CS 3430: S19: SciComp with Py
Assignment 8
Definite Integral Approximation and Image Blurring

Vladimir Kulyukin
Department of Computer Science
Utah State University

March 2, 2019

Learning Objectives

1. Riemann Sums
2. Midpoint, Trapezoidal, and Simpson Rules for Definite Integral Approximation
3. Image Blurring

Problem 1: Riemann Sums (2 points)

Problem 1.1

Implement the function `riemann_approx(fexpr, a, b, n, pp=0)` in the file `riemann.py` that takes a function expression `fexpr`, three constants `a`, `b`, and `n`, and the keyword argument `pp` defaulting to 0. The constants `a` and `b` are the lower and upper bounds of the interval over which the function represented by `fexpr` is integrated. The positive constant `n` specifies the number of subintervals in a partition used to approximate the integral. When the value of `pp` (stands for `partition point`) is 0, `riemann_approx` returns the middle point approximation of the integral. When the value of `pp` is 1, `riemann_approx` returns the right point approximation. When the value of `pp` is -1, the left point approximation is returned.

Use your implementation of `riemann_approx` to implement the function `riemann_approx_with_gt(fexpr, a, b, gt, n_upper, pp=0)` in which the parameters `fexpr`, `a`, `b`, and `pp` are the same as in `riemann_approx`. The

constant `n_upper` specifies the upper bound on the number of the subintervals in a partition. Let $f(x)$ be the function represented by `fexpr`. Then, the constant `gt` is the ground truth value, i.e., $gt = \int_a^b f(x)dx$. The function computes the appropriate riemann sum approximation, determined by the value of `pp`, for each number of subintervals n such that $n \in [1, \texttt{n\_upper}]$. Let r_n be the value of the riemann sum approximation for a given value of n. The function returns a list of 2-tuples where the first element is a value of n and the second is the corresponding error $|gt - r_n|$.

Save your code in `riemann.py`.

Test 01

Let's approximate $\int_{-1}^{1}(3x^2+e^x)dx$ with the midpoint riemann sum on a partition of 10 subintervals.

```
def test_01():
    print('\n***** Test 01 *****')
    fex = make_prod(make_const(3.0), make_pwr('x', 2.0))
    fex = make_plus(fex, make_e_expr(make_pwr('x', 1.0)))
    print(fex)
    err_list = riemann_approx_with_gt(fex,
                                      make_const(-1.0),
                                      make_const(1.0),
                                      make_const(4.35),
                                      make_const(10),
                                      pp=0)
    for n, err in err_list:
        print(n, err)
    print('Test 01: pass')
```

Here is the output of `test_01` in the Py shell.

```
***** Test 01 ***********
((3.0*(x^2.0))+(2.71828182846^(x^1.0)))
(1, 2.3499999999999996)
(2, 0.5947480695872382)
(3, 0.26478811549737724)
(4, 0.14890361544358122)
(5, 0.0951941454853591)
(6, 0.06599949969924701)
(7, 0.04838951138028058)
(8, 0.036957312835646405)
(9, 0.02911823294982696)
(10, 0.023510384611694413)
Test 01: pass
```

Test 02

Let's approximate $\int_{-1}^{1}(3x^2+e^x)dx$ with the left point riemann sum on a partion of 10 subintervals.

```
def test_02():
    print('\n***** Test 02 *****')
    fex = make_prod(make_const(3.0), make_pwr('x', 2.0))
    fex = make_plus(fex, make_e_expr(make_pwr('x', 1.0)))
    print(fex)
    err_list = riemann_approx_with_gt(fex,
                                      make_const(-1.0),
                                      make_const(1.0),
                                      make_const(4.35),
                                      make_const(10),
                                      pp=-1)
    for n, err in err_list:
        print(n, err)
    print('Test 02: pass')
```

Here is the output of `test_02` in the Py shell.

```
***** Test 02 ***********
((3.0*(x^2.0))+(2.71828182846^(x^1.0)))
(1, 2.385758882342885)
(2, 0.017879441171443133)
(3, 0.25220677100134203)
(4, 0.28843431420789756)
(5, 0.27842264444234743)
(6, 0.2584974432493592)
(7, 0.23776930910274885)
(8, 0.2186689648257394)
(9, 0.2017062311704798)
(10, 0.1868083949638537)
Test 02: pass
```

Test 03

Let's approximate $\int_{-1}^{1}(3x^2 + e^x)dx$ with the right point riemann sum on a partition of 10 subintervals.

```
def test_03(self):
    print('\n***** Test 03 *****')
    fex = make_prod(make_const(3.0), make_pwr('x', 2.0))
    fex = make_plus(fex, make_e_expr(make_pwr('x', 1.0)))
```

```
    print(fex)
    err_list = riemann_approx_with_gt(fex, make_const(-1.0),
                                      make_const(1.0),
                                      make_const(4.35),
                                      make_const(10),
                                      pp=+1)
    for n, err in err_list:
        print(n, err)
    print('Test 03: pass')
```

Here is the output of `test_03` in the Py shell.

```
***** Test 03 ***********
((3.0*(x^2.0))+(2.71828182846^(x^1.0)))
(1, 7.08656365691809)
(2, 2.368281828459045)
(3, 1.3147281538570592)
(4, 0.8867668794359034)
(5, 0.6617383104726935)
(6, 0.5249700191798405)
(7, 0.43377423012228)
(8, 0.3689316319961611)
(9, 0.32060541044898727)
(10, 0.2832720824936672)
Test 03: pass
```

Test 04

Let's test `riemann_approx` by approximating $\int_1^2 ln(x)dx$ with the middle point riemann sum on a partition of 100 subintervals.

```
def test_04():
    print('\n***** Test 04 *****')
    fex = make_ln(make_pwr('x', 1.0))
    print(fex)
    err = 0.0001
    approx = riemann_approx(fex,
                            make_const(1.0),
                            make_const(2.0),
                            make_const(100),
                            pp=0)
    assert abs(approx.get_val() - 0.386296444432) <= err
    print('Test 04: pass')
```

Here is the output of `test_04` in the Py shell.

```
***** Test 04 ***********
ln(x^1.0)
0.386296444432
Test 04: pass
```

Problem 1.2

Implement the function `plot_riemann_error(fexpr, a, b, gt, n_upper)` that takes the same arguments as `riemann_approx_with_gt`, but no `pp`, and plots the numbers of subintervals against the middle point, left point, and right point riemann sum approximations. This function makes 3 calls `riemann_approx_with_gt` to obtain the error lists for the middle point, left point, and right point riemann sum approximations and then uses them to plot the errors. The title of the plot should be "Riemann Approximation Error." The midpoint error line should be red, the left point – green, and the right point – blue.

Figure 1 shows the plot generated by the following code that plots the approximation error lines for $\int_{-1}^{1}(3x^2 + e^x)dx$ over partitions of up to 10 subintervals.

```
fex = make_prod(make_const(3.0), make_pwr('x', 2.0))
fex = make_plus(fex, make_e_expr(make_pwr('x', 1.0)))
plot_riemann_error(fex, make_const(-1.0),
                   make_const(1.0),
                   make_const(4.35),
                   make_const(10))
```

Figure 2 shows the plot generated by the following code that plots the approximation error lines for $\int_{-1}^{1}(3x^2 + e^x)dx$ over partitions of up to 50 subintervals.

```
fex = make_prod(make_const(3.0), make_pwr('x', 2.0))
fex = make_plus(fex, make_e_expr(make_pwr('x', 1.0)))
plot_riemann_error(fex, make_const(-1.0),
                   make_const(1.0),
                   make_const(4.35),
                   make_const(50))
```

Note that as the number of subintervals increases, the errors in all three approximations start to converge to 0, which verifies the theorem on slide 30 of Lecture 14.

Problem 2: Midpoint, Trapezoidal, and Simpson Rules (2 points)

Extend your antidifferentiation engine in `antideriv.py` that you implemented in Assignment 07 with the function `antiderivdef(fexpr, a, b)` in which

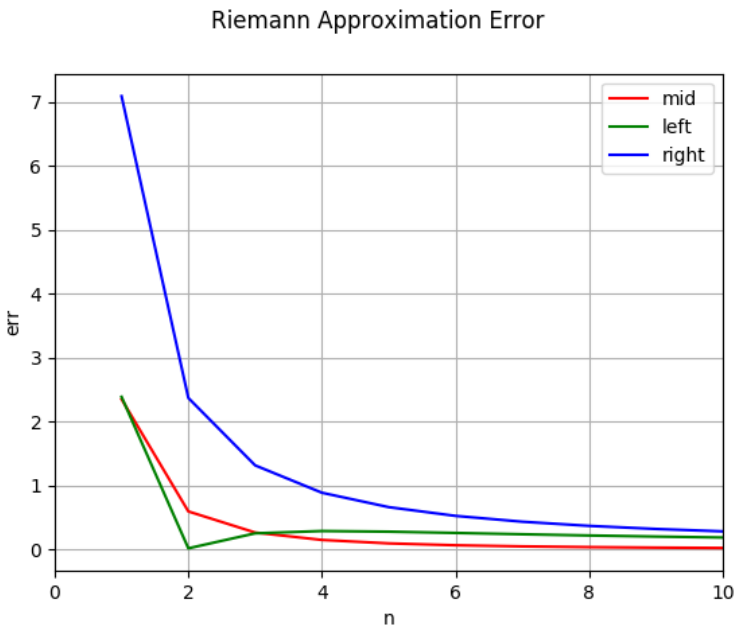

Figure 1: Riemann sum error approximation where $n \in [1, 10]$.

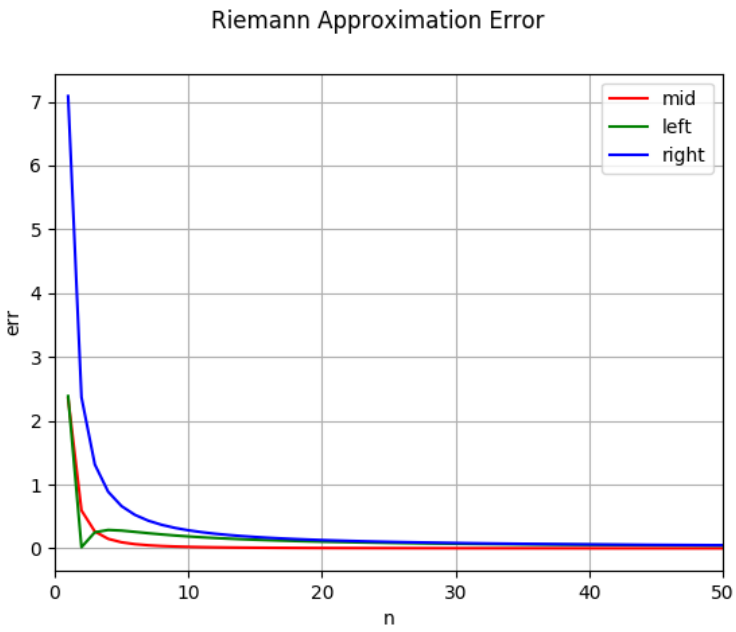

Figure 2: Riemann sum error approximation where $n \in [1, 50]$.

`fexpr` is a function expression and `a` and `b` are constants. Let $f(x)$ be the function represented by `fexpr`. The function `antiderivdef` returns a constant object whose value is $\int_a^b f(x)dx$.

You should use your implementation of `antideriv` from Assignment 07 to implement `antiderivdef`. With a properly working `antideriv` your implementation of `antiderivdef` should be at most 10 lines of code (with assertions).

Save your implementation in `antideriv.py`. We'll use `antiderivdef` in the unit tests for the midpoint, trapezoidal, and simpson approximation rules below.

Implement the functions `midpoint_rule(fexpr, a, b, n)`, `trapezoidal_rule(fexpr, a, b, n)`, and `simpson_rule(fexpr, a, b, n)` that compute the midpoint, trapezoidal, and simpson rule approximations to $\int_a^b f(x)dx$, where $f(x)$ is represented by `fexpr`.

Save your implementations of `midpoint_rule(fexpr, a, b, n)`, `trapezoidal_rule(fexpr, a, b, n)`, and `simpson_rule(fexpr, a, b, n)` in `defintegralapprox.py`.

Test 05

Let's approximate $\int_0^4 (x^2 + 5)dx$ with the midpoint rule on a partition of 250 subintervals and compare the returned value with the one returned by `antiderivdef`.

```
def test_05():
    print('\n***** Test 05 *****')
    fexpr = make_plus(make_pwr('x', 2.0),
                      make_const(5.0))
    a, b, n = make_const(0.0), make_const(4.0), make_const(250)
    approx = midpoint_rule(fexpr, a, b, n)
    print(approx)
    err = 0.0001
    iv = antiderivdef(fexpr, a, b)
    print(iv)
    assert abs(approx.get_val() - iv.get_val()) <= err
    print('Test 05: pass')
```

Here is the output of `test_05` in the Py shell.

```
***** Test 05 ***********
41.333248
41.3333333333
Test 05: pass
```

Test 06

Let's approximate $\int_0^4 (x^2 + 5)dx$ with the trapezoidal rule on a partition of 350 subintervals and compare the returned value with the one returned by `antiderivdef`.

```
def test_06():
    print('\n***** Test 06 *****')
    fex = make_plus(make_pwr('x', 2.0), make_const(5.0))
    a, b, n = make_const(0.0), make_const(4.0), make_const(350)
    approx = trapezoidal_rule(fex, a, b, n)
    print(approx)
    err = 0.0001
    iv = antiderivdef(fex, a, b)
    print(iv)
    assert abs(approx.get_val() - iv.get_val()) <= err
    print('Test 06: pass')
```

Here is the output of `test_06` in the Py shell.

```
***** Test 06 ***********
41.3334204082
41.3333333333
Test 06: pass
```

Test 07

Let's approximate $\int_0^4 (x^2+5)dx$ with the simpson rule on a partition of 10 subintervals and compare the returned value with the one returned by `antiderivdef`.

```
def test_07():
    print('\n***** Test 07 *****')
    fex = make_plus(make_pwr('x', 2.0), make_const(5.0))
    a, b, n = make_const(0.0), make_const(4.0), make_const(10)
    approx = simpson_rule(fex, a, b, n)
    err = 0.0001
    iv = antiderivdef(fex, a, b)
    assert abs(approx.get_val() - iv.get_val()) <= err
    print('Test 07: pass')
```

Here is the output of `test_07` in the Py shell.

```
***** Test 07 ***********
41.3333333333
41.3333333333
Test 07: pass
```

It is not always possible to evaluate definite integrals that arise in practical problems by computing antiderivatives. Mathematicians keep compiling ever larger tables of antiderivatives; software engineers working on scientific computing systems keep integrating these rules into ever more complex differentiation and integration engines. However, in many practical situations it may not be possible to reduce a complex antiderivative to a set of elementary antiderivatives. Moreover, sometimes the function we want to integrate is simply unknown. So, let's evaluate the simpson rule on the integrals which our antidifferentiation engine cannot currently handle.

Test 08

Let's approximate $\int_0^2 2xe^{x^2}dx$ with the simpson rule on a partition of 100 subintervals.

```
def test_08():
    print('\n***** Test 08 *****')
    fex = make_prod(make_prod(make_const(2.0),
                              make_pwr('x', 1.0)),
                    make_e_expr(make_pwr('x', 2.0)))
    a, b, n = make_const(0.0), make_const(2.0), make_const(100)
    approx = simpson_rule(fex, a, b, n)
    print(approx)
    err = 0.0001
    assert abs(approx.get_val() - 53.5981514272) <= err
    print('Test 08: pass')
```

Here is the output of `test_08` in the Py shell.

```
***** Test 08 ***********
53.5981514272
Test 08: pass
```

Test 09

Let's approximate $\int_0^2 \sqrt{1+x^3}dx$ with the simpson rule on a partition of 100 subintervals.

```
def test_09():
    print('\n***** Test 09 *****')
    fex = make_plus(make_const(1.0),
                    make_pwr('x', 3.0))
    fex = make_pwr_expr(fex, 0.5)
    a, b, n = make_const(0.0), make_const(2.0), make_const(100)
```

```
    approx = simpson_rule(fex, a, b, n)
    print(approx)
    err = 0.0001
    assert abs(approx.get_val() - 3.24124) <= err
    print('Test 09: pass')
```

Here is the output of `test_09` in the Py shell.

```
***** Test 09 ***********
3.24130926301
Test 09: pass
```

Problem 3: Image Blurring (1 point)

Let's continue our journey into image processing. Implement the function `amplify_grayscale_blur_img_dir(ftype, in_img_dir, kz, c, amount)`. The parameter `ftype` is a string that specifies a file extension (e.g., `'.png'` or `'.jpg'`). The parameter `in_img_dir` is a string specifying a directory with images of the specified type. The parameter `kz` is an odd positive integer specifying the size of a mean blur filter. The parameter `c` is a string specifying the channel (i.e., `'b'`, `'g'`, or `'c'`). The parameter `amount` is a non-negative integer specifying the amount of amplification on the specified channel.

This function reads all images with the specified extension, amplifies each image on the specified channel by the specified amount, grayscales the amplified image, blurs the image with the square mean filter of the specified size, and saves the resulting image in the same directory in the file that has the same file name as the original image's file except that the suffix `'_blur'` is added to the end of the original file name. Both files have the same file extension.

Suppose the directory `/home/vladimir/images/` contains the following 8 images:

1. `/home/vladimir/images/output11844.jpg`;
2. `/home/vladimir/images/output11849.jpg`;
3. `/home/vladimir/images/output11842.jpg`;
4. `/home/vladimir/images/output11848.jpg`;
5. `/home/vladimir/images/output11948.jpg`;
6. `/home/vladimir/images/output11907.jpg`;
7. `/home/vladimir/images/output11843.jpg`;
8. `/home/vladimir/images/output11884.jpg`.

Let's run `amplify_grayscale_blur_img_dir` on this directory to amplify the green channel of each image by 100, grayscale it, and then blur with a 15x15 mean filter.

```
>>> amplify_grayscale_blur_img_dir('.jpg',
                                   '/home/vladimir/images/',
                                   15, 'g', 100)
```

After executing the above call, the directory will have 8 new images:

1. `/home/vladimir/images/output11844_blur.jpg`;
2. `/home/vladimir/images/output11849_blur.jpg`;
3. `/home/vladimir/images/output11842_blur.jpg`;
4. `/home/vladimir/images/output11848_blur.jpg`;
5. `/home/vladimir/images/output11948_blur.jpg`;
6. `/home/vladimir/images/output11907_blur.jpg`;
7. `/home/vladimir/images/output11843_blur.jpg`;
8. `/home/vladimir/images/output11884_blur.jpg`.

Save your coding solution to Problem 3 in `hw08_s19.py`. You can use `cv2.imwrite(file_path, img)` to save an image `img` in a file specified by `file_path`. Experiment with different filters and amplification amounts to achieve and observe various blurring effects.

What to Submit

You have to submit `riemann.py`, `antideriv.py`, `defintegralapprox.py`, and `hw08_s19.py`. You should also submit all the files needed to run your code. Zip your entire working directory in `hw08.zip` and submit it via Canvas.

And here comes my perpetual refrain! Do not change the names of the files that were given to you or the names of the functions you are asked to implement. The unit tests that I write for the grader every week depend on these names remaining the same.

Happy Hacking!