CS 3430: S24: Scientific Computing
Assignment 07
Central Divided Difference, Richardson Extrapolation, Romberg Integration

Vladimir Kulyukin
Department of Computer Science
Utah State University

March 2, 2024

Learning Objectives

1. Central Divided Difference (CDD)
2. Richardson Extraploation (RE)
3. Romberg Integration

Introduction

In this assignment, we will implement several CDD formulas to approximate $f'(x)$ and $f''(x)$ and use these formulas in conjunction with the Richardson Extrapolation to approximate $f'(x)$ and $f''(x)$. Then we will implement the Romberg Integration method to approximate $\int_a^b f(x)dx$.

You will code your solutions to Problem 1 in `cdd.py` and to Problem 2 in `rmb.py`. Included in the zip is `cs3430_s24_hw07_uts.py` with my unit tests for this assignment. Remember to proceed one unit test at a time and to work it out step by step.

Optional Problem 0: (0 points)

You may want to Review your class notes and/or the slides for Lectures 13 and 14 and become comfortable with the Central Divided Difference (CDD), Richardson Extrapolation (RE), the true error, and the absolute relative true error, the true value (TV), the approximate value (AV), the Trapezoidal Rule (T Rule) of the definite integral approximation, the Romberg Lattice, the Romberg Matrix, and the Romberg Integration.

For your convenience, I included in the zip of this assignment the pdf document CDD.pdf. This document contains a segment of Ch. 6 from "Numerical Methods Using MATLAB" 4th edition by J. Mathews and K. Fink. Those of you, who cannot attend in-class sessions in peron and follow it through my class materials in Canvas, have most likely read this PDF, because it was included in the zip of Lecture 12 when we discussed the CDD for the first time.

Problem 1: (2 points)

The module `cdd.py` contains the stubs of four static methods you will implement to approximate $f'(x)$ and $f''(x)$. Open up CDD.pdf and go to Tables 6.3 and 6.4 on p. 339. The formulas in

these tables use slightly different notation from the one we used in class and in the lecture PDFs in Canvas. But, it is easy to convert one to the other and vice versa. In the PDF, f_{-1} refers to $f(x-h)$ and f_1 refers to $f(x+h)$. Analogously, f_{-2} refers to $f(x-2h)$ and f_2 refers to $f(x+2h)$, etc. To generalize, f_{-k} is $f(x-kh)$ and f_k is $f(x+kh)$, or

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

in our notation is the same as the formula

$$f'(x_0) = \frac{f_1 - f_{-1}}{2h}.$$

in CDD.pdf document. Of course, $\approx$ instead of $=$ would be more appropriate in both formulas, but let us stick with $=$ for the sake of simplicity but remember that we are always computing approximations with some level of error.

Now let us start implementing the methods whose stubs are in `cdd.py`. All the four methods need a greater precision. One way to get more precision is to use the C++ long double type, which is available with `np.longdouble`.

A brief note on my naming conventions in the method stubs in `cdd.py`, e.g., `cdd.drv1_ord2(f, x, h)` and `cdd.drv2_ord4(f, x, h)`. The abreviations `drv1` and `drv2` mean that the methods approximate the 1st and 2nd derivatives, respectively. The abbreviations `ord2` and `ord4` refer to the order of the error in each method, i.e., $O(h^2)$ and $O(h^4)$. Thus, the method signature `cdd.drv1_ord2(f, x, h)` means that this method estimates the 1st derivative with the error of $O(h^2)$, i.e., some positive real constant multiple of h^2, and the signature `cdd.drv2_ord4(f, x, h)` means that this method estimates the 2nd derivative with the error of $O(h^4)$.

Those of you who attended Lectures 12, 13, and 14 in person may recall or have it in your notes that I briefly mentioned that these error levels can be estimated with the Taylor series. We did not go deeper into it, because the only mathematical prerequisite for this class is Calculus I. However, those of you who are mathematically included or curious can read the proofs of Theorems 6.1 and 6.2 in CDD.pdf to see how the second- and fourth-degree Taylors can be used to derive these error estimates. Rest assured though that I will not ask you to reproduce these proofs in exams. This is strictly FYI.

Implement the first formula in Table 6.3 as `cdd.drv1_ord2(f, x, h)` and the first formula in Table 6.4 as `cdd.drv1_ord4(f, x, h)`. These two methods take a Python function `f`, a real number `x`, and a real number `h` (i.e., the step size), and approximate $f'(x)$ at the given value of the step size `h`, e.g., $h = 0.01$.

Implement the second formula in Table 6.3 as `cdd.drv2_ord2(f, x, h)` and the second formula in Table 6.4 as `cdd.drv2_ord4(f, x, h)` to approximate $f''(x)$ with the error levels of $O(h^2)$ and $O(h^4)$, respectively.

Let us run the first unit test for Example 6.2, p. 325 in `CDD.pdf`. In this example, the function is $f(x) = cos(x)$ and the authors ask us to estimate $f'(0.8)$ with the first formulas in Tables 6.3 and 6.4 (Mathews and Fink refer to them as formulas (3) and (10)), i.e., the formulas we implemented as `cdd.drv1_ord2()` and `cdd.drv1_ord4()`. Before reading the output of this unit test below, pause and take a look at `test_hw07_example_6_2_cdd_pdf_ut01a`.

Running this unit test produces the following output in Python 3.6.7 on Ubuntu 18.04 LTS (Bionic Beaver). The characteristics are the same, i.e., -0, and the four digits of the mantissas are the same as well, 7173, which is not bad. In the order 4 approximation in `CDD.pdf`, the book's mantissa coincides with the `sympy`'s value mantissa on the first six digits, which is better. If we take a look at our approximations in the last two lines of the output below, we notice that our order 2 approximation coincides with `sympy`'s on the first four digits of the mantissa and our order 4 approximation conincides with `sympy`'s on the first 9 digits of the mantissa, i.e., 717356090.

```
Sympy         TV = -0.7173560908995228
CDD.pdf Ord 2 AV = -0.71734415
CDD.pdf Ord 4 AV = -0.717356108
cdd.drv1_ord2 AV = -0.7173441350244558
cdd.drv1_ord4 AV = -0.7173560906604131
```

So, with our order 4 approximation, we are better than the book, which could be attributed to the fact that `numpy` gives us more precision than the version of MATLAB used in the book, and we are almost as good as `sympy`. After typing the previous sentence, I went to my Python interpreter Linux window and did this.

```
### sympy's tv
>>> spv = -0.7173560908995228
### our order 4 approximation
>>> our_ord4_av = -0.7173560906604131
>>> abs(spv-our_ord4_av)
2.3910973201424213e-10
```

2.3910973201424213e-10 is a small real number. I know, I know, what's the big deal? The big deal is that we did not have to differentiate anything or install `sympy` (or any other differentiation engine, or implement our own) to get this value!

Can we do better? We can give it a try. Take a look at `test_hw07_example_6_2_cdd_pdf_ut01b()`, where we place steps 8 and 9 inside a for loop to iteratively halve the step size h like so.

```
h = 0.01
for k in range(5):
    ## 8. compute approximate values av of orders 2 and 4 at h
    hv = h/(2**k)
    av_ord2 = cdd.drv1_ord2(py_f, 0.8, hv)
    av_ord4 = cdd.drv1_ord4(py_f, 0.8, hv)
    print('Sympy         TV = {}'.format(sp_tv))
    print('cdd.drv1_ord2 AV = {} at h = {}'.format(av_ord2, hv))
    print('cdd.drv1_ord4 AV = {} at h = {}'.format(av_ord4, hv))
```

So, the last three lines printed by the above code segment are as follows.

```
Sympy         TV = -0.7173560908995228
cdd.drv1_ord2 AV = -0.7173560441966487 at h = 0.000625
cdd.drv1_ord4 AV = -0.7173560908995421 at h = 0.000625
```

So, after 5 iterations of the loop, the mantissa of our order 2 approximation conincides with `sympy`'s on the first 7 digits, i.e., 7173560, and the mantissa of our order 4 approximation conincides with `sympy`'s on the first 13 digits, i.e., 7173560908995, which is really good.

Can we do even better? Well, frankly, I would stop here and enjoy the precision of the obtained results. Here is why. There comes a point after which the smaller the value of h, the further away we may get from the the true value due to numerical instability. Here is what happens on my computer when I iterate the for-loop in `test_hw07_example_6_2_cdd_pdf_ut01b()` 100 times. The last three lines print out as follows.

```
Sympy         TV = -0.7173560908995228
cdd.drv1_ord2 AV = 0.0 at h = 1.5777218104420236e-32
cdd.drv1_ord4 AV = -1759218604441600.0 at h = 1.5777218104420236e-32
```

Our order 2 approximation is 0, which is false, and our order 4 approximation is -1,759,218,604,441,600.0. Quite an estimate, but also waaaaaaay off the mark!

The unit tests `test_hw07_example_6_4_cdd_pdf_ut01a` and `test_hw07_example_6_4_cdd_pdf_ut01b` do the same types of tests for Example 6.4, p. 340 in CDD.pdf. In both cases, we first compute an estimate of the ground truth with `sympy` and then use our implementations of `cdd.drv2_ord2` and `cdd.drv2_ord4` to approximate `sympy`'s value. Here's what the second unit tests prints out at the end.

```
Sympy         TV = -0.6967067093471654
cdd.drv2_ord2 AV = -0.6967066866536697 at h = 0.000625
cdd.drv2_ord4 AV = -0.6967067096042 at h = 0.000625
Sympy         TV = -0.6967067093471654
```

Both approximations are pretty close to `sympy`'s value. In fact, the following 2 assertions should pass on your computer.

```
assert abs(sp_tv - av_ord2) < 1.0e-5
assert abs(sp_tv - av_ord4) < 1.0e-7
```

A takeaway is that we again were able to get pretty close to `sympy`'s estimate that does explicit differentiation in just a few iterations without performing any differentiation.

What happens if we add RE? Let us do it and see what happens. The module `rxp.py` implements the formula for the 2nd Richardson on Slide 7 in Lecture 13 and on Slide 10 in Lecture 14. You do not need to change anyting in that file. The unit test `test_hw07_example_6_2_cdd_pdf_ut01c` does what we did in `test_hw07_example_6_2_cdd_pdf_ut01b` with RE added. The last three lines in this test print out the following message.

```
Sympy         TV = -0.7173560908995228
RE ORD 2 AV = -0.7173560908995125 at h = 0.000625
RE ORD 4 AV = -0.7173560908995718 at h = 0.000625
```

So, with just four applications of RE added, the order 2 and order 4 aproximations of the first derivative coincide with the first 13 digits of `sympy`'s mantissa, i.e., 7173560908995, which is awesome! In fact, both assertions below should pass.

```
assert abs(re_ord2 - sp_tv) < 1.0e-12
assert abs(re_ord4 - sp_tv) < 1.0e-12
```

Problem 2: (3 points)

The file `rmb.py` contains the stubs of the static methods to compute elements of the Romberg Lattice $R_{j,j}$ (cf. Slide 20, Lecture 14) in order to approximate $\int_a^b f(x)dx$.

Start with the method `rmb.T_k_1(f, a, b, k)`. The T Rule formula that this method implements is on Slide 08, Lecture 13. The first three unit tests for this problem test Examples 1 and 2 in Lecture 14. This method computes the bottom level of the Romberg Lattice (cf. Slide 20, Lecture 14). The method takes a Python function `f`, two real numbers `a` and `b`, and two positive integers `j` and `l` and returns the `np.longdouble` value of $R_{j,1}$ (i.e., the j-th Romberg at level 1).

Once you have implemented this method, proceed to `rmb.R_j_l(f, a, b, j, l)`. This method computes the value at the node $R_{j,l}$ in the Romberg Lattice. Use the dynamic programming optimization tricks discussed beginning at Slide 24 in Lecture 14. Implementing this formula recursively will be slow for larger values of j and l.

I would like to draw your attention to the following unit test.

```
def test_hw07_prob02_ut08(self):
    def f(t):
        return 2000*math.log(140000.0/(140000.0 - 2100*t), math.e) - 9.8*t
    err = 1.e-5
    tv  = 11061.335535081103 ## this is sympy's value (cf. Slide 17, Lecture 14)
    ## Approximating integral of f(x) on [a=8, b=30].
    for j in range(1, 101):
        av  = rmb.R_j_l(f, 8, 30, j, j)
        if abs(tv - av) < err:
           print('R[{},{}] = {}'.format(j, j, av))
           break
```

We are looking for the smallest value of $R_{j,j}$, for which the Romberg definite integral approximation comes close, i.e., within a small error level, to the true value of the integral in Example 2 of Lecture 14 obtained with `sympy`. On my laptop, this unit test prints out the following message.

```
R[5,5] = 11061.335535249562
```

What this means is that only 5 levels (14 nodes) of the Romberg Lattice were needed to approximate fairly closely the definite integral of a rather involved physical function without any explicit differentiation.

What To Submit

Submit your code in `cdd.py` and `rmb.py`.

Happy Hacking and Enjoy Romberg!