CS 3430: S24: Scientific Computing
Takehome Exam 03

Vladimir Kulyukin
Department of Computer Science
Utah State University

April 24, 2024

Instructions

1. This takehome exam has 10 problems worth a total of 20 points. You may use your class notes, my lecture PDFs and code samples in Canvas. You may use your own homework solutions. You may not use any other materials (digital or paper).

2. Save your solutions in `cs3430_s24_takehome03.py` and submit this file (and, if necessary, other files – see item 7 below) **in Canvas by 11:59pm on April 28, 2024**.

3. Write your name and A-number in `cs3430_s24_takehome03.py`.

4. You may not talk to anyone when you are working on this exam orally, digitally, in writing, or telepathically.

5. You may use your interactive Python IDE, including the Python documentation that comes with it.

6. As stated above, you may use your own solutions to the previous assignments. For example, you can do imports from your previous solutions as

```
from cs3430_s24_hw09 import *
from cs3430_s24_hw10 import *
from cs3430_s24_hw11 import *
from rmb import rmb
```

   and then use your implementation of functions and class methods from these modules to solve the midterm problems.

7. **Remember to include in your submission zip all the Python files you import from.** E.g., if you import from your `cs_3430_s24_hw09.py`, `cs_3430_s24_hw10.py`, `cs_3430_s24_hw11.py`, or `rmb.py`, include these files in the zip. When I run unit tests on your submission, I will put all your files into the same working directory with your `cs3430_s24_takehome03.py`. If something is not included, the unit tests may fail, and I will have to deduct points. There may be no time for back-and-forth Canvas messaging.

8. You may not use any third party libraries in this exam, except the ones we have used in this class, e.g., `numpy`,`decimal`, `sympy`, `PIL`, `matplotlib`, `scipy`, etc. You may use **only** your own solutions to previous/current assignments. My unit tests are in `cs3430_s24_takehome03_uts.py` to test your solutions. Type your answers to conceptual problems as multiline comments in `cs3430_s24_takehome03.py`.

9. If you can, write below your name and A-number in `cs3430_s24_takehome02.py` how much time you spent on this exam. I will not make it public anywhere. This is only for me to assess the easiness/difficulty/reasonableness of this exam.

10. I wish you best of luck and, as always, Happy Hacking and Thinking!

Problem 1 (2 points)

Implement the function

`acoeffs, bcoeffs = fourier_coeffs(f, num_coeffs=10, a=-math.pi, b=math.pi, romb=7).`

The first argument is a one-argument function `f`. The second keyword argument is the number of Fourier coefficients to compute, i.e., the number of a-coefficients (or cosine coefficients). Recall that the number of cosine coefficients is the number of b-coefficients (or sine coefficients) plus 1. The keyword parameters `a` and `b` specify the interval on which the definite integral is taken to compute the coefficients. They default to $-\pi$ and π, respectively. The keyword parameter `romb` gives the top node of the Romberg lattice whose value is used to estimate the definite integral used in the computation of the Fourier coefficients. Thus, if `romb=7`, then we compute `R(7,7)`. The function returns a 2-tuple of arrays. The first is the array of cosine coefficients; the second is the array of sine coefficients.

If you are still not sure about your Rombergs, Richardsons, and Trapezoidals, you may integrate with `sympy` (cf. Slide 17, Lecture 14 and `sympy_integration.py` in the zip for that lecture). Of course, some precision assertions in the unit tests may not pass. But, I will definitely give you partial credit.

Support materials: Lectures 18, 19; HW09.

Problem 2 (1 point)

Implement the function

`nth_partial_sum(x, acoeffs, bcoeffs)`

that returns the n-th partial sum of the Fourier series at x. The second and third arguments are arrays of the cosine and sine coefficients computed by the function in Problem 1.

Support materials: Lectures 18, 19; HW09.

Problem 3 (1 point)

a) Implement the function

`nth_partial_sum(x, acoeffs, bcoeffs)`

that returns the n-th partial sum of the Fourier series at x. The second and third arguments are arrays of the cosine and sine coefficients computed by the function in Problem 1.

b) Implement the function

```
plot_function_and_nth_partial_sum(f, f_def_str,
                                  num_coeffs=10,
                                  num_points=1000,
                                  a=-math.pi, b=math.pi,
                                  romb=7).
```

The first parameter is a one-argument function. The second parameter is a function description string which will be used in the plot (e.g., `'y=x^2'`), the keyword arguments `num_coeffs`, `a`, `b`, and `romb` have the same meanings as in `fourier_coeffs`. The keyword `num_points` specifies the number of points in the linear space between `a` and `b`, i.e., `np.linspace(a, b, num_points)`. This function does

```
xvals = np.linspace(a, b, num_points)
yvals1 = np.array([f(x) for x in xvals])
yvals2 = np.array([nth_partial_sum(x, acoeffs, bcoeffs) for x in xvals])
```

and then plots `yvals1` and `yvals2` in the same `matplotlib` plot and shows it to the user. I saved the images of all my unit test plots for this problem in the zip, e.g., `problem_03_plot_01.png`, etc. Your plots may not coincide with mine point by point but should, nonetheless, have similar curve shapes.

Support materials: Lectures 18, 19; HW09.

Problem 4 (1 point)

Implement the generator

```
gen_lcg(a, b, m, n, x0=0)
```

to generate `n` random numbers with the linear congruential generation (LCG) algorithm. The last argument is the seed that defaults to 0.

Support materials: Lectures 21, 22; HW10.

Problem 5 (1 point)

Implement the generator

```
gen_xorshift_32(a, b, c, n, x0=1)
```

that implements a 32-bit XORShift generator.

Support materials: Lectures 21, 22; HW10.

Problem 6 (2 points)

Implement the function

```
make_lcg_pix(a, b, m, n, seed=11235813)
```

that returns a numpy array of RGB 3-tuples extracted from LCG random numbers generated with `gen_lcg`.

I included all my random PNGs generated with this function in the unit tests in the zip, e.g., `random_lcg_pil_59.png`, etc. Some images look amazingly random to me.

Support materials: Lectures 21, 22; HW10.

Problem 7 (2 points)

Implement the function

```
make_xorshift_32_pix(a, b, c, n, seed=1)
```

that returns an numpy array of RGB 3-tuples extracted from 32-bit XORShift random numbers generated with `gen_xorshift_32`. Again, I included all my random PNGs generated with this function in the unit tests in the zip, e.g., `random_xor32_pil_31.png`, etc. Some images do look random to me.

Support materials: Lectures 21, 22; HW10.

Problem 8 (4 points)

a) Implement the function

```
get_coin_flip_seqs(num_coin_flips)
```

that returns all experimental outcome string sequences of flipping a fair coin `num_coin_flips` times. The outcome `Heads` is mapped to 1. The outcome `Tails` is mapped to 2. A few trial runs.

```
>>> from cs3430_s24_takehome03 import *
>>> get_coin_flip_seqs(2)
['11', '21', '12', '22']
>>> get_coin_flip_seqs(3)
['111', '211', '121', '221', '112', '212', '122', '222']
```

```
>>> get_coin_flip_seqs(4)
['1111', '2111', '1211', '2211', '1121', '2121', '1221',
'2221', '1112', '2112', '1212', '2212', '1122', '2122',
'1222', '2222']
>>> len(get_coin_flip_seqs(10)) == 2**10
True
```

b) Implement the boolean function

```
is_sufficiently_knuth_random(str_digit_seq, probs)
```

that returns `True` if the string outcome sequence is sufficiently random, according to Dr. Knuth criteria for sequence randomness (i.e., its p-value is in [0.1, 0.9]). The second argument is the dictionary that maps each trial outcome to its probability. In case all outcomes are equiprobable, this dictionary can be generated with `make_equal_probs_table` in `cs3430_s24_takehome03.py`. Here is a sample run.

```
>>> probs = make_equal_probs_table(lower=1, upper=2)
>>> probs
{1: 0.5, 2: 0.5}
>>> is_sufficiently_knuth_random('111', probs)
False
>>> is_sufficiently_knuth_random('121', probs)
True
```

c) Implement the function

```
get_sufficiently_knuth_random_coin_flip_seqs(num_coin_flips)
```

that returns all string outcome sequences in the experiment of flipping a coin `num_coin_flips` times that are sufficiently random, according to Dr. Knuth criteria.

d) Implement the function

```
get_insufficiently_knuth_random_coin_flip_seqs(num_coin_flips)
```

that returns the set of all string outcome sequences of generated by the experiment of flipping a coin `num_coin_flips` times that are insufficiently random, according to Dr. Knuth criteria.

```
>>> get_sufficiently_knuth_random_coin_flip_seqs(3)
{'122', '121', '211', '212', '112', '221'}
>>> get_insufficiently_knuth_random_coin_flip_seqs(3)
{'222', '111'}
>>> srs=get_sufficiently_knuth_random_coin_flip_seqs(3)
>>> isrs=get_insufficiently_knuth_random_coin_flip_seqs(3)
>>> len(srs) + len(isrs) == 2**3
True
```

Support materials: Lectures 22, 23; HW10.

Problem 9 (2 points)

This problem is analogous to Problem 8, except the underlying experiment involves throwing a 6-face die a finite number of times.

a) Implement the function

```
get_die_throw_seqs(num_die_throws)
```

that returns all experimental outcome string sequences of throwing a 6-face die `num_die_throws` times. The trial outcomes are 1, 2, 3, 4, 5, and 6. A few trial runs.

```
>>> get_die_throw_seqs(2)
['11', '21', '31', '41', '51', '61', '12', '22',
'32', '42', '52', '62', '13', '23', '33', '43', '53',
'63', '14', '24', '34', '44', '54', '64', '15', '25',
'35', '45', '55', '65', '16', '26', '36', '46', '56', '66']
>>> len(get_die_throw_seqs(5)) == 6**5
True
```

b) Implement the function

```
get_sufficiently_knuth_random_die_throw_seqs(num_die_throws)
```

that returns all string outcome sequences in the experiment of throwing a 6-face die `num_die_throws` times that are sufficiently random, according to Dr. Knuth criteria.

c) Implement the function

```
get_insufficiently_knuth_random_die_throw_seqs(num_die_throws)
```

that returns the set of all string outcome sequences in the experiment of flipping a coin `num_coin_flips` times that are insufficiently random, according to Dr. Knuth criteria.

```
>>> len(get_die_throw_seqs(5)) == 6**5
True
>>> srs=get_sufficiently_knuth_random_die_throw_seqs(3)
>>> isrs=get_insufficiently_knuth_random_die_throw_seqs(3)
>>> len(srs) + len(isrs) == 6**3
### the devil is always in the insufficiently random details; hence
>>> '666' in isrs
True
```

Support materials: Lectures 22, 23; HW10.

Problem 10 (2 points)

a) Implement the function

```
learn_bin_id3_dt(examples_file_path)
```

that takes a path to a csv file with `PlayTennis` examples and builds a binary decision tree for it. The unit test for this problem uses the file `play_tennis_takehome03.csv` in the zip.

b) Implement the function

```
display_bin_id3_dt(dtr)
```

that takes the root of a binary decision tree built with `BinID3` and prints its nodes in the same fashion as was adopted in HW11. E.g. the decision tree we built for HW11 for Dr. Quinlan's set of 14 examples may be displayed as follows.

```
Outlook
         Sunny
                Humidity
                        Normal
                                Yes
                        High
                                No
        Overcast
                Yes
        Rain
```

```
Wind
        Strong
                No
        Weak
                Yes
```

c) Implement the function

```
entropy(examples, target_attrib, avt)
```

that computes the entropy of the target attribute given a set of `examples`, the `target_attrib` (a string), and the parameter `avt` is a dictionary that maps each attribute to its values. All attributes and values are strings.

d) Implement the function

```
gain(examples, target_attrib, attrib, avt)
```

that computes the information gain of the attribute `attrib` with the respect to the target arrbiute `target_attrib` on the given set of `examples`. The parameter `avt` is a dictionary that maps each attribute to its values.

Support materials: Lectures 24, 25; HW11.

Problem 11 (2 points)

a) Suppose that we have 3 symbols, A, B, and C whose frequencies are all equal to some positive integer n. Build a Huffman tree code to encode/decode messages of these symbols and typeset it as text (cf. `cs3430_s24_takehome03.py` for examples).

b) Encode the following messages with your Huffman tree: A, B, C, AB, AC.

c) Suppose that we have k symbols s_1, s_2, ..., s_k, $k > 0$, of the same frequency. Under what conditions will a Huffman Tree emit the same number of bits for each symbol s_i, $1 \leq i \leq k$?

Support materials: Lectures 25, 26, 27.

What to Submit

Submit your solutions in `cs3430_s24_takehome03.py`. **Remember to include in your submission zip all the Python files you import from.**

This takehome brings CS3430: S24 to an end. I want to thank all of you for throwing your hats on my scientific computing hanger this semester.

My special thanks, from the bottom of my heart, go to those of you who regularly attended my F2F lectures. I would not have been able to make it through this semester without you guys. I appreciate all your questions as well as our spontaneous in-class discussions, arguments, and, of course, jokes, e.g., the digital Hamlet, `2b XOR 0b => ?`, made my semester. These are all precious gifts to me. I did my best to incorporate most discussions and arguments in my slides and homework/exam comments.

If you are graduating, may your journey have a purpose. If you need to stick around a while yet, may the FOMO never guide your learning.