CS 3430: S19: SciComp with Py
Exam 2

Vladimir Kulyukin
Department of Computer Science
Utah State University

March 26, 2019

Introduction

1. This exam has 5 problems worth a total of 15 points; it is open books, open notes in that it is perfectly fine for you to look at the lecture slides or the reading handouts; it is also OK to consult online documentation at `www.python.org`; it is **not OK** to go to `stackoverflow` or any other technical forum and copy and paste someone else's code – this is plagiarism and, if discovered, will be treated as such; you have to do your own work.

2. You must use your source code from the assignments (e.g., `antideriv.py`, `defintegralapprox.py`, `tof.py`, etc.); you may not use any third-party libraries; the imports of standard Python libraries (e.g., `math`, `numpy`, `matplotlib`, `scipy`) and `cv2` are fine.

3. You may not talk to anyone during this exam orally, digitally, or in writing.

4. You have 1 hour and 15 minutes to complete this exam; the exam is due on Canvas at 4:20pm today (03/26/19), which means that **the submission will close at 4:19:59pm**. I will use the formula $p(t) = 2^t$ to deduct points for late submissions, where t is the number of minutes past 4:20pm. (2 points for 1 minute, 4 - for 2 minutes, etc.) This is a firm deadline. You are always better off submitting something on time for partial credit.

5. The zip for this exam contains three folders: `code`, `img`, and `csv`.

6. The file `code/cs3430_s19_exam_02.py` is the file where you will write and save all your solutions. It has some starter code for you. You must also submit all the files needed to run your functions in `cs3430_s19_exam_02.py` by saving them in the `code` directory. The safest thing for you to do is to zip your whole directory into `exam02.zip` and upload it to Canvas.

7. Write your name and `A#` in the header of `cs3430_s19_exam_02.py`. It makes it easier for me to grade your submissions.

8. Do not change the names of the functions in `cs3430_s19_exam_02.py`.

9. Relax, concentrate, and do your best! I wish you all best of luck and, as always, Happy Hacking!

Problem 1: (3 points)

Write the function `test_antideriv(fexpr, gt, lwr, uppr, err)` that takes a function representation `fexpr`, computes its antiderivative with **your** implementation of `antideriv`, converts it to a Python function with **your** implementation of `tof`, and compares it with the ground truth function `gt` on the interval `[lwr, uppr]` with the assertion testing whether `|adrvf(x) - gt(x)|` $\leq$ `err`, where `adrvf` is the Python function of the antiderivative of `fexpr` returned by `tof()` and $x \in$ `[lwr, uppr]`. The parameters `lwr`, `uppr`, and `err` are constant objects.

The function prints the antiderivative and the values of `adrvf(x)` and `gt(x)` for each $x \in$ `[lwr, uppr]`. Note that this function assumes that your `antideriv` function returns a function representation of the antiderivative with a zero constant.

I encoded 10 function expressions and wrote 10 corresponding ground truth functions in `cs3430_s19_exam_02.py`. I also included in the `code` directory all my maker files I used to encode the function expressions. You can use these to test your implementation of `test_antideriv()`.

Let's do $\int e^{-2x} dx$.

```
fexpr_02 = make_e_expr(make_prod(make_const(-2.0),
                                 make_pwr('x', 1.0)))
gt_02 = lambda x: -0.5*(math.e**(-2*x))

>>> test_antideriv(fexpr_02, gt_02, make_const(0), make_const(10),
                   make_const(0.0001))
(((1.0/-2.0)*(2.71828182846^(-2.0*(x^1.0))))+0.0)
(-0.5, -0.5)
(-0.06766764161830635, -0.06766764161830635)
(-0.009157819444367093, -0.009157819444367093)
(-0.0012393760883331797, -0.0012393760883331797)
(-0.00016773131395125598, -0.00016773131395125598)
(-2.2699964881242437e-05, -2.2699964881242437e-05)
(-3.072106176664107e-06, -3.072106176664107e-06)
(-4.1576435955178426e-07, -4.1576435955178426e-07)
(-5.6267587359629605e-08, -5.6267587359629605e-08)
```

```
(-7.614989872356321e-09, -7.614989872356321e-09)
(-1.03057681121928e-09, -1.03057681121928e-09)
```

Save your implementation of `test_antideriv` in `cs3430_s19_exam_02.py`.

Problem 2: (3 points)

Write the function `taylor(fexpr, a, n)` that takes a function representation `fexpr` and two constants, `a` and `b`, and returns a Python function that approximates the function represented by `fexpr` with the n-th Taylor polynomial at $x = a$.

Write the function `test_taylor(fexpr, x, n, err, gt)` that takes a function representation `fexpr` and checks if the absolute difference between the value of n-th Taylor polynomial (computed by `taylor()` defined above) at `x` and the ground truth function `gt` at `x` does not exceed the value of `err`. The function also prints out the value of the Taylor polynomial and the ground truth function at `x`.

Use your implementation to approximate with the 2nd and 3rd Taylor polynomials and the error of 0.0001 the following functions: 1) $f(x) = xe^x$ at $x = 2.001$; 2) $f(x) = x^4 + x + 1$ at $x = 5.03$; 3) $f(x) = (5 - x)^{-1}$ at $x = 4.002$.

I encoded these functions with the function expressions `fexpr2_01`, `fexpr2_02`, and `fexpr2_03` and wrote the corresponding ground truth functions in `cs3430_s19_exam_02.py`. Use them to approximate the above three functions.

Let's approximate $f(x) = xe^x$ at 2.001 with the 2nd and 3rd Taylors. The expression that encodes this function in `cs3430_s19_exam_02.py` is `fexpr2_01`.

```
fexpr2_01 = make_prod(make_pwr('x', 1.0),
                      make_e_expr(make_pwr('x', 1.0)))
```

The two ground truth functions for this function expression are given below. The first one is for the 2nd Taylor, the second – for the 3rd.

```
def gt21_02(x):
    ''' ground truth for 2nd taylor of fexpr2_01. '''
    f0 = tof(fexpr2_01)
    f1 = tof(deriv(fexpr2_01))
    f2 = tof(deriv(deriv(fexpr2_01)))
    return f0(2.0) + f1(2.0)*(x - 2.0) + \
           (f2(2.0)/2)*(x - 2.0)**2

def gt21_03(x):
    ''' ground truth for 3rd taylor for fexpr2_01. '''
```

```
    f0 = tof(fexpr2_01)
    f1 = tof(deriv(fexpr2_01))
    f2 = tof(deriv(deriv(fexpr2_01)))
    f3 = tof(deriv(deriv(deriv(fexpr2_01))))
    return f0(2.0) + f1(2.0)*(x-2.0) + (f2(2.0)/2)*(x - 2.0)**2 + \
           (f3(2.0)/6)*(x - 2.0)**3
```

```
>>> test_taylor(fexpr2_01, make_const(2.001), make_const(2),
                make_const(0.0001), gt21_02)
(14.800294144270286, 14.800294144270286)
```

Let's quickly check how close we are.

```
>>> import math
>>> 2.001*math.e**2.001
14.800294150429682
>>> 14.800294144270286 - 14.800294150429682
-6.159396548355289e-09
```

Not bad! But we can get even closer with the third Taylor.

```
>>> test_taylor(fexpr2_01, make_const(2.001), make_const(3),
                make_const(0.0001), gt21_03)
(14.800294150427833, 14.800294150427833)
>>> 14.800294150427833 - 14.800294150429682
-1.8491874698156607e-12
```

Save your implementations of `taylor` and `test_taylor` in `cs3430_s19_exam_02.py`.

Problem 3: (3 points)

The directory `exam02/img/` contains 98 PNG images. Use `read_img_dir` given in `cs3430_s19_exam_02.py` (or your own version from Assignment 7) to read the images from the directory into an array of 2-tuples of file paths and numpy matrices.

```
>>> il = read_img_dir('.png', 'img/')
>>> len(il)
98
```

Implement `top_n_std(imglist, n, c='B')` that takes an array of 2-tuples of file paths and numpy matrices, `imglist`, returned by `read_img_dir`, a non-negative integer `n`, and the keyword argument `c` that specifies a channel (`'B'` – for black, `'G'` – for green, `'R'` – for red).

The function iterates through `imglist`, computes the mean and sample standard deviation of the specified channel for each image, and returns a list of `n` 3-tuples

specifying the images with the top `n` largest sample standard deviations in the specified channel and their means. Each 3-tuple consists of a file path, the mean pixel value, and the sample standard deviation of the pixel values in the specified channel of the image in the file path. You can use `numpy.mean` and `numpy.std` to compute the mean and sample standard deviation of numpy arrays. Here is a test run.

```
>>> top_5B = top_n_std(il, 5, c='B')
>>> len(top_5B)
5
>>> for ip, mean, std in top_5B:
        print(ip, mean, std)
('img/1b_bee_10.png', 160.12475555555557, 81.66040227197152)
('img/1b_bee_09.png', 129.31911111111111, 79.71371471530297)
('img/2b_bee_17.png', 184.12197530864196, 76.607429761434048)
('img/1b_bee_01.png', 148.69675555555557, 71.53538464732361)
('img/2b_nb_10.png', 137.45679012345678, 70.796557799020491)
```

Write the function `blur_img_list(imglist, kz)` that takes an image list `imglist` returned by `read_img_dir` and returns another image list where the second element in each 2-tuple is the numpy matrix of the original image blurred with the `kz x kz` mean kernel. Here is a test run.

```
>>> il = read_img_dir('.png', 'img/')
>>> bil = blur_img_list(il, 5)
>>> top_5G = top_n_std(bil, 10, c='G')
>>> for ip, mean, std in top_5G:
        print(ip, mean, std)
('img/1b_bee_09.png', 141.77195555555556, 73.538455792700262)
('img/1b_bee_10.png', 169.36004444444444, 72.666016703496609)
('img/2b_bee_17.png', 181.93555555555557, 69.534314631148902)
('img/1b_nb_14.png', 108.11259259259259, 69.245869249679203)
('img/2b_nb_08.png', 110.04876543209876, 67.373686983107731)
('img/1b_bee_01.png', 163.30488888888888, 67.247952715426919)
('img/1b_nb_08.png', 156.28456790123457, 65.094670267011622)
('img/2b_bee_02.png', 134.81530864197532, 64.914208869216267)
('img/1b_bee_11.png', 176.94759999999999, 62.233670861498155)
('img/2b_nb_10.png', 153.29185185185185, 61.908320011972592)
```

Save your implementations of `top_n_std` and `blur_img_list` in `cs3430_s19_exam_02.py`.

Problem 4: (4 points)

Implement the function `fit_regression_line(x, y)` where `x` and `y` are 1D numpy arrays of the same length. The function returns a Python function of

one argument that computes the value of its argument on the least squares line fit to the `x`, `y` data. Here is a quick test.

```
>>> import numpy as np
>>> x = np.array([1, 2, 3, 4, 5])
>>> y = np.array([2, 3, 4, 5, 6])
>>> rlf = fit_regression_line(x, y)
>>> rlf(1)
2.0
>>> rlf(2)
3.0
>>> rlf(3)
4.0
>>> rlf(7)
8.0
>>> rlf(10)
11.0
>>> rlf(101)
102.0
```

The directory `exam02/csv` contains a few csv files from Assignment 9. Implement the function `analyze_bee_traffic_data(csv_fp, d='u')` that takes a path to the csv file with bee traffic data and the keyward argument `d` specifying traffic direction (`'u'` – for up, `'d'` – for down, `'l'` – for lateral).

This function reads in the file with `read_bee_traffic_csv_file` (this function is defined in `cs3430_s19_exam_02.py`), fits the 1st-, 2nd-, 3rd-, and 10th-degree polynomials to the appropriate arrays (the first array is always the time array with seconds; the second is the corresponding array of upward, downward, or lateral estimates of bee moves, depending on the value of `d`), generates a figure that contains the scatter plot of the data in pale blue (this is the default color), the 1st-degree polynomial curve (the one returned by your `fit_regression_line(x, y)`), the 2nd-, 3rd-, 10th-degree poly curves, and outputs the error report that prints the error of the fit (the function `error` in `cs3430_s19_exam_02.py` computes the error of the fit) for each curve fit to the data. You can use `scipy.polyfit` function to fit the 2nd-, 3rd, and 10th-degree polynomials.

Let's analyze upward bee traffic in `192_168_4_5-2018-07-01_08-00-10.csv`.

```
>>> csv_fp_01 = '192_168_4_5-2018-07-01_08-00-10.csv'
>>> analyze_bee_traffic_data(csv_fp_01, d='u')
Upward Bee Traffic Data Report:
least squares error: 1.22702998091
sp.polyfit 2 error:  0.994272306456
sp.polyfit 3 error:  0.990574466368
sp.polyfit 10 error: 0.692368644894
```

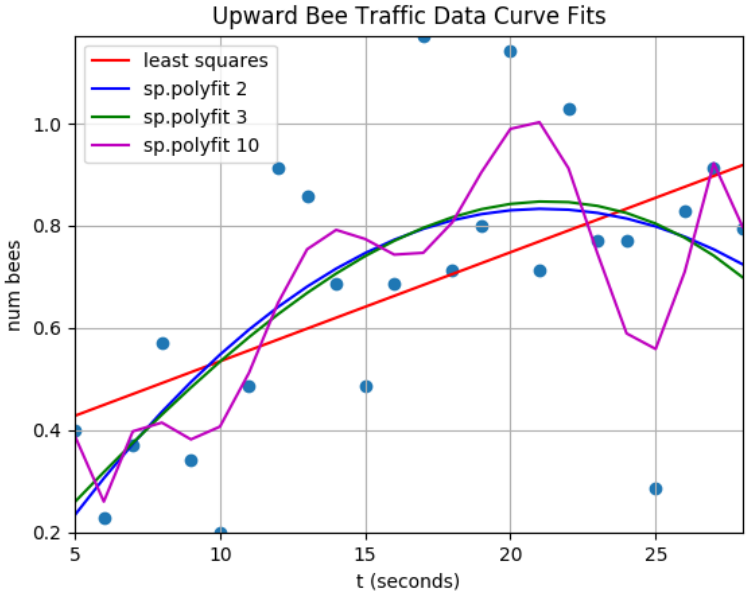

Figure 1: Upward Bee Traffic.

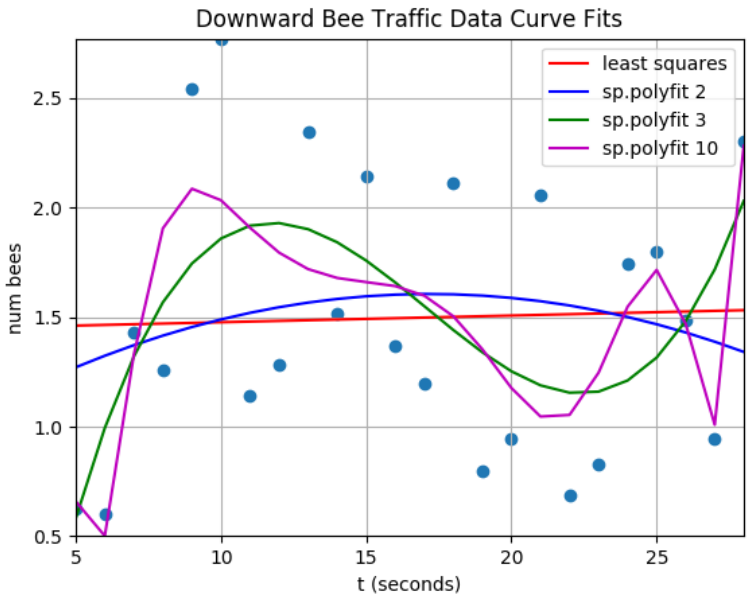

Figure 2: Downward Bee Traffic.

The graph generated by this call is shown in Fig. 1.

Let's analyze downward bee traffic in `192_168_4_5-2018-07-01_08-00-10.csv`.

```
>>> csv_fp_01 = '192_168_4_5-2018-07-01_08-00-10.csv'
>>> analyze_bee_traffic_data(csv_fp_01, d='d')
Downward Bee Traffic Data Report:
least squares error: 9.43468027068
sp.polyfit 2 error:  9.2110371501
sp.polyfit 3 error:  6.52138839221
sp.polyfit 10 error: 5.03400458683
```

The graph generated by this call is shown in Fig. 2.

Let's analyze lateral bee traffic in `192_168_4_5-2018-07-01_08-00-10.csv`.

```
>>> csv_fp_01 = '192_168_4_5-2018-07-01_08-00-10.csv'
>>> analyze_bee_traffic_data(csv_fp_01, d='l')
Lateral Bee Traffic Data Report:
least squares error: 3.52368055327
sp.polyfit 2 error:  3.4497548972
```

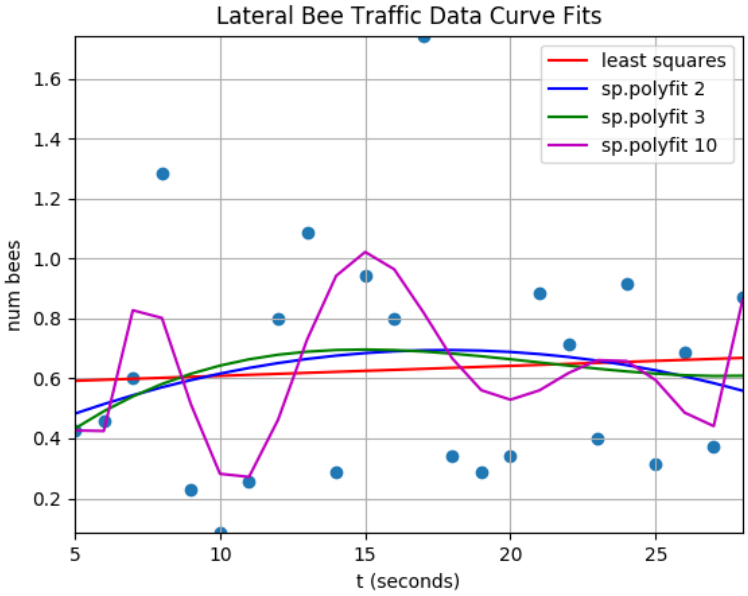

Figure 3: Lateral Bee Traffic.

```
sp.polyfit 3 error:  3.43564839326
sp.polyfit 10 error: 2.54908883542
```

The graph generated by this call is shown in Fig. 3.

Save your implementations of `fit_regression_line` and `analyze_bee_traffic_data` in `cs3430_s19_exam_02.py`.

Problem 5: (2 points)

Implement the function `bell_curve_iq(a, b, r='m', n=2)` that uses the Bell Curve IQ model to estimate the percentage of the population whose IQs are in `[a, b]`. The arguments `a` and `b` are floats. The keyword argument `r` specifies the integral approximation rule used by the function (`'m'` – for midpoint, `'s'` – for Simpson's, `'t'` – for trapezoidal). The keyword argument `n` specifies the number of intervals used in the partition. The function returns a percentage, i.e, a float in $[0, 100]$.

Here are a few tests.

```
>>> bell_curve_iq(120, 126, r='s', n=6)
5.356847165747042
>>> bell_curve_iq(120, 126, r='m', n=6)
5.355950711154588
>>> bell_curve_iq(120, 126, r='t', n=6)
5.358646430197745
>>> bell_curve_iq(5, 10, r='s', n=100)
7.828393510585502e-07
>>> bell_curve_iq(5, 10, r='m', n=100)
7.828291840574067e-07
>>> bell_curve_iq(5, 10, r='t', n=100)
7.828596840227379e-07
>>> bell_curve_iq(20, 120, r='s', n=200)
89.43499414198169
>>> bell_curve_iq(20, 120, r='m', n=200)
89.43592303771317
>>> bell_curve_iq(20, 120, r='t', n=200)
89.43313586163396
```

Save your implementations of `bell_curve_iq` in `cs3430_s19_exam_02.py`.

What to Submit

You must submit `cs3430_s19_exam_02.py` and the files needed to run your functions in `cs3430_s19_exam_02.py` by placing them in the `exam02/code/` folder. Zip your whole directory into `exam02.zip` and upload it to Canvas.