Numerical Differentiation In APL

Bob Smith
Sudley Place Software
Originally Written
2 Sep 2019
Updated
16 Sep 2019

Introduction

While working on the Matrix operator[1], I've found I needed a Numerical Differentiation (ND) operator in order to apply the Matrix operator to non-diagonalizable matrices. This operator uses Numerical Analysis methods to calculate the value of the Derivative of a function at a given point.

Notation

The symbol chosen for this operator is Curly D (∂) (Alt-'D' U+2202) used in mathematics for Partial Derivatives.

For example,

```
      !∂0 ⋄ ¯1g1
¯0.5772156649015197
¯0.577215664901532 9
      !∂∂0 ⋄ 1g2+1p2÷6
1.978111990652564 6
1.97811199065594 5
```

That is, exact value of the first derivative of the Factorial function at 0 is -γ, where γ is Gamma, the Euler-Mascheroni constant, and the

exact value of the second derivative is $\gamma^2+\pi^2/6$. The derived function may be called dyadically, too:

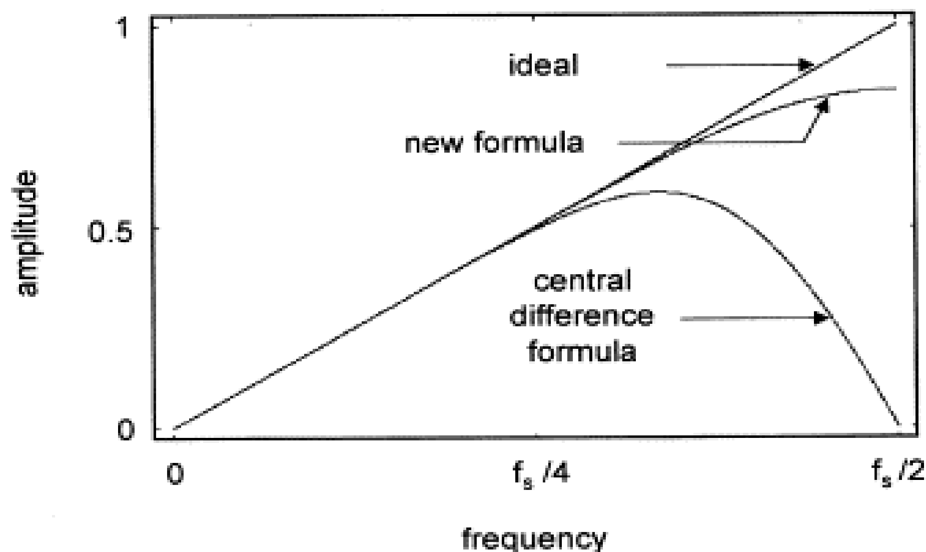
```
      X←0.6 ⋄ 1○∂X ⋄ 2○X
0.8253356149096904
0.8253356149096783
```

where the first derivative of the Sine function is the Cosine function. In this and following examples, the rightmost of consecutive digits that all agree with the exact result is marked.

Accuracy

This topic has many interesting facets. During a literature search, I found a 20-year old paper[3] that describes a new set of differencing formulas for ND – when I implemented them in APL, they turned out to have superior accuracy. In fact, when compared to the ND routines in the GNU Scientific Library (GSL), the new approximations are consistently more accurate – up to 13 or 14 significant digits vs. 9 or 10 for GSL. The accuracy on both Multiple-Precision FP and Ball Arithmetic numbers is even more impressive because there the precision can be increased so as to deliver a result with exactly the desired precision.

I became convinced that this algorithm was worth investigating when I saw the following “worth a thousand words” picture in the original[3] paper:



Classification

This algorithm is in the class of ND differencing algorithms, all based on Taylor series[5] expansions. This type of algorithm samples nearby points around the target point **x**, evaluates the function at these points, differences them in pairs, weights those differences, and then adds them together to achieve the final result. For example, the first order approach is to calculate **f(x+h) – f(x-h)** all divided by **2h**, where **h** is some small number. This means the function is sampled at two points **x±h**, differenced (subtract one from the other) and weighted by **1/2h**. The parameters to this algorithm are the Sampling Frequency (**h**, the radius around **x**) and the Sample Size (number of points, e.g. **x±h**, **x±h/2**, **x±h/4**, etc.). All ND differencing algorithms use some variation on this template.

APL Code

The (origin-1) APL code to implement this algorithm is as follows:

The three values below are used to bridge the difference between even and odd degrees of the Derivative.

```
c←⌊(p-1)÷2 ⋄ c1←0=2|c ⋄ c2←0=2|p
```

The following lines calculate into `tab` *the weights on the sample values. These calculations are stored entirely in a static table in the library code. The Variant operator on the Factorial function implements a Pochhammer Symbol*[6] *which is a generalization of Double Factorial as in !!(2N-1) = (2N-1)×(2N-3)×(2N-5)…1.*

```
X←×/¨(⊂[2]010 2‼c (N-1))∘.(⎕¨)⊂¨(¯1+2×(⊂⍳N)~¨⍳N)÷2
X←+⌿X*¯2
j←(1-N)..N
Cnj←((!⍠((-N)2)¯1+2×N)*2)÷(4*N)×(!N-j)×!N+j-1
tab←(¯1*j+c1)×((!p)÷((¯1+2×j)÷2)*2+c2)×Cnj×(⌽X),X
```

All of the lines above (including the calculation of `c`*,* `c1`*, and* `c2`*) are part of the static tables when calculated across all relevant* `p` *and* `N` *as a* `p` *by* `N` *matrix of length* `2×N` *vectors. In this implementation, I chose* `p←1..9` *and* `N←1..7`*. In practice, the above calculations are all done as Rational Numbers so as not to lose precision were they to be done in Floating Point.*

The following three lines constitute the entire algorithm exclusive of generation of the static tables:

Call the function on the `2×N` *Sample Values spaced apart by* `T` *around* `R` *weighting the result by the appropriate* `2×N` *length vector from* `tab`*.*

```
w←p N⊃tab ◇ Z←w+.×F R+T×(¯1+2×(1-N)..N)÷2
```

The sum of the weights is `0` *for odd degrees. For even degrees that sum is the weight on the value of function at the center point whose product is subtracted from the weighted sum above.*

```
:if 0=2|p ◇ Z-←(+/w)×F R ◇ :end
```

Finally, scale the result by `T` *for each degree of the derivative.*

```
Z÷←T*p
```

where `F` is the function whose derivative is being calculated, `N` is the Order (Sample Size), `p` is the Degree (1st, 2nd,3rd), and `T` is the Sampling Frequency (10^{SFE}) of the Derivative.

Benefits

Delightfully, this paper (along with two related papers[2, 4] from the same authors) provides a number of benefits:

- The algorithm is quite simple as seen above
- Because it is so simple, it extends easily to
  - Double-Precision FP numbers (GSL)

- Multiple-Precision FP numbers (MPFR)
- Ball Arithmetic numbers (ARB)
- Varying sample sizes and spacing
- Higher degrees such as 2nd, 3rd, 4th derivative, etc.
- Hypercomplex numbers

I have finished implementing all of this except the extension to Hypercomplex numbers. The latter looks doable, but I don't understand the theory as yet.

Interestingly, the degree of the Derivative is obtained simply by counting the number of occurrences of ∂ in the Left Operand. This number is passed to the basic library routine which uses it (along with the Order) as an index into a two-dimensional array (`p` by `N`) of vectors of length `2×N` to obtain the weights on the values of the function at the `2×N` sampling points. Because the Degree is used solely as an index to an array to retrieve a vector of weights, there is no performance impact whatsoever for using one Degree over another.

I intend to contribute this algorithm (already written in C) to each of the GSL, MPFR, and ARB open source libraries.

Forward and Backward Differencing

Central Differencing samples points on both sides of a center point which works fine for a function everywhere defined. However if you need a derivative of a function undefined below a certain point and you need the derivative at that point, Central Differencing won't work.

Forward differencing samples the values of the function at or above the given point, never below. For example, in the set of Real numbers, the square root of `X` is undefined below `0`, and so if you want to calculate a derivative of that function at `0`, you'll need Forward Differencing. Backward Differencing is simply the dual to Forward Differencing.

At the moment, only Central Differencing is implemented, but this

might be extended in the future.

Variants

The Variant operator has been extended to the ND operator so as to gain access to finer control of the algorithm. In particular, it may be used to override the default Order (5) of the derivative as well as the Sampling Frequency Exponent as in `f∂⍠(Ord SFE) R`, where `Ord` must be first, `SFE` second. To override just the Order, use `f∂⍠Ord R`. To change the Sampling Frequency Exponent but not the Order, use `0` for the Order.

For example, trying several different Orders:

```
      ⎕FPC←128 ⋄ ⎕PP←40
      !∂⍠4 0v
¯0.5772156649015328606065120900819721716 34
      !∂⍠5 0v   ⍝ Default Order
¯0.5772156649015328606065120900824024251 36
      !∂⍠6 0v
¯0.5772156649015328606065120900824024038 75
      !∂⍠7 0v
¯0.5772156649015328606065120900824024445 79
      ¯1g1v     ⍝ Exact value
¯0.5772156649015328606065120900824024310 43
```

For this example at least, Order `5` provides a marked improvement over the previous Order without much improvement above that.

The default Sampling Frequency Exponent is

```
      SFE←-⌊0.5+⎕FPC÷32
```

which translates to a Sampling Frequency of

```
      SF←10*SFE
```

This value is used as the spacing between Sample Values passed to the function. The `SFE` may be overridden as described above.

Forward and Backward Differencing may be specified by including a

character scalar anywhere in the Right Operand of the Variant operator. The possibilities are `'b'`, `'f'`, and `'c'` for Backward, Forward, and Central Differencing. Again, at the moment, only Central Differencing is implemented in the new code.

For reference, the original Order 2 ND code from GSL is available through the Variant operator by using the uppercase letter in the above list. The GSL code implements all three forms of Differencing, and has been translated into both Multiple-Precision Floating Point and Ball Arithmetic.

For example,

```
      !∂⍠'C' 0   ⍝ GSL code
¯0.5772156648930862
      !∂⍠'c' 0   ⍝ New code
¯0.5772156649015363
      !∂⍠'C' 0±  ⍝ GSL Ball code
¯0.577215664901532860606512052695926256206±2.2E¯24
      !∂⍠'c' 0±  ⍝ New Ball code, same as !∂0±
¯0.577215664901532860606512090082402327882±1.6E¯33
      ¯1g1v      ⍝ Exact value
¯0.577215664901532860606512090082402431043
```

Idioms and Inverses

An extension to this approach to ND is one taken by J which first attempts to recognize the ND operator's Right Operand function as an idiom by looking up its derivative in a table and, if successful, execute the derivative function directly on the argument. For example, the (1st) derivative of sine (`1○ω`) is cosine (`2○ω`), easily resolved by a table lookup. Failing that, numerically differentiate the function.

A successful table lookup can both speed up the calculation and improve the accuracy of the result. Moreover, higher degree derivatives can be calculated through recursive table lookups.

Another opportunity for idiom recognition used by J is when evaluating

the inverse of the Derivative which is expressed in APL as `f∂⍣¯1 ω`. Both of these excellent ideas are on my list of future work.

Online Version

This paper is an ongoing effort and can be out-of-date the next day. To find the most recent version, go to http://sudleyplace.com/APL/ and look for the title of this paper on that page.

Executable Version

The latest Alpha version of the NARS2000 software may be found in http://www.nars2000.org/download/binaries/alpha/ in either 32- or 64-bit versions. This software runs natively under Microsoft Windows XP or later as well as any Linux or Mac OS version which supports Wine (32-bit only) which acts as a translation layer.

References

1. NARS2000 Wiki, “Matrix Operator” http://wiki.nars2000.org/index.php/Matrix
2. “Closed-form Expressions for the Finite Difference Approximations of First and Higher Derivatives Based on Taylor Series”, I.R. Khan, R. Ohba / Journal of Computational and Applied Mathematics 107 (1999) pp. 179-193
3. “New Finite Difference Formulas for Numerical Differentiation”, I.R. Khan, R. Ohba / Journal of Computational and Applied Mathematics 126 (2000) pp. 269-276
4. “Taylor Series Based Finite Difference Approximations of Higher-Degree Derivatives”, I.R. Khan, R. Ohba / Journal of Computational and Applied Mathematics 154 (2003) pp. 115-124
5. Wikipedia, “Taylor Series”, https://en.wikipedia.org/wiki/Taylor_series

6. NARS2000 Wiki, “Variant – Rising and Falling Factorials”, http://wiki.nars2000.org/index.php/Variant#Rising_and_Falling_Factorials