Complex-Step Differentiation

Sébastien Boisgérault, Mines ParisTech, under CC BY-NC-SA 4.0

March 24, 2016

Contents

Introduction

You may already have used numerical differentiation to estimate the derivative of a function, using for example Newton's finite difference approximation

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}.$$

The implementation of this scheme in Python is straightforward:

```
def FD(f, x, h):
    return (f(x + h) - f(x)) / h
```

However, the relationship between the value of the step h and the accuracy of the numerical derivative is more complex. Consider the following sample data:

$\exp'(0)$	`FD(exp, 0, 1e-12)`	`FD(exp, 0, 1e-8)`	`FD(exp, 0, 1e-4)`
1	`1.000088900582341`	`0.99999999392252903`	`1.000050001667141`

The most accurate value of the numerical derivative is obtained for $h = 10^{-8}$ and only 8 digits of the result are significant. For the larger value of $h = 10^{-4}$, the accuracy is limited by the quality of the Taylor development of exp at the first order; this truncation error decreases linearly with the step size. For the smaller value of $h = 10^{-12}$, the accuracy is essentially undermined by round-off errors in computations.

In this document, we show that *complex-step differentiation* may be used to get rid of the influence of the round-off error for the computation of the first derivative. For higher-order derivatives, we introduce a *spectral method*, a fast algorithm with an error that decreases exponentially with the number of function evaluations.

Computer Arithmetic

You may skip this section if you are already familiar with the representation of real numbers as "doubles" on computers and with their basic properties. At the opposite, if you wish to have more details on this subject, it is probably a good idea to have a look at the classic "What every computer scientist should know about computer arithmetic" (Goldberg 1991).

In the sequel, the examples are provided as snippets of Python code that often use the Numerical Python (NumPy) library; first of all, let's make sure that all NumPy symbols are available:

```
>>> from numpy import *
```

Floating-Point Numbers: First Contact

The most obvious way to display a number is to print it:

```
>>> print pi
3.14159265359
```

This is a lie of course: `print` is not supposed to display an accurate information about its argument, but something readable. To get something unambiguous instead, we can do:

```
>>> pi
3.141592653589793
```

When we say "unambiguous", we mean that there is enough information in this sequence of digits to compute the original floating-point number; and indeed:

```
>>> pi == eval("3.141592653589793")
True
```

Actually, this representation is *also* a lie: it is not an exact decimal representation of the number `pi` stored in the computer memory. To get an exact representation of `pi`, we can request the display of a large number of the decimal digits:

```
>>> def all_digits(number):
...     print "{0:.100g}".format(number)
>>> all_digits(pi)
3.1415926535897931159979634685441851615905761718750
```

Asking for 100 digits was actually good enough: only 49 of them are displayed anyway, as the extra digits are all zeros.

Note that we obtained an exact representation of the floating-point number `pi` with 49 digits. That does *not* mean that all – or even most – of these digits are significant in the representation the real number of π. Indeed, if we use the Python library for multiprecision floating-point arithmetic mpmath, we see that

```
>>> import mpmath
>>> mpmath.mp.dps = 49; mpmath.mp.pretty = True
>>> +mpmath.pi
3.1415926535897932384626433832795028841971693993751
```

and both representations are identical only up to the 16th digit.

Binary Floating-Point Numbers

Representation of floating-point numbers appears to be complex so far, but it's only because we insist on using a *decimal* representation when these numbers are actually stored as *binary* numbers. In other words, instead of using a sequence of *(decimal) digits* $f_i \in \{0, 1, \ldots, 9\}$ to represent a real number x as

$$x = \pm(f_0.f_1 f_2 \ldots f_i \ldots) \times 10^e$$

we should use *binary digits* – aka *bits* – $f_i \in \{0, 1\}$ to write:

$$x = \pm(f_0.f_1 f_2 \ldots f_i \ldots) \times 2^e.$$

These representations are *normalized* if the leading digit of the *significand* $(f_0.f_1 f_2 \ldots f_i \ldots)$ is non-zero; for example, with this convention, the rational number 999/1000 would be represented in base 10 as 9.99×10^{-1} and not as

0.999×10^0. In base 2, the only non-zero digit is 1, hence the significand of a normalized representation is always $(1.f_1 f_2 \dots f_i \dots)$.

In scientific computing, real numbers are usually approximated to fit into a 64-bit layout named "double"[1]. In Python standard library, doubles are available as instances of `float` – or alternatively as `float64` in NumPy.

A triple of

- *sign bit* $s \in \{0, 1\}$,
- *biased exponent* $e \in \{1, \dots, 2046\}$ (11-bit),
- *fraction* $f = (f_1, \dots, f_{52}) \in \{0, 1\}^{52}$.

represents a normalized double

$$x = (-1)^s \times 2^{e-1023} \times (1.f_1 f_2 \dots f_{52}).$$

The doubles that are not normalized are not-a-number (`nan`), infinity (`inf`) and zero (`0.0`) (actually *signed* infinities and zeros), and denormalized numbers. In the sequel, we will never consider such numbers.

With the bitstream Python library, it's actually quite easy to decompose a double into sign bit, biased exponent and significand. Start with

```
>>> from bitstream import BitStream
```

and define the function

```
def s_e_f(number):
    stream = BitStream(number, float)
    s = stream.read(bool)
    e = 0
    for bit in stream.read(bool, 11):
        e = (e << 1) + bit
    f = stream.read(bool, 52)
    return s, e, f
```

Getting the floating-point number from the values of s, e and f is as easy:

```
def number(s, e, f):
    bits = []
    bits.append(s)
    for i in range(11):
        bits.insert(1, bool(e % 2))
        e = e >> 1
    bits.extend(f)
    return BitStream(bits, bool).read(float)
```

[1] **"Double"** is a shortcut for "double-precision floating point format", defined in the IEEE 754 standard, see (IEEE Task P754 1985). A single-precision format is also defined, that uses only 32 bits. NumPy provides it under the name `float32`.

Accuracy

Almost all real numbers cannot be represented exactly as doubles. It makes sense to associate to a real number x the nearest double $[x]$. A "round-to-nearest" method that does this is fully specified in the IEE754 standard (see IEEE Task P754 1985), together with alternate ("directed rounding") methods.

To have any kind of confidence in our computations with doubles, we need to be able to estimate the error in the representation of x by $[x]$. The *machine epsilon*, denoted ϵ in the sequel, is a key number in this respect. It is defined as the gap between 1.0 – that can be represented exactly as a double – and the next double in the direction $+\infty$.

```
>>> after_one = nextafter(1.0, +inf)
>>> after_one
1.0000000000000002
>>> all_digits(after_one)
1.0000000000000002220446049250313080847263336181640625
>>> eps = after_one - 1.0
>>> all_digits(eps)
2.2204460492503130808472633361816406 25e-16
```

This number is also available as an attribute of the `finfo` class of NumPy that gathers machine limits for floating-point data types:

```
>>> all_digits(finfo(float).eps)
2.2204460492503130808472633361816406 25e-16
```

Alternatively, the examination of the structure of normalized doubles yields directly the value of ϵ: the fraction of the number after 1.0 is $(f_1, f_2, \ldots, f_{51}, f_{52}) = (0, 0, \ldots, 0, 1)$, hence $\epsilon = 2^{-52}$, a result confirmed by:

```
>>> all_digits(2**-52)
2.2204460492503130808472633361816406 25e-16
```

The machine epsilon matters so much because it provides a simple bound on the relative error of the representation of a real number as a double. Indeed, for any sensible rounding method, the structure of normalized doubles yields

$$\frac{|[x] - x|}{|x|} \leq \epsilon.$$

If the "round-to-nearest" method is used, you can actually derive a tighter bound: the inequality above still holds with $\epsilon/2$ instead of ϵ.

Significant Digits

This relative error translates directly into how many significant decimal digits there are in the best approximation of a real number by a double. Consider the

exact representation of $[x]$ in the scientific notation:

$$[x] = \pm(f_0.f_1 \dots f_{p-1} \dots) \times 10^e.$$

We say that it is significant up to the p-th digit if

$$|x - [x]| \leq \frac{10^{e-(p-1)}}{2}.$$

On the other hand, the error bound on $[x]$ yields

$$|x - [x]| \leq \frac{\epsilon}{2}|x| \leq \frac{\epsilon}{2} \times 10^{e+1}.$$

Hence, the desired precision is achieved as long as

$$p \leq -\log_{10} \epsilon/2 = 52 \log_{10} 2 \approx 15.7.$$

Consequently, doubles provide a 15-th digit approximation of real numbers.

Functions

Most real numbers cannot be represented exactly as doubles; accordingly, most real functions of real variables cannot be represented exactly as functions operating on doubles either. The best we can hope for are *correctly rounded* approximations. An approximation $[f]$ of a function f of n variables is *correctly rounded* if for any n-uple $(x_1, \dots, x_n)$, we have

$$[f](x_1, \dots, x_n) = [f([x_1], \dots, [x_n])].$$

The IEEE 754 standard (see IEEE Task P754 1985) mandates that some functions have a correctly rounded implementation; they are:

> add, substract, multiply, divide, remainder and square root.

Other standard elementary functions – such as sine, cosine, exponential, logarithm, etc. – are usually *not* correctly rounded; the design of computation algorithms that have a decent performance and are *provably* correctly rounded is a complex problem (see for example the documentation of the Correctly Rounded mathematical library).

Complex Step Differentiation

Forward Difference

Let f be a real-valued function defined in some open interval. In many concrete use cases, we can make the assumption that the function is actually analytic

and never have to worry about the existence of derivatives. As a bonus, for any real number x in the domain of the function, the (truncated) Taylor expansion

$$f(x+h) = f(x) + f'(x)h + \frac{f''(x)}{2}h^2 + \cdots + \frac{f^{(n)}}{n!}h^n + \mathcal{O}(h^{n+1})$$

is locally valid[2]. A straighforward computation shows that

$$f'(x) = \frac{f(x+h) - f(x)}{h} + \mathcal{O}(h)$$

The asymptotic behavior of this *forward difference* scheme – controlled by the term $\mathcal{O}(h^1)$ – is said to be of order 1. An implementation of this scheme is defined for doubles x and h as

$$\mathrm{FD}(f, x, h) = \left[\frac{[[f]([x] + [h]) - [f](x)]}{[h]}\right].$$

or equivalently, in Python as:

```
def FD(f, x, h):
    return (f(x + h) - f(x)) / h
```

Round-Off Error

We consider again the function $f(x) = \exp(x)$ used in the introduction and compute the numerical derivative based on the forward difference at $x = 0$ for several values of h. The graph of $h \mapsto \mathrm{FD}(\exp, 0, h)$ shows that for values of h near or below the machine epsilon ϵ, the difference between the numerical derivative and the exact value of the derivative is *not* explained by the classic asymptotic analysis.

If we take into account the representation of real numbers as doubles however, we can explain and quantify the phenomenon. To focus only on the effect of the round-off errors, we'd like to get rid of the truncation error. To achieve this, in the following computations, instead of exp, we use $\exp_0$, the Taylor expansion of exp of order 1 at $x = 0$; we have $\exp_0(x) = 1 + x$.

Assume that the rounding scheme is "round-to-nearest"; select a floating-point number $h > 0$ and compare it to the machine epsilon:

- If $h \ll \epsilon$, then $1 + h$ is close to 1, actually, closer to 1 than from the next binary floating-point value, which is $1 + \epsilon$. Hence, the value is rounded to $[\exp_0](h) = 1$, and a *catastrophic cancellation* happens:

$$\mathrm{FD}(\exp_0, 0, h) = \left[\frac{[[\exp_0](h) - 1]}{h}\right] = 0.$$

[2]**Bachmann-Landau notation.** For a real or complex variable h, we write $\psi(h) = \mathcal{O}(\phi(h))$ if there is a suitable deleted neighbourhood of $h = 0$ where the functions ψ and ϕ are defined and the inequality $|\psi(h)| \leq \kappa|\phi(h)|$ holds for some $\kappa > 0$. When N is a natural number, we write $\psi(N) = \mathcal{O}(\phi(N))$ if there is a n such that ψ and ϕ are defined for $N \geq n$ and for any such N, the inequality $|\psi(N)| \leq \kappa|\phi(N)|$ holds for some $\kappa > 0$.

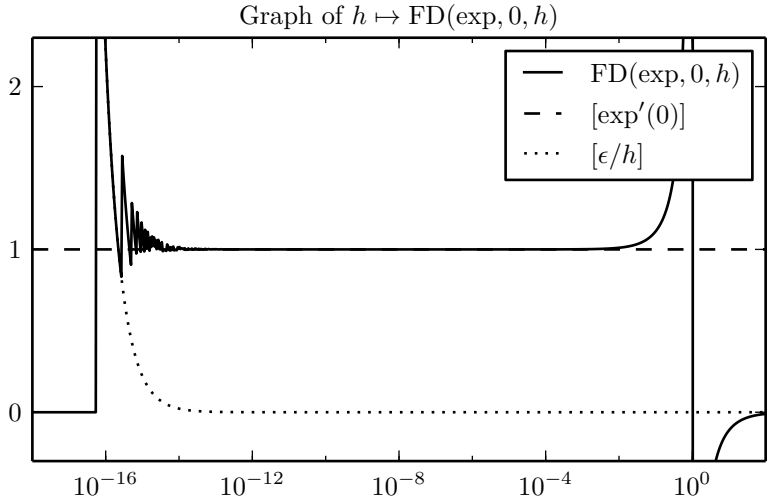

Figure 1: Forward Difference Scheme Values.

- If $h \approx \epsilon$, then $1 + h$ is closer from $1 + \epsilon$ than it is from 1, hence we have $[\exp_0](h) = 1 + \epsilon$ and

$$\mathrm{FD}(\exp_0, 0, h) = \left[\frac{[[\exp_0](h) - 1]}{h}\right] = \left[\frac{\epsilon}{h}\right].$$

- If $\epsilon \ll h \ll 1$, then $[1 + h] = 1 + h \pm \epsilon(1 + h)$ (the symbol $\pm$ is used here to define a confidence interval[3]). Hence

$$[[\exp_0](h) - 1] = h \pm \epsilon \pm \epsilon(2h + \epsilon + \epsilon h)$$

and

$$\left[\frac{[[\exp_0](h) - 1]}{h}\right] = 1 \pm \frac{\epsilon}{h} + \frac{\epsilon}{h}(3h + 2\epsilon + 3h\epsilon + \epsilon^2 + \epsilon^2 h)$$

therefore

$$\mathrm{FD}(\exp_0, 0, h) = \exp_0'(0) \pm \frac{\epsilon}{h} \pm \epsilon',\ \epsilon' \ll \frac{\epsilon}{h}.$$

Going back to $\mathrm{FD}(\exp, 0, h)$ and using a log-log scale to display the total error, we can clearly distinguish the region where the error is dominated by the round-off error – the curve envelope is $\log(\epsilon/h)$ – and where it is dominated by the truncation error – a slope of 1 being characteristic of schemes of order 1.

[3] **Plus-minus sign and confidence interval.** The equation $a = b \pm c$ should be interpreted as the inequality $|a - b| \leq |c|$.

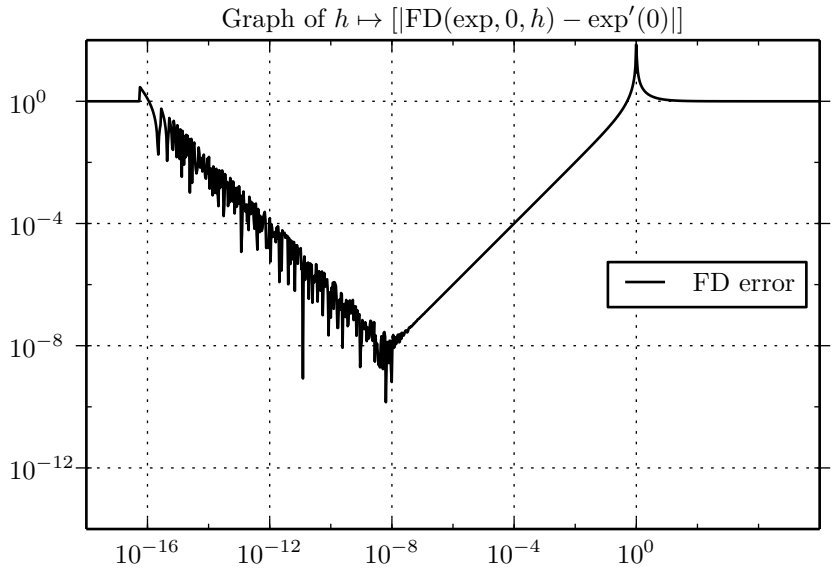

Figure 2: Forward Difference Scheme Error.

Higher-Order Scheme

The theoretical asymptotic behavior of the forward difference scheme can be improved, for example if instead of the forward difference quotient we use a central difference quotient. Consider the Taylor expansion at the order 2 of $f(x+h)$ and $f(x-h)$:

$$f(x+h) = f(x) + f'(x)(+h) + \frac{f''(x)}{2}(+h)^2 + \mathcal{O}\left(h^3\right)$$

and

$$f(x-h) = f(x) + f'(x)(-h) + \frac{f''(x)}{2}(-h)^2 + \mathcal{O}\left(h^3\right).$$

We have

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + \mathcal{O}(h^2),$$

hence, the *central difference* scheme is a scheme of order 2, with the implementation:

$$\mathrm{CD}(f, x, h) = \left[\frac{[[f]([x] + [h]) - [f]([x] - [h])]}{[2 \times [h]]}\right].$$

or equivalently, in Python:

```
def CD(f, x, h):
    return 0.5 * (f(x + h) - f(x - h)) / h
```

The error graph for the central difference scheme confirms that a truncation error of order two may be used to improve the accuracy. However, it also shows that a higher-order actually *increases* the region dominated by the round-off error, making the problem of selection of a correct step size h even more difficult.

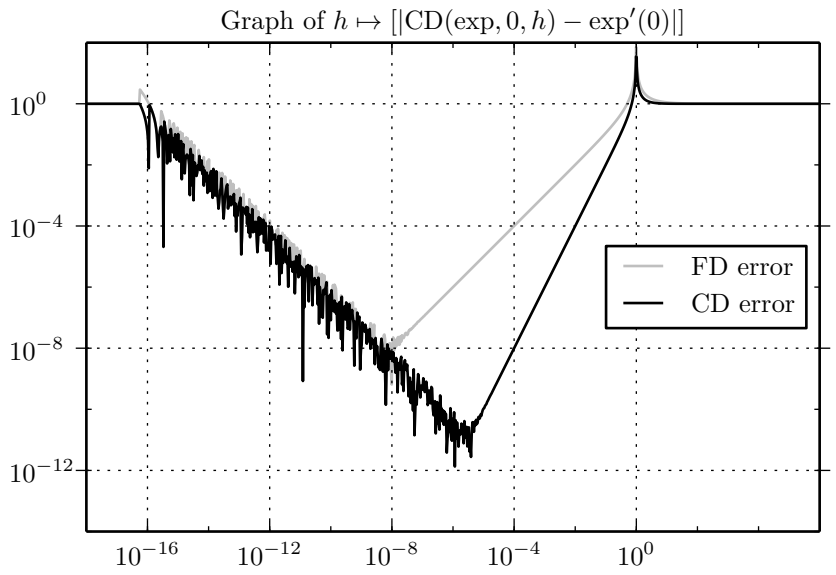

Figure 3: Central Difference Scheme Error.

Complex Step Differentiation

If the function f is analytic at x, the Taylor expansion is also valid for (small values of) complex numbers h. In particular, if we replace h by a pure imaginary number ih, we end up with

$$f(x + ih) = f(x) + f'(x)ih + \frac{f''(x)}{2}(ih)^2 + \mathcal{O}(h^3)$$

If f is real-valued, using the imaginary part yields:

$$\Im\left(\frac{f(x + ih)}{h}\right) = f'(x) + \mathcal{O}(h^2).$$

This is a method of order 2. The straightforward implementation of the complex-step differentiation is

$$\mathrm{CSD}(f, x, h) = \left[\frac{\Im([f]([x] + i[h]))}{[h]}\right].$$

or equivalently, in Python:

```
def CSD(f, x, h):
    return imag(f(x + 1j * h)) / h
```

The distinguishing feature of this scheme: it almost totally gets rid of the truncation error. Indeed, let's consider again $\exp_0$; when x and y are floating-point real numbers, the sum $x + iy$ can be computed with any round-off, hence, if h is a floating-point number, $[\exp_0](ih) = [1 + ih] = 1 + ih$ and consequently, $\Im([\exp_0](ih)) = h$, which yields

$$\mathrm{CSD}(\exp_0, 0, h) = \left[\frac{h}{h}\right] = 1 = \exp_0'(0).$$

Spectral Method

The complex step differentiation is a powerful method but it also has limits. We can use it to compute the first derivative of a real analytic function f, but not its second derivative because our estimate $x \mapsto \mathrm{CSD}(f, x, h)$ of the first derivative is only available for real values of x, hence the method cannot be iterated. We cannot use it either if we know that f is analytic but not real-valued.

We introduce in this section an alternate method to compute first, second and higher-order derivatives of – real or complex-valued – analytic functions. More details may be found in (Fornberg 2006) and (Trefethen 2000).

Computation Method

Let f be a function that is holomorphic in an open neighbourhood of $x \in \mathbb{R}$ that contains the closed disk with center x and radius r. In this disk, the values of f can be computed by the Taylor series

$$f(z) = \sum_{n=0}^{+\infty} a_n (z - x)^n, \quad a_n = \frac{f^{(n)}(x)}{n!}.$$

The coefficients of this series satisfy[4]

$$\exists\, \kappa > 0, \forall\, n \in \mathbb{N},\ |a_n| \leq \kappa\, r^{-n}.$$

[4] **Growth bound.** The number r has to be smaller than the radius of convergence of the series:

$$r < \left(\limsup_{n \to +\infty} \sqrt[n]{|a_n|}\right)^{-1}.$$

This is equivalent to $\limsup_{n\to+\infty} \sqrt[n]{|a_n r^n|} < 1$; consequently, for n large enough we have $|a_n| \leq r^{-n}$. With a suitable constant $\kappa > 0$, we can turn this into $|a_n| \leq \kappa\, r^{-n}$ for *every* integer n.

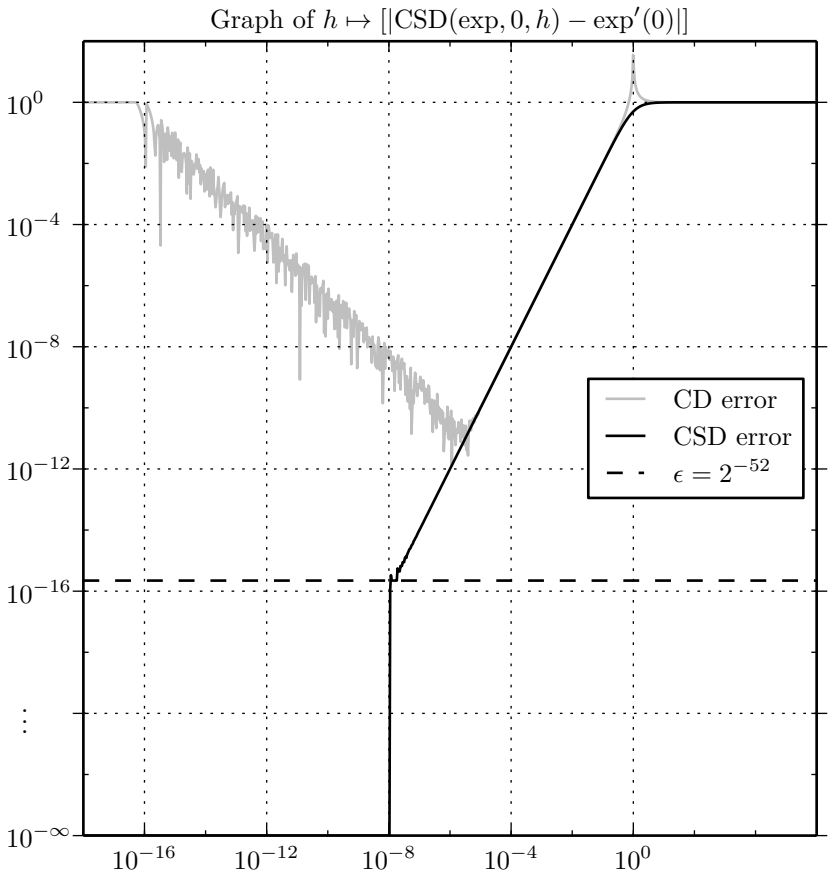

Figure 4: Complex Step Difference Scheme Error.

Let $h \in (0, r)$ and N be a positive integer; let f_k be the sequence of N values of f on the circle with center x and radius h defined by

$$f_k = f(x + hw^k),\ w = e^{-i2\pi/N},\ k = 0, \ldots, N-1.$$

Estimate and Accuracy

The values f_k can be computed as

$$f_k = \sum_{n=0}^{+\infty} a_n (hw^k)^n = \sum_{n=0}^{N-1} \left[\sum_{m=0}^{+\infty} a_{n+mN} h^{n+mN} \right] w^{k(n+mN)}.$$

Notice that we have $w^{k(n+mN)} = w^{kn}(w^N)^{km} = w^{kn}$. Hence, if we define

$$c_n = a_n h^n + a_{n+N} h^{n+N} + \cdots = \sum_{m=0}^{+\infty} a_{n+mN} h^{n+mN},$$

we end up with the following relationship between the values f_k and c_n:

$$f_k = \sum_{n=0}^{N-1} w^{kn} c_n.$$

It is useful because the coefficients c_n/h^n provide an approximation of a_n:

$$\left| a_n - \frac{c_n}{h^n} \right| \leq \kappa r^{-n} \sum_{m=1}^{+\infty} (h/r)^{mN} = \kappa r^{-n} \frac{(h/r)^N}{1 - (h/r)^N}$$

There are two ways to look at this approximation: if we freeze N and consider the behavior of the error when the radius h approaches 0, we derive

$$a_n = \frac{c_n}{h^n} + \mathcal{O}(h^N)$$

and conclude that the approximation of a_n is of order N with respect to h; on the other hand, if we freeze h and let N grow to $+\infty$, we obtain instead

$$a_n = \frac{c_n}{h^n} + \mathcal{O}(e^{-\alpha N}) \ \text{ with } \ \alpha = -\log(h/r) > 0,$$

in other words, the approximation is exponential with respect to N.

Computation of the Estimate

The right-hand side of the equation

$$f_k = \sum_{n=0}^{N-1} w^{kn} c_n.$$

can be interpreted as a classic matrix-vector product; the mapping from the c_n to the f_k is known has the *discrete Fourier transform* (DFT). The inverse mapping – the *inverse discrete Fourier transform* – is given by

$$c_n = \frac{1}{N} \sum_{n=0}^{N-1} w^{-kn} f_k.$$

Both the discrete Fourier transform and it inverse can be computed by algorithms having a $\mathcal{O}(N \log N)$ complexity (instead of the $\mathcal{O}(N^2)$ of the obvious method), the aptly named *fast Fourier transform* (FFT) and *inverse fast Fourier transform* (IFFT). Some of these algorithms actually deliver the minimal complexity only when N is a power of two, so it is safer to pick only such numbers if you don't know exactly what algorithm you are actually using.

The implementation of this scheme is simple:

```
from numpy.fft import ifft
from scipy.misc import factorial

def SM(f, x, h, N):
    w = exp(-1j * 2 * pi / N)
    k = n = arange(N)
    f_k = f(x + h * w**k)
    c_n = ifft(f_k)
    a_n = c_n / h ** n
    return a_n * factorial(n)
```

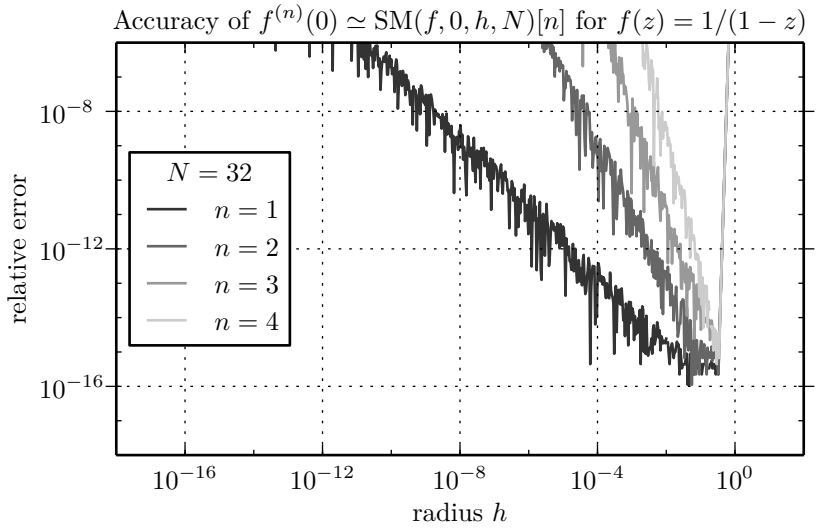

Figure 5: Spectral Method Error

Error Analysis

The algorithm introduced in the previous section provides approximation methods with an arbitrary large order for n-th order derivatives. However, the region in which the round-off error dominates the truncation error is large and actually *increases* when the integer n grows. A specific analysis has to be made to control both kind of errors.

We conduct the detailled error analysis for the function

$$f(z) = \frac{1}{1-z}$$

at $x = 0$ and attempt to estimate the derivatives up to the fourth order. We have selected this example because $a_n = 1$ for every n, hence the computation of the relative errors of the results are simple.

Round-off error

We assume that the main source of round-off errors is in the computations of the f_k. The distance between f_k and its approximation $[f_k]$ is bounded by $|f_k| \times \epsilon/2$; the coefficients in the IFFT matrix are of modulus $1/N$, hence, if the sum is exact, we end up with an absolute error on c_n bounded by $M(h) \times \epsilon/2$ with

$$M(h) = \max_{|z-x|=h} |f(z)|.$$

Hence the absolute error on $a_n = c_n/h^n$ is bounded by $M(h)\epsilon/(2h^n)$. Using the rough estimate $|a_n| \simeq \kappa r^{-n}$, we end up with a relative error for a_n controlled by

$$\left(\frac{M(h)}{\kappa}\frac{\epsilon}{2}\right)\left(\frac{h}{r}\right)^{-n}$$

On our example, we can pick $M(h) = 1/(1-h)$, $\kappa = 1$, and $r = 1$, hence the best error bound we can hope for is obtained for the value of h that minimizes $1/((1-h)h^n)\epsilon/2$; the best h and round-off error bound are actually

$$h = \frac{n}{n+1} \quad \text{and} \quad \text{round-off}(a_n) \leq \frac{(n+1)^{n+1}}{n^n}\frac{\epsilon}{2}.$$

The error bound is always bigger than the structural relative error $\epsilon/2$ and increases with n, hence the worst case is obtained for the highest derivative order that we want to compute, that is $n = 4$. If for example we settle on a round-off relative error of 1000 times $\epsilon/2$, we can select $h = 0.2$.

Truncation error

We have already estimated the difference between a_n and c_n/h^n; if we again model $|a_n|$ as κr^{-n}, the relative error of this estimate is bounded by

$$\frac{(h/r)^N}{1-(h/r)^N} \simeq \left(\frac{h}{r}\right)^N,$$

hence to obtain a truncation error of the same magnitude than the truncation error – that is $1000 \times \epsilon/2$, we may select N such that $0.2^N \leq 1000 \times \epsilon/2$, that is

$$N \geq \left\lceil \frac{\log(1000 \times \epsilon/2)}{\log 0.2} \right\rceil = 19.$$

We pick for N the next power of two after 19; the choices $h = 0.2$ and $N = 32$ yield the following estimates of the first 8 n-th order derivatives of f.

n	$f^{(n)}(0)$ estimate	relative error
0	1.0000000000000000	0.0
1	0.9999999999999998	2.2×10^{-16}
2	1.9999999999999984	7.8×10^{-16}
3	6.0000000000000284	4.7×10^{-15}
4	23.999999999999996	1.1×10^{-16}
5	120.00000000001297	1.1×10^{-13}
6	720.00000000016007	2.2×10^{-13}
7	5040.0000000075588	1.5×10^{-12}

Appendix

Augustin-Louis is the proud author of a very simple Python CSD code fragment that people cut-and-paste in their numerical code:

```
from numpy import imag
def CSD(f, x, h=1e-100):
    return imag(f(x + 1j * h)) / h
```

One day, he receives the following mail:

> Dear Augustin-Louis,
>
> We are afraid that your Python CSD code is defective; we used it to compute the derivative of $f(x) = \sqrt{|x|}$ at $x = 1$ and got 0. We're pretty sure that it should be 0.5 instead.
>
> Yours truly,

Isaac & Gottfried Wilhelm.

1. Should the complex-step differentiation method work in this case?
2. How do you think that Isaac and Gottfried Wilhelm have implemented the function f? Would that explain the value of `CSD(f, x=1.0)` that they report?
3. Can you modify the CSD code fragment to "make it work" with the kind of function and implementation that Isaac and Gottried Wilhelm are using? Of course, you cannot change their code, only yours.

Bibliography

Fornberg, Bengt. 2006. "Numerical Differentiation of Analytic Functions." *ACM Trans. Math. Softw.* 7 (4): 512–26. `http://dblp.uni-trier.de/db/journals/toms/toms7.html#Fornberg81`.

Goldberg, David. 1991. "What Every Computer Scientist Should Know About Floating Point Arithmetic." *ACM Computing Surveys* 23 (1): 5–48.

IEEE Task P754. 1985. *ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic*. New York, NY, USA: IEEE.

Trefethen, Lloyd N. 2000. *Spectral Methods in MATLAB*. Vol. 10. Siam. `http://www.mathworks.ir/downloads/english/Spectral%20Methods%20in%20MATLAB.pdf`.

Fornberg, Bengt. 2006. "Numerical Differentiation of Analytic Functions." *ACM Trans. Math. Softw.* 7 (4): 512–26. `http://dblp.uni-trier.de/db/journals/toms/toms7.html#Fornberg81`.

Goldberg, David. 1991. "What Every Computer Scientist Should Know About Floating Point Arithmetic." *ACM Computing Surveys* 23 (1): 5–48.

IEEE Task P754. 1985. *ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic*. New York, NY, USA: IEEE.

Trefethen, Lloyd N. 2000. *Spectral Methods in MATLAB*. Vol. 10. Siam. `http://www.mathworks.ir/downloads/english/Spectral%20Methods%20in%20MATLAB.pdf`.