enough $f(x + h)$ and $f(x)$ will correspond to the same machine number and so their computed difference will be zero.

In summary, we **can not** improve the accuracy of numerical computations of derivatives by simply making $h$ small enough. What we shall try to do instead is to make our computations as accurate as possible for fixed values of $h$.

We'll thus need to analyze the error inherent in the approximation

\[ f'(x) \approx \frac{f(x + h) - f(x)}{h} \]

Recall that the 1st order Taylor formula (with Lagrange Remainder)

\[ f(x + h) = f(x) + f'(x)h + \frac{1}{2}f''(\xi_x)h^2 \]

is an exact identity for some point $\xi_x \in [x, x + h]$. Solving this equation for $f'(x)$ we obtain

\[ f'(x) = \frac{f(x + h) - f(x)}{h} - \frac{1}{2}f''(\xi_x)h \]

This tells us that the error involved in estimating $f'(x)$ using (18.1) is of order $h$. Now as we seen above, making $h$ smaller as a means of improving our accuracy is only effective up to a point (the point where subtraction and floating point errors kick in). We might therefore look for means for estimating $f'(x)$ such that the error term is of higher order in $h$.

Here's one simple way to do that. Suppose we take the difference of the following two second order Taylor formulae

\begin{align*}
5 & \quad f(x + h) = f(x) + f'(x)h + \frac{1}{2}f''(x)h^2 + \frac{1}{6}f'''(\xi_1)h^3, \quad \xi_1 \in [x, x + h] \\
6 & \quad f(x - h) = f(x) - f'(x)h + \frac{1}{2}f''(x)h^2 - \frac{1}{6}f'''(\xi_2)h^3, \quad \xi_2 \in [x - h, x] \\
\end{align*}
\[
\begin{align*}
\Rightarrow & \quad f(x + h) - f(x - h) = 2f'(x)h + \frac{1}{6} (f'''(\xi_1) - f'''(\xi_1)) h^3 \\
\Rightarrow & \quad f'(x) = \frac{f(x + h) - f(x - h)}{2h} + \frac{1}{6} \left( f'''(\xi_1) - f'''(\xi_2) \right) h^2 \\
\Rightarrow & \quad f'(x) = \frac{f(x + h) - f(x - h)}{2h} + \frac{1}{6} f'''(\xi) h^2, \quad \xi \in [x - h, x + h]
\end{align*}
\]

where in the last step we have applied the Mean Value Theorem for \( f'''(x) \) on the interval \([x - h, x + h]\). We thus arrive at an estimate for \( f'(x) \) for which the error term is of order \( h^2 \).

If we replace the do-loop in the Maple code above with

```maple
for i from 1 to n do
    Deltaf := f(x0 + h) - f(x0 - h);
    fiapprox := Deltaf/(2*h);
    error := f1exact - fiapprox;
    lprint(i, 'h =', h, 'fiapprox =', fiapprox, 'error =', error);
    h := h/10;
end do;
```

we get the following output

```
1  h = .5  fiapprox = .5180694480  error = .223238879e-1
2  h = .5000000000e-1  fiapprox = .5400772080  error = .2250979e-3
3  h = .5000000000e-2  fiapprox = .5403006590  error = .22559e-5
4  h = .5000000000e-3  fiapprox = .5403023000  error = .59e-8
5  h = .5000000000e-4  fiapprox = .5403030000  error = -.6941e-6
6  h = .5000000000e-5  fiapprox = .5403000000  error = .23059e-5
7  h = .5000000000e-6  fiapprox = .5403000000  error = .23059e-5
8  h = .5000000000e-7  fiapprox = .5400000000  error = .3023059e-3
9  h = .5000000000e-8  fiapprox = .5400000000  error = .3023059e-3
10 h = .5000000000e-9  fiapprox = .8000000000  error = -.2596796941
11 h = .5000000000e-10 fiapprox = 0  error = .5403023059
12 h = .5000000000e-11 fiapprox = 0  error = .5403023059
13 h = .5000000000e-12 fiapprox = 0  error = .5403023059
14 h = .5000000000e-13 fiapprox = 0  error = .5403023059
15 h = .5000000000e-14 fiapprox = 0  error = .5403023059
```

Looking at this data, we see that we have the same problem as before with extremely small values of \( h \). However, we are able to achieve an absolute error of \( 0.59 \times 10^{-8} \) in 4 steps; which is much better than the earlier method (for which the least error was \( 0.23059 \times 10^{-5} \) and which took 6 steps to reach.)
1. Richardson Extrapolation

We can do even better though. Let’s assume \( f(x) \) is a smooth function so that we can write

\[
\begin{align*}
  f(x + h) &= \sum_{k=0}^{\infty} \frac{1}{k!} f^{(k)}(x) h^k = f(x) + f'(x) h + \frac{1}{2} f''(x) h^2 + \frac{1}{6} f'''(x) h^3 + \cdots \\
  f(x - h) &= \sum_{k=0}^{\infty} \frac{1}{k!} f^{(k)}(x) (-h)^k = f(x) - f'(x) h + \frac{1}{2} f''(x) h^2 - \frac{1}{6} f'''(x) h^3 + \cdots
\end{align*}
\]

Note that since the series expansions are infinite, we can regard these as exact equations. Subtracting these two equations and solving for \( f'(x) \) yields

\[
  f'(x) = \frac{1}{2h} [f(x + h) - f(x - h)] - \left[ \frac{1}{3!} f'''(x) h^2 + \frac{1}{5!} f^{(5)}(x) h^4 + \frac{1}{7!} f^{(7)}(x) h^6 + \cdots \right]
\]

Let us write this as

\[
  f'(x) = \phi_0(h) + a_2 h^2 + a_4 h^4 + a_6 h^6 + \cdots
\]

where

\[
\begin{align*}
  \phi_0(h) &= \frac{1}{2h} [f(x + h) - f(x - h)] \\
  a_2 &= \frac{1}{3!} f'''(x) \\
  a_4 &= \frac{1}{5!} f^{(5)}(x) \\
  a_6 &= \frac{1}{7!} f^{(7)}(x)
\end{align*}
\]

This equation should be true for all small \( h \), in particular for \( h/2 \). So we should also have

\[
  f'(x) = \phi_0 \left( \frac{h}{2} \right) + a_2 \left( \frac{h}{2} \right)^2 + a_4 \left( \frac{h}{2} \right)^4 + a_6 \left( \frac{h}{2} \right)^6 + \cdots
\]

If we then subtract 1/3 times equation (18.4) from 4/3 times equation (18.5) we can arrange it so that the terms of order \( h^3 \) cancel, obtaining

\[
  f'(x) = -\frac{1}{3} \phi_0 (h) + \frac{4}{3} \phi_0 \left( \frac{h}{2} \right) - \frac{1}{4} a_4 h^4 - \frac{5}{16} a_6 h^6 + \cdots
\]

We thus achieve an expression for \( f'(x) \) where the error term is of order \( h^4 \).

Having achieved this success, we might as well continue. Equation (18.6) is good (and in fact, exact) for all sufficiently small \( h \). Setting

\[
\begin{align*}
  \phi_1(h) &= \frac{1}{3} \phi_0(h) + \frac{4}{3} \phi_0 \left( \frac{h}{2} \right) \\
  b_4 &= -\frac{1}{4} a_4 \\
  b_6 &= -\frac{5}{16} a_6
\end{align*}
\]

we can again write down two equivalent expressions for \( f'(x) \)

\[
\begin{align*}
  f'(x) &= \phi_1(h) + b_4 h^4 + b_6 h^6 + \cdots \\
  f'(x) &= \phi_1(h/2) + \frac{1}{16} b_4 h^4 + \frac{1}{64} b_6 h^6 + \cdots
\end{align*}
\]
If we then subtract \( \frac{1}{15} \) times the first equation from \( \frac{16}{15} \) times the first we obtain

\[
f' (x) = - \frac{1}{15} \phi_1 (h) + \frac{16}{15} \phi_1 \left( \frac{h}{2} \right) + \frac{1}{15} \left( \frac{1}{4} - 1 \right) b_6 h^6
\]

We thus obtain an expression for \( f'(x) \)

\[
f'(x) \approx \phi_2 (x) \equiv - \frac{1}{15} \phi_1 (h) + \frac{16}{15} \phi_1 \left( \frac{h}{2} \right)
\]

that is accurate to order \( h^6 \). It should be clear that this process can be continued until

Let’s now turn this into a numerical algorithm. The first thing to do is to identify the pattern that mediates the successive expressions for \( f(x) \). We have

\[
\phi_0 (h) \equiv \frac{f(x + h) - f(x - h)}{2h}
\]

\[
\phi_1 (h) = - \frac{1}{3} \phi_0 (h) + \frac{4}{3} \phi_0 \left( \frac{h}{2} \right) = - \frac{1}{4^1 - 1} \phi_0 (h) + \frac{4^1}{4^1 - 1} \phi_0 \left( \frac{h}{2} \right)
\]

\[
\phi_2 (h) = - \frac{1}{15} \phi_1 (h) + \frac{16}{15} \phi_1 \left( \frac{h}{2} \right) = - \frac{1}{4^2 - 1} \phi_1 (h) + \frac{4^2}{4^2 - 1} \phi_1 \left( \frac{h}{2} \right)
\]

and so it would seem

\[
\phi_i (h) = - \frac{1}{4^i - 1} \phi_{i-1} (h) + \frac{4^i}{4^i - 1} \phi_{i-1} \left( \frac{h}{2} \right)
\]

Before translating this into computer code. Let’s make the following definition. Let

\[
R_h [n, i] := \phi_i \left( h 2^{-n} \right)
\]

We then have

\[
R_h [n, 0] \equiv \phi_0 \left( 2^{-n} \right) = \frac{f(x + 2^{-n} h) + f(x - 2^{-n} h)}{2^{-n+1} h}
\]

and the recursive formulae

\[
R_h [n, i] \equiv - \frac{1}{4^i - 1} R_h [n, i - 1] + \frac{4^i}{4^i - 1} R_h [n + 1, i - 1]
\]

This quantity \( R_h [n, i] \) will be the \( i^{th} \) order Richardson Expolation of \( f'(x) \) with \( h = 2^{-n} \).

The following program calculates the fourth order Richardson Extrapolation of \( f'(1.0) \) for \( f(x) = \sin(x) \).

```plaintext
R[0] := (f(x+h) - f(x-h))/(2*h);
for i from 1 to 4 do
    p1 := R[i-1]/(4^i - 1);
    p2 := (4^i)*subs({h=h/2},p1);
    R[i] := p2 - p1;
do:
R4 := R[4];
f := x -> evalf(sin(x));
dfapprox := (x1,h1) -> subs({x=x1,h=h1},R4);
x0 := 1.0;
h0 := 0.1;
for i from 1 to 10 do
    dfR4 := evalf(dfapprox(x0,h0));
    lprint('h =',h0,'dfR4 = ', dfR4);
```
h0 := h0/10;

This program produces the following output.

\[
\begin{array}{ll}
\hline
h & dfR4 \\
\hline
.1 & .5403023038 \\
.1000000000e-1 & .5403023410 \\
.1000000000e-2 & .5403028570 \\
.1000000000e-3 & .5403029364 \\
.1000000000e-4 & .5402236374 \\
.1000000000e-5 & .5418266476 \\
.1000000000e-6 & .506746747 \\
.1000000000e-7 & .71775075 \\
.1000000000e-8 & 2.9495862 \\
.1000000000e-9 & 0.0 \\
\hline
\end{array}
\]

Of course, exact answer is \( \cos(1.0) = 0.5403023059 \). We thus see that we can achieve a very accurate result (correct to 7 decimal places on the very first iteration. The fact that we don’t get much better results for smaller values of \( h \) is of course due to the fact that, for a given value of \( h \), subtraction errors kick in much earlier for the Richardson Extrapolation. For example in computing the fourth order Richardson Extrapolation the program needs to calculate

\[
f \left( x + \frac{h}{2^n} \right) - f \left( x - \frac{h}{2^n} \right);
\]

And so in practice, when one employs an \( n^{th} \) order Richardson Extrapolation one has to be sure that \( h/2^n \) is not too small.