Math 246 Unit 6: Definite Integrals with the Trapezoid, Midpoint and Simpson Rules

Brenton LeMesurier, Revised November 18, 2015

Now that we have seen all the core Python features needed to implement mathematical algorithms, we start on some more substantial computations: numerical approximation of definite integrals.

We will learn two new Python features: the command to **sum** a list of values, and organizing collections of functions and such into **modules**, for convenient reuse elsewhere.

General Notes

- Use a (multi-line) ’’’triple-quoted''' comment atop each file to provide documentation, including at least your name and the date. This makes **help(module_name)** work after **import module_name**.
- Also use a triple-quoted *documentation* quote at the top of each function definition in a module. This makes **help(module_name.function_name)** work after **import module_name**, and **help(function_name)** after **from module_name import function_name**.
- In most other places, normal single-line “#” comments are fine.
- When I specify the file names to use, *use exactly those names!*
- Modules should in general only *define* functions and such, leaving the *use* of those functions to other files (or notebook) which import them from the module.
- The main exception is adding testing or demonstrations to a module file: put these (indented) inside an `if` block at the bottom of the module file, started with:

  if __name__ == "__main__":

(Each if those four long dashes is a *pair* of underscores!)

- Whenever possible, check your Python files with me before submitting them to OAKS; in particular, check that they run successfully and that the output at least appears to be correct.
- **Tip:** Separate *evaluation* (with formulas and function calls) from *output* (with print statements and such): compute values and store them into variables, and then print the values of those variables.
- While developing and in interactive work like notebooks, it can be convenient to access all the needed stuff from *numpy* with the *wild import*

  ```
  from numpy import *
  ```
- If basic *matplotlib* stuff is also needed to produce graphs, one can also use

  ```
  from matplotlib.pyplot import *
  ```
- An alternative that only works within notebooks and when working with the IPython interactive command window is the IPython *magic command*

```
%pylab
```

which essentially does

```
from numpy import *

from matplotlib.pyplot import *
```

plus a few other things.

- When using matplotlib graphics in a notebook, also use either

```
%matplotlib inline
```

or

```
%matplotlib tk
```

according to where you want the graphs to appear.

- **However**, in Python files and modules, it is not possible to use magic commands like `%pylab`, and it is best to import items needed explicitly with one of the following patterns, illustrated here by plotting the graph of $y = \sin(x)$.

A)

```
import numpy
import matplotlib.pyplot

x = numpy.linspace(-numpy.pi, numpy.pi)
y = numpy.sin(x)
matplotlib.pyplot.plot(x, y)
```

B) Save some typing later by using the standard nicknames for these modules:

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(-np.pi, np.pi)
y = np.sin(x)
plt.plot(x, y)
```

C) Save even more typing later by explicitly importing the items needed:

```
from numpy import linspace, pi, sin
from matplotlib.pyplot import plot

x = linspace(-pi, pi)
y = sin(x)
plot(x, y)
```

In [1]:

```
# No graphs needed today, so:
from numpy import *
```

Objectives

We will learn several ways to perform summations, applying this to several methods for numerically approximating definite integrals as seen in calculus courses: the composite trapezoid and midpoint rules. As a bonus, we will then see an easy way of greatly improving the accuracy of the results.

Exercise 6.1: the (composite) trapezoid rule

To approximate the definite integral

$$I = \int_a^b f(x)dx$$

the composite trapezoid rule with n equal intervals is

$$T_n = h\left(\frac{f(a)}{2} + \sum_{i=1}^{n-1} f(x_i) + \frac{f(b)}{2}\right)$$

with

$$h = \frac{b-a}{n}, \quad x_i = a + ih, \quad 0 \le i \le n$$

Create a module **integration.py** and within this a function with *signature*

```
trapezoid1(f, a, b, n)
```

which uses a loop to do the necessary summations, and returns T_n. That is, its usage form is

```
T_n = trapezoid1(f, a, b, n)
```

Then read up on the Python function `sum`, and create a new version `trapezoid2` using that instead of a loop. Put it in the same module **integration**.

Test from a file `unit6_exercise1.py`, using initially the example of

$$\int_1^3 \frac{dx}{x^2}$$

This has the exact value 2/3, so also compute and display the error in each of your approximations.

Exercise 6.2: the (composite) midpoint rule

Mimic the above for the composite midpoint rule (the natural best Riemann sum).

Just do one version, using the second "sum" method above, and again put the definition

```
def midpoint(f, a, b, n): ...
```

in module **integration**.

Exercise 6.3: more thorough testing

Choose a second example that is not a polynomial (so that the result is unlikely to be exact) and not zero at either endpoint (so that coding errors at the endpoints are not hidden), but for which you know the exact value of the integral.

Then output the error in each approximation to the integral, along with the approximation and the corresponding value of n.

Test with $n = 10, 100, 1000$ and maybe more, to check against the expectation that errors are proportional to $1/n^2$.

Finally, choose third example for which the exact value of the definite integral cannot be got using anti-derivative an the Fundamental Theorem of Calculus. Again use the values $n = 10, 100$, and so on, and try to estimate the accuracy of your approximations by comparing the results for different values of n.

Exercise 6.4: nicely formatted output

Once your code is doing the math correctly, produce a final testing file **unit6_exercise4.py** which presents the results nicely:

- Titles saying which method is being tested, and on which integral.
- Values for n, T_n, M_n and (when known) the error, lined up in columns.

The last feature requires methods for formatting strings and output, which can be done several ways in Python. For now, just one example:

In [2]:

```
# Three arbitrary values, just for printing:
x = 237/11.
n = 99
err = 0.000000236572
print('In test %4i, the solution is %-12.5f with estimated error %9.2e' % (n, x,
err))
# Three more arbitrary values:
x = x**3
n += 1
err = err/7.
print('In test %4i, the solution is %-12.5f with estimated error %9.2e' % (n, x,
err))
```

```
In test   99, the solution is 21.54545     with estimated error  2.3
7e-07
In test  100, the solution is 10001.54245  with estimated error  3.3
8e-08
```

The chunks starting with "%" and ending with a letter specify that a value from the tuple at right after the final "%" gets inserted, with certain formating:

- %i (or %d) is for integers
- %f is for floating point numbers in *fixed point* format (no exponent)
- %e is floating point numbers in scientific notation (with exponent)
- %g chooses between "e" and "f" format, according to which is more compact

What do the numbers in between mean? What does a minus sign after the "%" and in front of a number mean?

To answer that, experiment, or read all the details online in Section 4.7.2, printf-style String Formatting (https://docs.python.org/3.4/library/stdtypes.html#printf-style-string-formatting) of the Python 3.4 Standard library manual (https://docs.python.org/3.4/library/).

Exercise 6.5: improved accuracy (Simpson's rule)

Check out

$$S_{2n} = \frac{T_n + 2 * M_n}{3}$$

Create a function for this, using the above-defined functions for the trapezoid and midpoint rules, and do the coding, testing, and output in the modular style developed above. (As always, try to provide at least estimates of errors, and actual error values where those are available.)

This is our first example of using a function that we have defined from within another function, and it illustrates why we generally want functions that calculate values to return those values through output variables *without* any printing from within the function, or at least with the option of not printing.