CS 3430: S24: Lecture 12
Central Divided Difference (CDD) and Richardson's Extrapolation (RE)

Vladimir Kulyukin
Department of Computer Science
Utah State University

Introduction

In this part of lecture 12, we will continue our investigation of differentiation in scientific computing. In lecture 11, we saw how to use differentiation in the Newton-Raphson Algorithm (NRA) to find zero roots of functions.

The ability to find zero roots of polynomials allows us to approximate the values of irrational numbers such as $\sqrt{2}$, $\sqrt{17}$, $\sqrt[3]{3}$, etc.

For example, if we want to approximate $\sqrt{2}$, we can use the NRA to approximate the zero root of $x^2 - 2$, if we want to approximate $\sqrt{3}$, we can use the NRA to approximate a zero root of $x^2 - 3$, if we want to approximate $\sqrt[3]{3}$, we can use the NRA to approximate a zero root of $x^3 - 3$. In general, if i and n are natural numbers greater than 0, and we want to approximate $\sqrt[i]{n}$, we can use the NRA to approximate a zero root of $x^i - n$.

Introduction

We will study the Central Divided Difference (CDD), a method to approximate the value of $f'(x)$ at a given x of some function $f(x)$. We will see how CDD is used by Richardson's Extrapolation (RE) to obtain closer approximations of $f'(x)$ through iteration.

Both CDD and RE are very useful when direct differentiation (i.e., direct computation of the derivative of some function) is not possible/practical. For example, our tiny differentiation engine will choke on functions like $5e^{2.5x}$. Every differentiation engine (including sympy) has its limits.

Introduction

These limits are not just the computational limitations of a given engine (e.g., due to lack of some differentiation rule) but also knowledge engineering limits (e.g., we may not have a mathematician on the team who can tell us how to take a derivative of some really complicated function).

In some projects, we may not even know the function that generated the values we are working with. For example, temperature sensors generate us some values, we do not really know the true temperature function. Weather forecasting is an art.

Under these scenarios, we may want to use CDD and RE.

Central Divided Difference (CDD)

CDD is a method to approximate $f'(x)$ for a given function f and a given value x. The simplest CDD has the following formula:

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

where h is the step size (e.g., $h = 0.1$) and $O(h^2) \approx Ch^2$, for some real constant C. You can think of h as the distance to the right of x and to the left of x so that the whole interval is $[x-h, x+h]$ for a given value of x.

The term $O(h^2)$ estimates the true error. The above CDD formula is sometimes called CDD of order 2, because of the degree 2 of h in the error term.

True Error and Absolute True Relative Error

Let TV be the true value (aka the ground truth). Let AV be the approximate value (aka the estimate). Actually, a more precise way of expressing the estimate is AV_h, because we always approximate the true value (TV) of the $f'(x)$ at a given value of the step size h (e.g., $h = 0.01$).

The true error, E_t, is computed as

$$E_t = TV - AV_h.$$

The absolute true relative error is computed as

$$|E_t| = |(TV - AV_h)/TV| \cdot 100.$$

Behavior of $O(h^2)$ as Approximation of E_t

$O(h^2)$ as the error approximation means that E_t is quartered (compared to its previous value) as the step size h is halved.

Differentiation with RE/CDD

Let us dive into RE. It is a method that uses CDD (of different orders) to obtain better approximations of $f'(x)$, which can be derived with Taylor series.

The 1st Richardson (first RE) (be careful here, because some technical texts refer to it as 0th Richardson) is the CDD formula of order 2. Technically, we can start with CDD of order 4, 8, etc.

Differentiation with RE/CDD

Let TV be a true value, i.e., the true value of $f'(x)$, h be the size of the interval. Furthermore, we assume that we know the function $f(x)$ and can compute AV_h as

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

so that

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

for some real number C.

Differentiation with RE/CDD

Now we can compute the following approximations:

1. $TV \approx AV_h + Ch^2$;
2. $TV \approx AV_{h/2} + C(h/2)^2$;
3. $TV \approx AV_{h/4} + C(h/4)^2$;
4. $TV \approx AV_{h/8} + C(h/8)^2$;
5. $TV \approx AV_{h/16} + C(h/16)^2$;

and on, and on, and on.

Differentiation with RE/CDD

The 1st RE is CDD of order 2.

The 2nd RE (which can be derived from equations 1 and 2 on the previous slide by subtracting 2 from 1) is

$$TV \approx \frac{4AV_{h/2} - AV_h}{3}$$

The 3rd RE (which can be derived from equations 2 and 3 on the previous slide by subracting 3 from 2) is

$$TV \approx AV_{h/4} + \frac{AV_{h/4} - AV_{h/2}}{3}.$$

And on and on and on...

Differentiation with RE/CDD

Thus, if we know $f(x)$ and h, we can implement the function `cdd_1st_drv_ord2(f, x, h)` to compute AV_h at a given value of h. Then we can compute $AV_{h/2}$ as

$$AV_{h/2} = \frac{f\left(x + \frac{h}{2}\right) - f\left(x - \frac{h}{2}\right)}{2\frac{h}{2}}.$$

Example 1

Let $f(x) = 5e^{2.5x}$ and let us approximate $f'(1.0)$ for three values of h: 0.1, 0.05, and 0.025. In other words, we will approximate $f'(x)$ at $x = 1.0$ starting at $h = 0.1$ and halving h twice.

Let us assume that we implemented the Python function `cdd_1st_drv_ord2(f, x, h)`, where `f` is a function, `x` is the point at which we want to evalute the derivative of `f`, and `h` is the step size. This function computes the ratio $(f(x+h) - f(x-h))/2h$. This ratio is our AV_h at a given value of h.

Let us also assume that we implemented the function `true_error(av, tv)` to compute the E_t formula, where `av` is the approximate value and `tv` is the true value. We can then compute the true value by differentiating the function, implementing the derivative in Python and evaluating it at 1.0.

Example 1

Let us run the code below to compute the values of E_t for various levels of h.

```
h = 0.1
f = lambda x: 5.0*(math.e**(2.5*x))     ## the original function
num_iters = 3                           ## number of iterations
gtf = lambda x: 12.5*(math.e**(2.5*x)) ## ground truth function (i.e., the derivative)
tv = gtf(1.0)                           ## true value
for i in range(num_iters):
    av = cdd_1st_drv_ord2(f, 1.0, h)
    terr = true_error(av, tv)
    print('h={}\tav={}\tEt={}'.format(h, av, terr))
    h /= 2
print(tv)
```

Example 1

The code on the previous slide when run in Python 3.6.7 on Ubuntu 18.04 LTS (it is quite possible that you will see different numbers in different Python distributions on other operating systems) produces the following output:

```
h=0.1     av=153.87240119574116  Et=-1.5912266869477776
h=0.05    av=152.6780499995371   Et=-0.3968754907437244
h=0.025   av=152.38033526380136  Et=-0.09916075500797206
h=0.0125  av=152.30596 Et=-0.024791 #skipped some digits
h=0.00625 av=152.28737 Et=-0.006201 #skipped some digits
```

As we can see, the true error is, indeed, approximately quartered as the step size is halved. Let us compare this approximation with `sympy`, `numpy`, and Python's `math`.

Example 1

We do this example in numpy and math first. See the file sympy_vs_math_exp.py.

```
import math
import numpy as np
mav = 12.5*math.e**(2.5*1.0)
nav = 12.5*np.exp(2.5*1.0)
print('math  av = {}'.format(mav)) ## Python math
print('numpy av = {}'.format(nav)) ## Numpy
```

In Py 3.6.7 on Ubuntu 18.04 LTS, the printed values are:

```
math  av = 152.28117450879338
numpy av = 152.2811745087934
```

Python's math library gives us 1 more precision digit.

Example 1

We move on to `sympy`.

```
import sympy as sp
from sympy           import symbols
from sympy.utilities import lambdify
x = symbols('x')
f = 5.0*sp.exp(2.5*x)
df = f.diff()
ldf = lambdify((x), df)
sav = ldf(1.0)
print('sympy av = {}'.format(sav))
```

In Py 3.6.7 on Ubuntu 18.04 LTS, the printed value is:

```
sympy av = 152.2811745087934
```

The `sympy` system gives us the value identical with `numpy`.

Comparison of Results

Let us compare the results.

```
math  av = 152.28117450879338
numpy av = 152.2811745087934
sympy av = 152.2811745087934
      av = 152.287370
```

The RE/CDD extrapolation (last value) puts us in the ballpark in 5 iterations.

Numerical Instability

Thee two assertions below pass.

```
assert np.allclose(sav, mav) and np.allclose(sav, nav)
assert sav == nav
```

This assertion fails. This is another example of being careful with ==.

```
assert sav == mav or nav == mav
```

Example 2

Let $f(x) = \frac{sin^2\left(\frac{\sqrt{x^2+x}}{cos x - x}\right)}{sin\left(\frac{\sqrt{x}-1}{\sqrt{x^2+1}}\right)}$. Let's use Richardson's exptrapolation with CDD of order 2 to approximate $f'(0.05)$.

First, let us obtain the `sympy` value (See `sympy_vs_math_exp.py`).

```
num2 = sp.sin((sp.sqrt(x**2 + x)) / (sp.cos(x) - x))**2
dnm2 = sp.sin((sp.sqrt(x) - 1) / (sp.sqrt(x**2 + 1)))
f2   =  num2 / dnm2
df2  = f2.diff()
ldf2 = lambdify((x), df2)
sav2 = ldf2(0.05)
print('sav2 = {}'.format(sav2))
```

This code segment prints the following value.

```
sav2 = -2.045361395458189
```

Example 2

Second, let's assume that we have implemented the following functions `rextp_1_cdd_ord2(f, x, h)`, `rextp_2_cdd_ord2(f, x, h)`, and `rextp_3_cdd_ord2(f, x, h)` to do 1st, 2nd, and 3rd REs, respectively.

Third, let us define our function in Python (without sympy).

```
def our_fun(x):
    num = math.sqrt(x**2 + x)/(math.cos(x) - x)
    num = math.sin(num)**2
    denom = (math.sqrt(x) - 1.0)/(math.sqrt(x**2 + 1.0))
    denom = math.sin(denom)
    return num/denom
```

Example 2

Fourth, now we can evaluate the 1st, 2nd, and 3rd REs and compute their true errors at $h = 0.01$.

```
## step size
h  = 0.01
## true value
tv  = -2.04536139545819
## approximate value for 1st richardson
av1 = rextp_1_cdd_ord2(our_fun, 0.05, h)
## approximate value for 2nd richardson
av2 = rextp_2_cdd_ord2(our_fun, 0.05, h)
## approximate value for 3rd richardson
av3  = rextp_3_cdd_ord2(our_fun, 0.05, h)
## true error for 1st richardson
te1 = true_error(av1, tv)
## true error for 2nd richardson
te2 = true_error(av2, tv)
## true error for 3rd richardson
te3 = true_error(av3, tv)
```

Example 2

These are the AVs and true errors I get with Python 3.6.7 on Ubuntu 18.04LTS. Below `rav` is our Richardson's approximation and `sav` is the sympy's value; `et` is the error of our Richardson's approximation.

```
1st Richardson:
rav = -2.04706766337919
et  = 0.0017062679210000375
sav = -2.04536139545819

2nd Richardson:
rav = -2.0453619893051673
et  = 5.938469773703048e-07
sav = -2.04536139545819

3rd Richardson:
rav = -2.045361431919385
et  = 3.6461194952153164e-08
sav = -2.04536139545819
```

The third approximation is adequate. Also, note we did not have to differentiate this function!

References

1. Reading handout with CDD formulas.
2. `www.sympy.org`.