Laboratory 9: Numerical integration, numpy

Prerequisites: +numpy, +scipy.integrate.quad, dictionaries

Contents

PEP8

To monitor the coding style we turn on the PEP8 style guide monitoring in Spyder by:

1. Going to preferences in Spyder menu
2. Clicking on `Editor` in the selection list on the left
3. Clicking on `Code Introspection/Analysis` (top right corner)
4. Ticking the box `Real-time code style analysis`
5. Clicking `Apply` and `OK` (bottom right corner)

Exercises

Download the file lab9.py and open it in your editor. Press F5 so that all definitions in the file are visible at the Python prompt. Add all functions that you are asked to write to this file.

1. Write a function `trapez(f, a, b, n)`:

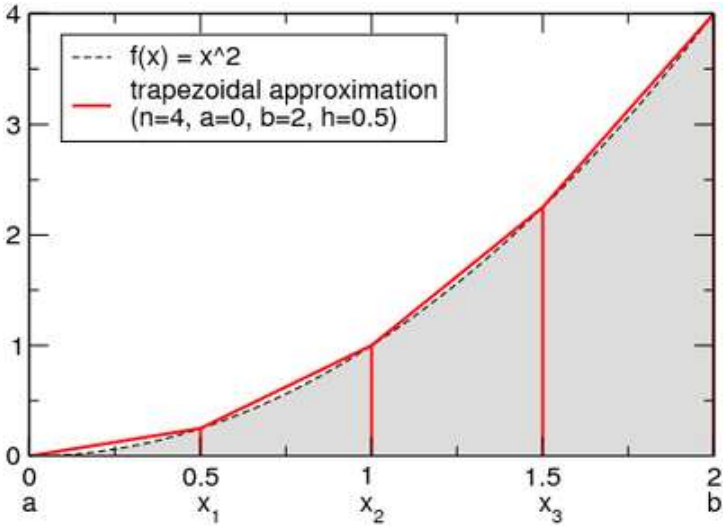
   Background: The equation below shows that the integral `I` of the function `f(x)` from `x=a` to `x=b` can be approximated by `A` through the so-called composite trapezoidal integration rule:

$$I = \int_a^b f(x)\mathrm{d}x \approx A = \frac{h}{2}\left(f(a) + f(b) + 2\sum_{i=1}^{n-1} f(x_i)\right)$$

   where h=(b-a)/n, x_i=a+i*h, and n is the number of subdivisions.

   Example:

   The figure demonstrates the idea for f(x)=x^2:



   The composite trapezoidal rule computes the area under the red line exactly, where the red line is an approximation of the actual function of interest (here f(x)=x^2) shown by a black dashed line.

   Exercise:

Write a function `trapez(f, a, b, n)` which is given

- a function `f` which depends on one parameter (i.e. f(x))
- an lower (`a`) and upper (`b`) integration limit
- and the number of subdivisions (`n`) to be used.

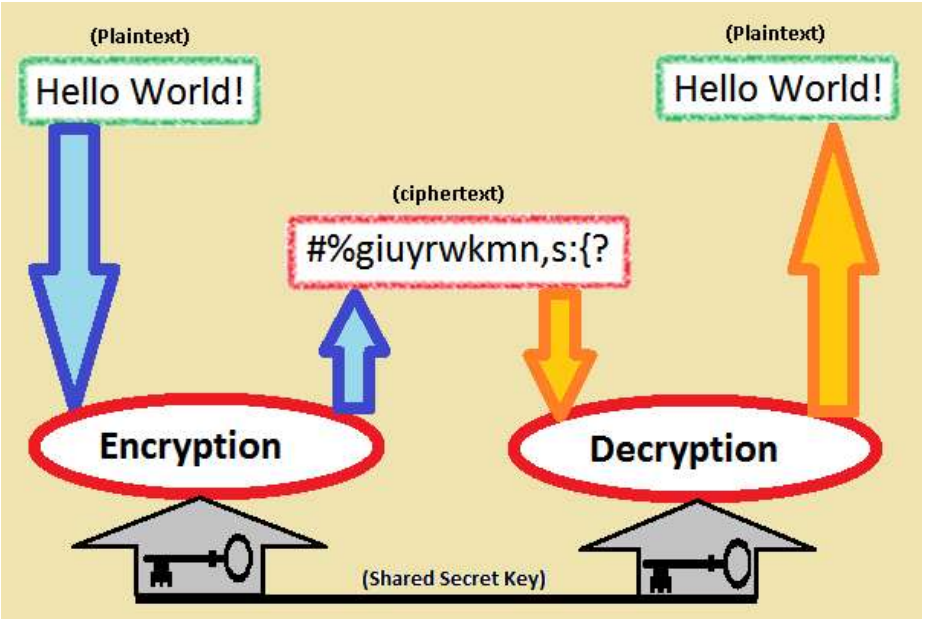
The function should use the composite trapezoidal rule to compute `A` and return this value.

Examples

```
In [ ]: def f(x):
   ...:     return x
   ...:

In [ ]: trapez(f, 0, 1, 1)
Out[ ]: 0.5

In [ ]: trapez(f, 0, 1, 5)
Out[ ]: 0.5

In [ ]: def f2(x):
   ...:     return x * x
   ...:

In [ ]: trapez(f2, 0, 1, 1)
Out[ ]: 0.5

In [ ]: trapez(f2, 0, 1, 10)
Out[ ]: 0.3350000000000001

In [ ]: trapez(f2, 0, 1, 100)
Out[ ]: 0.33335000000000004

In [ ]: trapez(f2, 0, 1, 1000)
Out[ ]: 0.33333349999999995
```

2. **Background**: In this part of the practical, we work with simple cryptographic codes. Imagine Alice wants to send a secret message to Bob and does not want third parties to be able to understand the message. Here, we imagine that Alice and Bob can agree on a particular *code* before they need to exchange messages.

   Alice and Bob will use the same key to (i) encode a message (i.e. change from plain text to something less readable) and later (ii) decode the encoded message back into plain text. This is known as Symmetric-key cryptography, where the same key is used both for encryption and decryption. The image below shows this schematically. The "encoded" message is called "cyphertext" in the image, and the code used for encoding and decoding is the "shared secret key".



A code in this lab is a mapping that takes an (input) letter of the alphabet and maps it to a different (output) letter. We can represent such a code in Python through a dictionary where the keys of the dictionary are the input characters and the values the output characters.

For example, let's consider the trivial code that replaces the letter 'e' in the input data with the letter 'x' in the output, and simultaneously replaces the letter 'x' in the input data with the letter 'e' in the output. This could be written as `e->x` and `x->e`.

This can be represented by the dictionary `{'x': 'e', 'e': 'x'}`. The function `code1` provides exactly this dictionary:

```
In [ ]: mycode = code1()

In [ ]: print(mycode)
{'x': 'e', 'e': 'x'}
```

We use the convention that if a character is not found in the keys of the dictionary, the character should not be changed by the code.

We now encode the string `Hello World` using the mapping described by `mycode`, and obtain the string `Hxllo World`.

(If we want to encode information without losing data, we need to make sure that no two keys map to the same value, i.e. the mapping has to be injective. Later, we want to reverse the mapping -- to *decode* a coded message -- and will need that the mapping has to be bijective, i.e. there has to be a one-to-one correspondence between input and output sets.)

Exercise: Write a function `encode(code, msg)` which takes the arguments:

- `code`: this is a dictionary describing a code
- `msg`: this is a string

The function should apply the mapping to each character of the string as described above and return the encoded output string.

Examples

```
In [  ]: code = code1()

In [  ]: print(code)
{'x': 'e', 'e': 'x'}

In [  ]: encode(code, "Hello World")
Out[  ]: 'Hxllo World'

In [  ]: encode(code, "Jimi Hendrix")
Out[  ]: 'Jimi Hxndrie'

In [  ]: encode(code, "The name xenon (Xe) is derived from greek for strange.")
Out[  ]: 'Thx namx exnon (Xx) is dxrivxd from grxxk for strangx.'
```

The sentence `"the quick brown fox jumps over the lazy dog"` contains all letters of the English alphabet, and might be useful here as an example:

```
In [ ]: msg = "the quick brown fox jumps over the lazy dog"

In [ ]: encode(code1(), msg)
Out[ ]: 'thx quick brown foe jumps ovxr thx lazy dog'
```

The `lab9.py` file provides other codes, for example `code2` which maps `i->s`, `s->g` and `g->i`:

```
In [ ]: code2()
Out[ ]: {'g': 'i', 'i': 's', 's': 'g'}

In [ ]: encode(code2(), msg)
Out[ ]: 'the qusck brown fox jumpg over the lazy doi'
```

The code provided by function `code3` makes the encoded message fairly unreadable:

```
In [22]: print(code3())
  {'!': '?', ' ': '$', '#': '.', '$': ' ', '.': '#', '?': '!', 'A': 'B', 'C': 'D',
   'B': 'C', 'E': 'F', 'D': 'E', 'G': 'H', 'F': 'G', 'I': 'J', 'H': 'I', 'K': 'L',
   'J': 'K', 'M': 'N', 'L': 'M', 'O': 'P', 'N': 'O', 'Q': 'R', 'P': 'Q', 'S': 'T',
   'R': 'S', 'U': 'V', 'T': 'U', 'W': 'X', 'V': 'W', 'Y': 'Z', 'X': 'Y', 'Z': 'A',
   'a': 'b', 'c': 'd', 'b': 'c', 'e': 'f', 'd': 'e', 'g': 'h', 'f': 'g', 'i': 'j',
   'h': 'i', 'k': 'l', 'j': 'k', 'm': 'n', 'l': 'm', 'o': 'p', 'n': 'o', 'q': 'r',
   'p': 'q', 's': 't', 'r': 's', 'u': 'v', 't': 'u', 'w': 'x', 'v': 'w', 'y': 'z',
   'x': 'y', 'z': 'a'}

In [23]: msg
Out[23]: 'the quick brown fox jumps over the lazy dog'

In [24]: encode(code3(), msg)
Out[24]: 'uif$rvjdl$cspxo$gpy$kvnqt$pwfs$uif$mbaz$eph'
```

3. Write a function `reverse_dic(d)` that takes a dictionary `d` as the input argument and returns a dictionary `r`. If the dictionary `d` has a key `k` and an associated value `v`, then the dictionary `r` should have a key `v` and a value `k`. (This is only expected to work for dictionaries that describe a bijective mapping although you do not need to check for this.)

   (This was an exercise in laboratory 8 and if you did this correctly, you can use the same code here.)

   Examples:

   ```
   In [ ]: reverse_dic({"dog": 10, "cat": 20})
   Out[ ]: {10: 'dog', 20: 'cat'}

   In [ ]: code2()
   Out[ ]: {'g': 'i', 'i': 's', 's': 'g'}

   In [ ]: reverse_dic(code2())
   Out[ ]: {'g': 's', 'i': 'g', 's': 'i'}
   ```

4. Write a function `finderror(n)` which uses the `trapez()` function to find the error of the trapezoidal integral approximation. The function should compute the integral of `f(x) = x*x` with integration limits a = -1 and b = 2 numerically. The function should then subtract this numerical result A from the exact integral value I and return the difference. Use analytical methods to find the exact integral value I.

   Example:

   ```
   In [ ]: finderror(5)
   Out[ ]: -0.17999999999999972
   ```

   (You should find approximately that the error decays by a factor 4 if you double `n` (because the trapez method has an error of order h^2 and h is proportional to 1/n. You can plot the error as a function h if you like and discuss the curve with a demonstrator although this is not assessed.)

5. Write a function `using_quad()` that computes the integral of x^2 from a=-1 to b=2 using function `quad` from `scipy.integrate`. The function `using_quad` should return the value that the `quad()` function returns, i.e. a tuple `(y, abserr)` where `y` is `quad`'s best approximation of the true integral, and `abserr` an absolute error estimate.

   You should find that this method (with the default settings) is (far) more accurate than our `trapez` function.

6. A function `std_dev(x)` which takes a list `x` of floating point numbers, and computes and returns the corrected sample standard deviation of the floating point numbers in the list `x`. For clarity, if the list `x` with `N` elements (assuming `N > 1`)

   $$x = [x_1, x_2, x_3, \ldots, x_N]$$

   then the corrected sample standard deviation is given by:

$$\sigma = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \mu)^2}$$

where `mu` is the (arithmetic) mean:

$$\mu = \frac{1}{N}\sum_{i=1}^{N} x_i$$

Examples:

```
In [ ]: std_dev([1.0, 1.0])
Out[ ]: 0.0

In [ ]: std_dev([1.0, 2.0])
Out[ ]: 0.7071067811865476
```

7. Write a function `decode(code, encoded_msg)` that takes a dictionary `code` that contains the mapping dictionary with which the string `encoded_msg` has been encoded. The function `decode` should return the decoded message.

   Example:

```
In [ ]: msg = "the quick brown fox jumps over the lazy dog"

In [ ]: code2()
Out[ ]: {'g': 'i', 'i': 's', 's': 'g'}

In [ ]: x = encode(code2(), msg)

In [ ]: x
Out[ ]: 'the qusck brown fox jumpg over the lazy doi'

In [ ]: decode(code2(), x)
Out[ ]: 'the quick brown fox jumps over the lazy dog'



In [ ]: y = encode(code3(), msg)

In [ ]: y
Out[ ]: 'uif$rvjdl$cspxo$gpy$kvnqt$pwfs$uif$mbaz$eph'

In [ ]: decode(code3(), y)
Out[ ]: 'the quick brown fox jumps over the lazy dog'
```

   Hints: Once you have the function `reverse_dic` and `encode` defined in the Training part working, you can decode a coded message by using the reversed code dictionary in the `encode` function:

```
In [ ]: msg = "the quick brown fox jumps over the lazy dog"

In [ ]: encoded = encode(code2(), msg)

In [ ]: encoded
Out[ ]: 'the qusck brown fox jumpg over the lazy doi'

In [ ]: code2_reversed = reverse_dic(code2())

In [ ]: encode(code2_reversed, encoded)
Out[ ]: 'the quick brown fox jumps over the lazy dog'
```

   Can you now decode the message:

```
Zpv$ibwf$tvddfttgvmmz$efdpefe$uijt$tfdsfu$nfttbhf#$Dpohsbuvmbujpot?
```

which has been encoded with the mapping provided by function `code3`? The encoded string is also available under the variable name `secretmessage` in `training9.py`.

Submit your files `lab9.py` by email as usual for testing.

Last updated: 2019-01-06 at 17:34

Return to Top

Page last modified Fri Dec 7 16:12:32 2018. |