CSC326: Assignment 2

Due: 10:50 AM, Monday, October 29

Problem 1. The integral $\int_a^b f(x)dx$ can be approximated using the Trapezoidal rule with n subintervals as follows:

$$\int_a^b f(x)dx \quad \approx \quad \frac{h \times (f(a) + f(b))}{2} + \sum_{i=1}^{n-1} f(a + i \times h)$$

Where $h = {}^{b-a}/_{n-1}$. In the file `q1.py`, define a function `integrate(f,a,b,n)` that *returns* the approximation of the above integral as a floating point number according to the above formula. Implement this function with a loop.

Define a second function in the same file called `integrate_fast(f,a,b,n)` that uses NumPy's `numpy.vectorize` and `numpy.sum` libraries to implement a faster version of the above algorithm, again *returning* the approximation as a floating point number.

Finally, define a third function `integrate_speedup(f,a,b,n)` that uses Python's `timeit` module (see `http://docs.python.org/library/timeit.html`, especially the last example) to measure the running time of `integrate(f,a,b,n)` and `integrate_fast(f,a,b,n)` and *returns* the speedup *as a floating point number*. For example, if A runs in 10 seconds and B runs in 3 seconds then 3.33 is the speedup. Round it off to two decimal points. In order to receive full marks, your NumPy implementation must be (for sufficiently large n) faster than your basic Python implementation.

Use the following two functions in your own tests; however, be sure not to include your test runs in the submitted assignment.

$$\begin{aligned} f_1(x) &= 10x^2 + 3x + 1 \\ f_2(x) &= e^{x^2} log_e(cos(x)) \end{aligned}$$

Problem 2. In the file `q2.py`, create two functions: `estimate_dist(n)` and `estimate_dist_fast(n)` for integral n where $0 \ll n$. In both functions, generate a vector of integers of length n, where the integers are taken from a uniform random distribution over $[1, 20]$.

The first function will construct this vector using Python's `random` module then count the number of times that 10 occurs in the vector using a loop.

The second function will construct this vector using NumPy's `numpy.random` module and count the number of times that 10 occurs in the vector using NumPy's `numpy.vectorize` and `numpy.sum`.

Both functions must *return* a floating point number representing an estimate of how often 10 occurs. For example, if $n = 20$ and 10 occurs twice, return $100 \times \frac{2}{10} = 20.00$, i.e. that 6 occurred 20% of the time. In this example, return 20.00 (upto two decimal points).

Create a third function in the same file, `estimate_speedup(n)`, that behaves similarly to `integrate_speedup` but for `estimate_dist` and `estimate_dist_fast`. Again, in order to receive full marks, the speedup should be greated than 1 for sufficiently large n.

Problem 3. In the file `q3.py`, create a function `word_hist(path)` that extracts all of the words from a file[1] located at `path`, counts the number of occurrences of each word, and *returns* a list of tuples, where the first item in the tuple is the word and the second is an integer representing the number of occurences of that word. The list should be sorted in the descending order by occurrence.

Problem 4. Given an $n \times n$ grid/matrix of live/dead (live = `1`, dead = `0`) cells, a single iteration of Conway's Game of Life applies one of the following rules to each cell of the grid (note: different cells will satisfy different rules) in order to get the next state of the grid.

1. Any live cell with fewer than two live neighbours dies, as if caused by under-population.
2. Any live cell with two or three live neighbours lives on to the next generation.
3. Any live cell with more than three live neighbours dies, as if by overcrowding.
4. Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.

These rules can be implemented in Python as follows:

```
def iterate(board):
    """board is a list of lists in row-major form."""

    num_rows, num_cols = len(board), len(board[0])

    # duplicate the old board, but extend it with rows/cols of zeros
    # along the edges so that we don't need to do bounds checking, but
    # instead do 1-based indexing instead of 0-based indexing
    old_board = [[0] * (num_cols + 2) for _ in xrange(num_rows + 2)]
    for r in xrange(num_rows):
        for c in xrange(num_cols):
            old_board[r+1][c+1], board[r][c] = board[r][c], 0

    # count the neighbors of each cell and apply the rules
    for r in xrange(1, num_rows + 1):
        for c in xrange(1, num_cols + 1):
            num_neighbors = old_board[r-1][c] + old_board[r][c-1] + \
                            old_board[r-1][c-1] + old_board[r+1][c] + \
                            old_board[r][c+1] + old_board[r+1][c+1] + \
                            old_board[r-1][c+1] + old_board[r+1][c-1]

            if old_board[r][c]:
                board[r-1][c-1] = int(num_neighbors in (2, 3))
            else:
                board[r-1][c-1] = int(3 == num_neighbors)
        return board
```

For example, the following is three iterations of Conway's Game of Life when run on the 3×3 starting grid:

$$\begin{array}{|c|c|c|}\hline 1 & 1 & 0 \\ \hline 0 & 1 & 0 \\ \hline 0 & 1 & 0 \\ \hline \end{array} \to \begin{array}{|c|c|c|}\hline 1 & 1 & 0 \\ \hline 0 & 1 & 1 \\ \hline 0 & 0 & 0 \\ \hline \end{array} \to \begin{array}{|c|c|c|}\hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline 0 & 0 & 0 \\ \hline \end{array} \to \begin{array}{|c|c|c|}\hline 1 & 0 & 1 \\ \hline 1 & 0 & 1 \\ \hline 0 & 1 & 0 \\ \hline \end{array}$$

Use NumPy's slicing and broadcasting features, create a function `iterate(board)` that performs one iteration of Conway's Game of Life on two dimension NumPy array. You will be penalized if you use `for` loops. Hint: count neighbor cells!

[1] One word per line of the file. All words will use only alphabetic characters. **Do not** include spaces or newlines in your words.

Notes

You have to investigate online documentations yourself to get infomration about the aforementioned modules (e.g. search for "NumPy reference"). This documentation will help you.

Also, for your convenience, Python includes a function called `help`; try it! For example, run `help("hello")` in the Python interpreter.

This time we do not provide any test code.

Submission deadline is 10:50 am on Monday, October 29, 2012

Submission Instructions

Compress all of your files and save them either as `asn2.zip` or `asn2.tar.gz` and submit the files from a computer in one of the following labs: GB243, GB251E, or SF2102, or by SSH by logging on to one of the aformentioned computers (ug132.eecg - ug180.eecg, ug201.eecg - ug249.eecg).

Submission Command Example

```
ug132:~% submitcsc326f 2 asn2.zip
```

SSH Example

```
$ ssh username@ug180.eecg.toronto.edu
...
username@ug180.eecg.toronto.edu's password: ...
ug132:~%
```