Instructions for the programming project

This document is explaining to you what you will have to do (and how you will do it) for the CS101 programming project in 2014/15 at *Mahindra Ecole Centrale*.

Topic: You and your team-mates will work on some algorithms used to numerically compute integrals (1D or more). This is not something new, as we studied some methods in MA101 (October, November), but you will also discover some other approaches. The goal of this project is to implement correctly some algorithms that will allow us to compute numerical approximations of 1D integrals (area under the curve), and possibly 2D integrals (surface) or 3D integrals (volume). At the end, you will be able to compute the same integral $\int_{x_{min}}^{x_{max}} f(x)\mathrm{d}x$ with half-a-dozen of different approaches, and compare their efficiency and accuracy. The user interface we want you to provide will have to be similar to this:

```python
import integrals  # importing the file integrals.py

def f1(x):  # here we define an example function
    return x**2 + x - 4

# Using the function riemann_left that you wrote in integrals.py
area = integrals.riemann_left(f1, xmin=0, xmax=10, n=1000)
print "For f: x -> x**2 + x - 4, the area under the curve on [0, 10] is ↩
    approximatively", area
```

General instructions: Please refer to the main document we gave you, it contains general advices, details about the deadline, and tells you how to submit your work.

1 What do you have to do?

1.1 In a file called `integrals.py`

1. Start by implementing the Riemann sum algorithm, as seen in MA101[1].

   You will first define a function `riemann_left`, that implements the left Riemann rectangle method. Such function can be defined with `def riemann_left(f, xmin, xmax, n=1e5)`, with `f` being the function[2] for which we want to compute $\int_{xmin}^{xmax} f(x)\mathrm{d}x$, so `xmin` (or `x_min`) and `xmax` (or `x_max`) are respectively left and right bounds of the integral. The last parameter `n` is the number of rectangles to be used. Remember that the left Riemann rule is stating that $\int_{x_{min}}^{x_{max}} f(x)\mathrm{d}x \simeq \left(\frac{x_{max} - x_{min}}{n}\right) \sum_{k=0}^{n-1} f(x_k)$, where $x_k = x_{min} + h \times k$ for $0 \leqslant k \leqslant n-1$ (for n big enough).

   Similarly, implement `riemann_right` and `riemann_center` for right and centred Riemann sums.

2. Similarly, use the *trapezoidal rule* to define a function called `trapez`. Which of these first 4 functions is the more accurate? Try to compare them on the same function, while increasing the number of rectangles. Give their complexity (as functions of `n` the number of rectangles).

3. The Monte-Carlo integration is an completely different algorithm, based on numbers (x, y) randomly picked in a rectangle $[x_{min}, x_{max}] \times [y_{min}, y_{max}]$ (with $y_{min} \leqslant \min_{x_{min} \leqslant x \leqslant x_{max}} f(x) \leqslant y_{max}$).

[1] For this entire project, a good reference is the Apostol Vol.II book, from page 602, paragraph 15.19 to 15.23.
[2] It should be callable, with one argument returning a numerical value, exactly like the example given above.

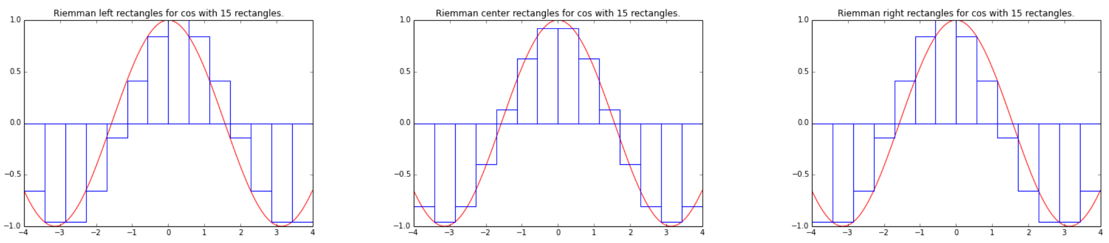

Figure 1: Three rectangles methods being illustrated with PyLab (MatPlotLib)

Please refer to en.wikipedia.org/wiki/Monte_Carlo_integration for more details, examples and illustrations. The basic concept is to randomly pick a huge number of points in this rectangle, and count how many of them are below the curve $y = f(x)$ (easy by comparing the picked value y_i and the computed value $f(x_i)$). So if you picked n points, and k are below the curve, the area under the curve is approximatively equal to the area of the rectangle $((x_{max} - x_{min}) \times (y_{max} - y_{min}))$ times the observed probability of getting a point under the curve (the rate k/n).

Is this method more or less efficient than the RIEMANN method? If you use $n = 1000$ rectangles and if you pick $n = 1000$ random points, which of the two methods give a more accurate approximation of the integral?

4. Implement the SIMPSON rule, which is similar to the trapezoidal rule (reference is given below). Is it more accurate? Is the O time complexity worse than for the previous methods?

5. Finally, use the *bases of plotting with Python*[3] that will be taught soon in lectures, in order to illustrate the different methods. You must illustrate *at least* the 3 RIEMANN rectangle methods (left, right, centred) on different examples. This part can look like this (you can use that code, and adapt it for right and centred rectangles):

```python
from pylab import *  # we load the PyLab environment

def plot_riemann_left(f, xmin, xmax, name_f, n=10):
    """ Plot the function f from xmin to xmax, and n left Riemann rectangles."""
    figure()  # new figure
    title("Riemman left rectangles for " + name_f + " with " + str(n) + " rectangles.")
    xvalues = linspace(xmin, xmax, 1000)  # values for the x axis
    yvalues = [ f(x) for x in xvalues ]  # and y axis
    plot(xvalues, yvalues, 'r-')  # plots the function as a red line
    h = (xmax - xmin) / float(n-1)
    xi = [ xmin + i*h for i in xrange(0, n-1) ]
    yi = [ f(x) for x in xi ]  # left rectangles!
    # Now plot the rectangles
    for x, y in zip(xi, yi):
        # Rectangle (x,0), (x,y), (x+h,y), (x+h,0)
        plot([x, x, x+h, x+h, x], [0, y, y, 0, 0], 'b-')

# One example (to moved to the tests.py file after)
print plot_riemann_left(cos, -4, 4, "cos", n=15)
```

Below is included three illustration of the RIEMANN sum methods (left, centered and right), cf. Fig.1.

[3] You can go through the first examples of that tutorial (www.labri.fr/perso/nrougier/teaching/matplotlib/).

1.2 On-line references, and references in Apostol, Vol.II

- Main references: en.wikipedia.org/wiki/Numerical_integration explains well the problem and give links to different approaches (see also mathworld.wolfram.com/NumericalIntegration.html).

  en.wikipedia.org/wiki/Newton-Cotes_formulas give details about Newton-Cotes formula, the generalization of Riemann and trapezoidal formulas.
- For Riemann rectangles rules, en.wikipedia.org/wiki/Riemann_sum#Methods give details, formulas and illustration of the different methods (left, right, middle).
- en.wikipedia.org/wiki/Trapezoidal_rule explains well the trapezoidal rule.
  You can also refer to the Theorem 15.13, in Apostol Vol.II (page 604).
- Simpson's rule is also well explained on-line[4] at mathworld.wolfram.com/SimpsonsRule.html.
  It is also proved and explained on Theorem 15.15, in Apostol Vol.II (page 609).

1.3 In a file called `tests.py`

This other file is mandatory, and will show many examples of use of all the algorithms you implemented.

1. Write *at least three examples* (of different kinds, like one polynomial, one trigonometric and another function) for *every* algorithm you implemented in the `integrals.py` file.
2. Do not use random or meaningless examples, try to take examples that can be computed mathematically (in order to check the correctness and precision of your results).
3. For example, this `tests.py` file will look like this:

```python
import math
import integrals

area = integrals.riemann(math.sin, 0, math.pi, n=1000)
# area should be almost 2.0
print "For sinus, the area under its curve on [0, pi] is ↩
    approximatively", area

# many more to do !

def finv(x): return 1.0 / x
area = integrals.montecarlo(finv, 1, 4, n=10000)
# area should be almost ln(4) which is approximatively 1.34
print "For the function inverse, the area under its curve on [0, pi]↩
     is approximatively", area
```

4. Feel free to get examples from the MA102 course, like the examples seen in lectures, or the problems in either the exams or the tutorial sheets. Any interesting examples that you want to illustrate or compute in your project is most surely welcome.

1.4 In your project report

- Explain any choice or special hypotheses you chose to follow for all the algorithms you implemented. E.g. if one function expect its argument `f` to be continuous or differentiable (or more), specify it.
- If you choose to implement any extra features to the `integrals.py` file (it can be an extra function or method), please explain it in details in your report.

[4] Also here en.wikipedia.org/wiki/Simpsons_rule.

- For *each* function and method you write, give its time complexity ($O(n), O(n^2)$ etc) in its *docstring*[5]. Here, a difficulty can be to know what is the parameter to count as size of the problem (it can be the precision threshold `h` or the number of rectangles or random points `n`).

---

2 Possible extra job?

Any of the following task will give you extra marks[6].

1. Do you have any idea[7] of a non-random method that can be used to compute surfaces, volumes or higher-dimension integrals ($2D$, $3D$ or more) ?
2. For all your functions and methods, you can take the time to check that the given arguments are valid. For example, `n` should be non-negative, and `x_max` $\geqslant$ `x_min`.
   Hint: it is easy to perform such checks, by using the `assert` keyword. Example: `assert n > 0 and x_max >= x_min` is a very easy way to check this.
3. Implement any of these additional methods:
   - These two methods are a little bit harder to program but more efficient :
     - Boole's rule (as explained here mathworld.wolfram.com/BoolesRule.html),
     - Romberg's method (as explained here en.wikipedia.org/wiki/Romberg's_method).
   - The general Gaussian quadrature method is really harder, but not out of your reach, if you have time: mathworld.wolfram.com/GaussianQuadrature.html.
   - Finally, advanced methods such as adaptive quadrature or adaptive Simpson rule can also be considered (en.wikipedia.org/wiki/Adaptive_quadrature).

---

3 Extra advices

- Each function or method should be correctly implemented. The *correctness* of everything you write *is crucial* (the required examples are in fact here to help you detect any issue).
- Try to be careful about the special cases (empty integrals, division by zero, integer division etc).
- Do not cheat from your friends or from Internet.
- Apply the rule "the more efficient, the better" for each function/method you write. Do not try to optimize each line, what is important is the *overall complexity* of the algorithms (e.g. $O(n^2)$ is better than $O(n^3)$). You can take some ideas from the exams or the labs.
- But keep in mind that *readability and simplicity of your programs are important*!
- Please contact me if needed (`CS101@crans.org` or `Lilian.Besson@ens-cachan.fr`).

---

[5] A good example of such practice is the solution for Problem III for CS second mid term exam (given on Moodle).
[6] It is not possible to aim the best mark without trying any of these...
[7] Hint: Wikipedia and Internet can help you, but do not copy-paste anything without trying to understand.

4 Symbolic and numerical integration in Python

Of course, all this has already been written in Python. The goal is not to use these packages, but to write your own `integrals.py` file, and all these algorithms by yourself.

If you are curious, you can compare your results with the ones obtained with one of the following package. These two modules are already included in Anaconda, and should be available directly in Spyder.

4.1 With SciPy/NumPy

The main reference is the scipy.integrate module (Scientific Python, cf. docs.scipy.org/doc/scipy/reference/tutorial/integrate.html#integration-scipy-integrate).

4.2 With SymPy

Another good reference is the `sympy` (Symbolical Python) module, which provides a general purpose sympy.integrate function, designed to compute formal integrals (ie. exact value and not numerical approximations).

For more details, the `sympy.mpmath` package provide implementation of the Gaussian quadrature and other numerical methods (docs.sympy.org/latest/modules/mpmath/calculus/integration.html) .

Last link is docs.sympy.org/latest/modules/integrals/integrals.html#numeric-integrals which explains how to use and compute GAUSSIAN quadratures.