ex01_solution

February 19, 2025

Computational Quantum Physics - PHYS 463

Lecturer: Prof. G. Carleo

Assistants: alessandro.sinibaldi@epfl.ch, linda.mauron@epfl.ch, lorenzo.fioroni@epfl.ch

0.1 Solution 01

0.1.1 Problem 1.1 - The perils of calculating derivatives

[1]:
```
# Import useful libraries
import numpy as np
import matplotlib.pyplot as plt
```

[2]:
```
# Define a function `f(x)` returning the value `x * (x - 1)`
def f(x):
    return x * (x - 1)
```

[3]:
```
# Numerical derivative by difference
def compute_df(x, delta):
    return (f(x + delta) - f(x)) / delta
```

[4]:
```
# Compute the numerical derivative at `x = 1` with `delta = 0.01`
print(compute_df(1, 0.01))
```

```
1.010000000000001
```

The analytical derivative is $f'(x) = 2x - 1$ and $f'(1) = 1$. The numerical result is not exactly the same, because we used a finite δ rather than took the limit $\delta \to 0$.

If we want to quantitatively estimate the error, we can use Taylor expansion:

$$\begin{aligned}\frac{f(x+\delta) - f(x)}{\delta} &= \frac{f(x) + f'(x)\delta + \frac{1}{2}f''(x)\delta^2 + O(\delta^3) - f(x)}{\delta} \\ &= f'(x) + \frac{1}{2}f''(x)\delta + O(\delta^2).\end{aligned}$$

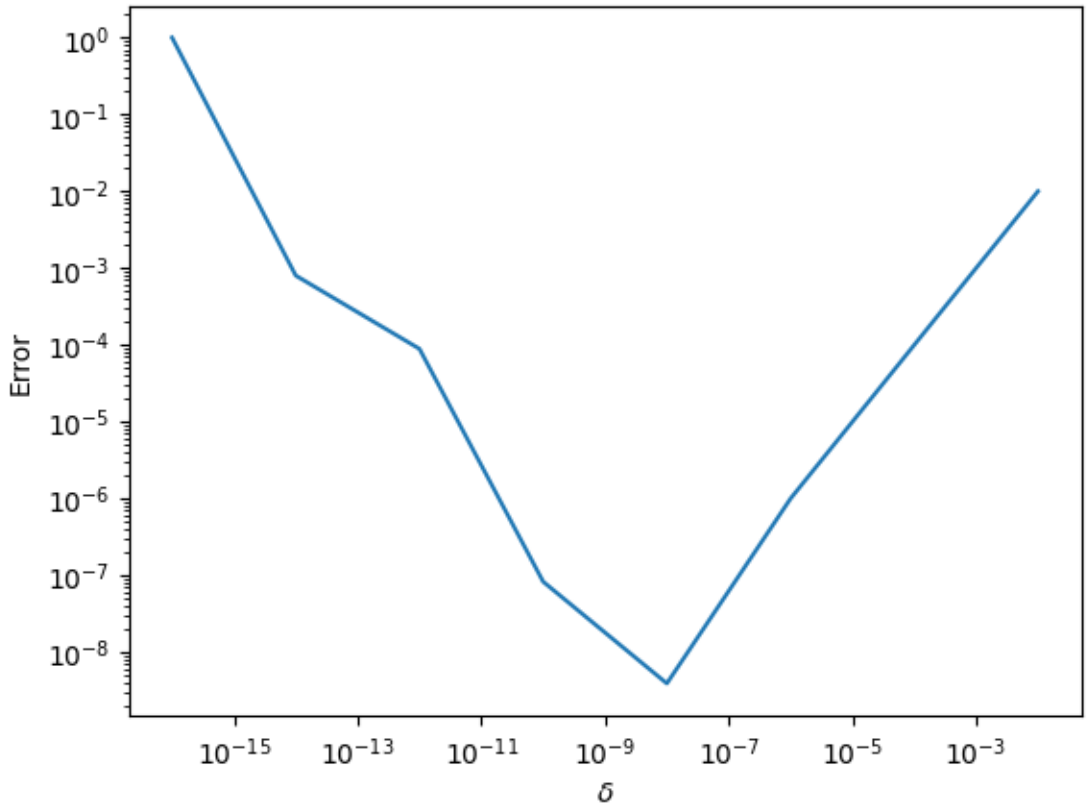
In our case $f(x) = x(x-1)$, it is especially simple because $f'''(x)$ and higher-order derivatives are all zero, and the error only contains the linear term of δ.

[5]:
```
# Use a for loop to compute the error for each `delta`,
# and save the results in the list `errors`
deltas = [1e-2, 1e-4, 1e-6, 1e-8, 1e-10, 1e-12, 1e-14, 1e-16]
```

```
errors = []
for delta in deltas:
    df = compute_df(1, delta)
    error = abs(df - 1)
    errors.append(error)
    print(delta, error)
```

```
0.01 0.010000000000000897
0.0001 9.999999988985486e-05
1e-06 9.99917733279787e-07
1e-08 3.922528746258536e-09
1e-10 8.284037100736441e-08
1e-12 8.890058334132256e-05
1e-14 0.0007992778373491216
1e-16 1.0
```

[6]:
```
# Let's make a plot
plt.figure()
plt.plot(deltas, errors)
plt.xlabel('$\\delta$')
plt.ylabel('Error')
plt.xscale('log')
plt.yscale('log')
plt.show()
```



As δ gets smaller, the error decreases linearly as predicted above, until $\delta = 10^{-8}$. Then the error increases because the *floating-point error* becomes significant.

In computers, there are no really 'real' numbers like $\pi = 3.14159\ldots$, only approximations to them, which we call 'floating-point numbers' or 'floats'. Whenever we do arithmetics of floats, it will introduce some error. This error is usually negligible for use in daily life, but in numerical computations we need to be careful about it!

Let's consider a simple model for the floating-point error: assume there is an error ϵ when we compute $x + \delta$ (and no floating-point error anywhere else). Then the total error is

$$\begin{aligned} e_{\text{tot}}(\delta) &= \frac{f(x+\delta+\epsilon) - f(x)}{\delta} - f'(x) \\ &= \frac{f(x+\epsilon) + f'(x+\epsilon)\delta + \frac{1}{2}f''(x+\epsilon)\delta^2 + O(\delta^3) - f(x)}{\delta} - f'(x) \\ &= \frac{f(x+\epsilon) - f(x)}{\delta} + f'(x+\epsilon) - f'(x) + \frac{1}{2}f''(x+\epsilon)\delta + O(\delta^2) \\ &= \frac{f'(x)\epsilon}{\delta} + \frac{1}{2}f''(x)\delta + O(\delta^2) + O(\epsilon). \end{aligned}$$

The function $e_{\text{tot}}(\delta)$ is minimized when δ is of the order $\sqrt{\epsilon}$. In our case, the default type of floating-point number is float64 with 53-bit significand precision, so $\epsilon \approx 2^{-53} \approx 10^{-16}$, and e_{tot} is minimized when $\delta \approx 10^{-8}$.

In general, if we subtract two large numbers to get a small number (like $f(x+\delta) - f(x)$) or divide something by a small number, there may be significant floating-point error. In future lectures we'll see more cases of floating-point error and how to deal with them.

0.1.2 Problem 1.2 - Game of Life

[7]:
```
import numpy as np

## import libraries to make animations
from matplotlib import animation, rc
import matplotlib.pyplot as plt
from IPython.display import HTML
```

[8]:
```
def init(x_size, y_size, coord):
    state = np.zeros((x_size,y_size))
    for c in coord:
        state[c] = 1
    plt.imshow(state)
    plt.show()
    return state
```

[9]:
```
def update(state):
    new_state = np.zeros(state.shape)
    temp_state = np.pad(state, 1, mode='constant')
```

```
    for i,j in np.ndindex(state.shape):
        temp = temp_state[i+1,j+1]
        n_alive = np.sum(temp_state[i:i+3, j:j+3]) - temp

        if temp == 1 and n_alive > 3 or n_alive < 2:
            continue

        elif temp == 1 and  n_alive >= 2 or temp == 0 and n_alive == 3:
            new_state[i,j] = 1
    return new_state
```

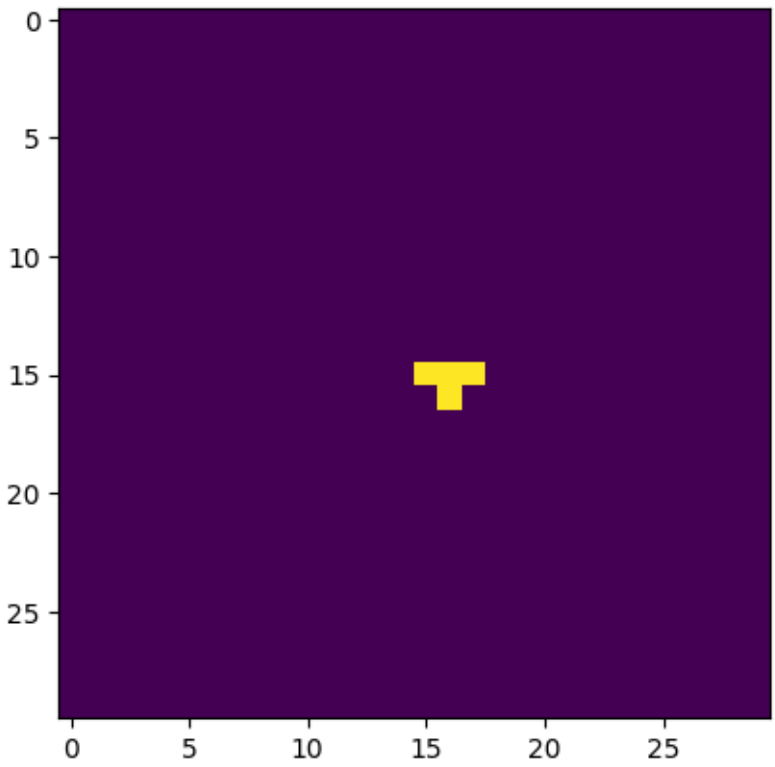
[10]:
```
n_iter = 500
size = 30

# here you can set your desired initial configuration
coords = [(15,15),(15,16),(15,17),(16,16)]

state = init(size, size, coords)

# all_states is the container in which we save
# the configurations of Conway's Game of Life
all_states = [state]

for _ in range(n_iter):
    state = update(state)
    all_states.append(state)
```



[11]:
```
# for the animation of Conway's Game of Life
# data should be a time-ordered list of
# configurations e.g. data[t,x,y] such that
# data[5] gives the configurations after 5
# evolution steps

def make_animation(data):
    fig = plt.figure(figsize=(5,5))
    rc('animation', html='html5')
    ax = plt.axes()

    im=plt.imshow(data[0])

    def init_anim():
        im.set_data(data[0])
        return [im]

    def animate(i):
        im.set_array(data[i])
        return [im]

    anim = animation.FuncAnimation(fig, animate, init_func=init_anim,
```

```
                           frames = len(data), interval =150, blit = True)

    plt.close(fig)

    return anim
```

[12]:
```
a = make_animation(all_states)
```

[13]:
```
## If this does not work, you have to install the ffmpeg package
## an easy way to do this is:
## with conda:    conda install -c conda-forge ffmpeg
## with homebrew: brew install ffmpeg
## on linux:      sudo apt install ffmpeg
## on windows:    install ffmpeg with conda, or alternatively,
##                download ffmpeg from https://ffmpeg.org/download.
 ↪html#build-windows
##                unpack the archive to somewhere and tell matplotlib where␣
 ↪ffmpeg.exe is located with:
# import matplotlib.pyplot as plt
# plt.rcParams['animation.ffmpeg_path'] = r'C:\...\bin\ffmpeg.exe'
# at the top of the notebook, where you have to replace ... with the actual path

# Save the animation as an mp4 file
a.save('solutions_ex01_videos/game_of_life.mp4')

HTML(a.to_html5_video())
```

[13]:
```
<IPython.core.display.HTML object>
```

0.1.3 Problem 1.3 - Diagonalizing random hermitian matrices

[14]:
```
def random_gue(n):
    '''
    Generate a random hermitian matrix of size nxn
    sampled from the gaussian unitary ensemble (GUE)
    i.e. from p(H) ~ exp(-n/2 tr(H^2))
    '''
    A = (np.random.normal(size=(n, n))+ 1j * np.random.normal(size=(n, n)))/np.
 ↪sqrt(2) # CN(0,1)
    H = (A+A.conj().T)/np.sqrt(2*n)

    return H
```

[15]:
```
H = random_gue(2048)
```

[16]:
```
# check it's really hermitian
np.allclose(H, H.conj().T)
```

[16]: True

[17]:
```
# verify it follows the correct distribution
np.mean(H), np.var(H)*2048
```

[17]: ((1.101080713562214e-05+2.991085094991748e-21j), 0.9996026567127662)

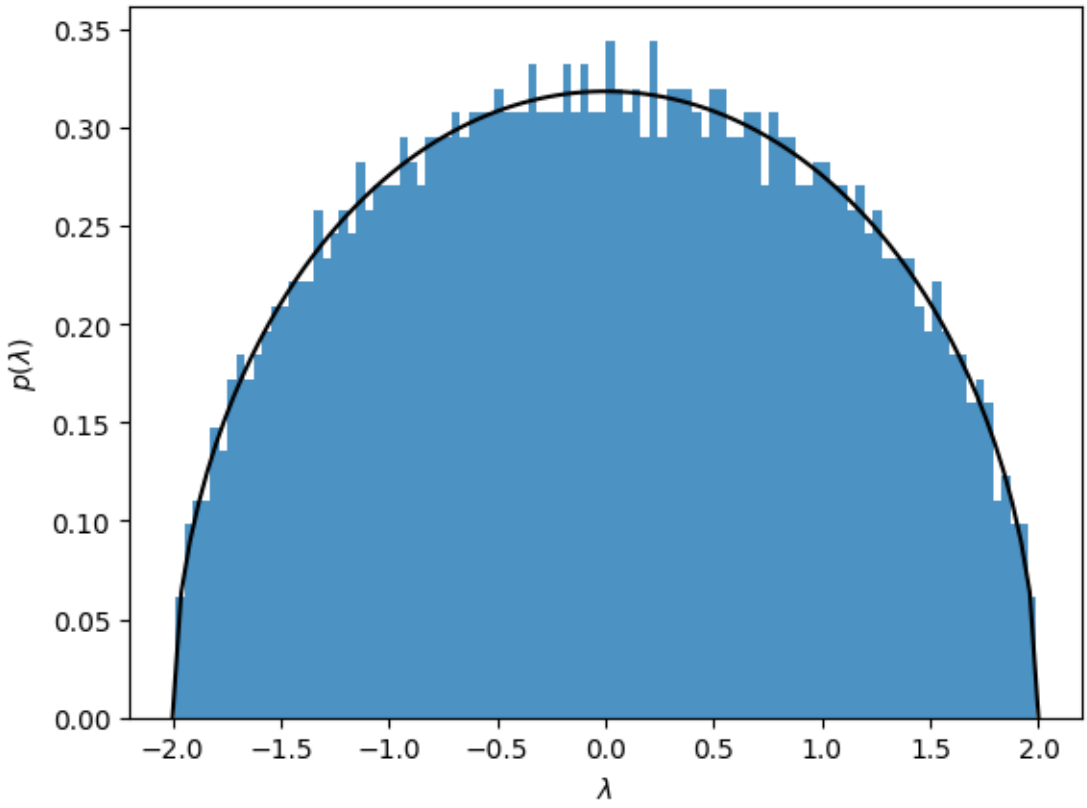
[18]:
```
# call numpy routine for computing the eigenvalues of a hermitian matrix
spectrum = np.linalg.eigvalsh(H)
```

[19]:
```
# probability density function of the wigner semicircle law for radius R
def f(r, R):
    p = 2./np.pi/(R**2) * np.sqrt(np.clip(R**2-r**2, 0, None))
    return p
```

[20]:
```
# plot a histogram of the spectrum
plt.figure()
plt.hist(spectrum, bins=100, density=True, alpha=0.8)

# plot the wigner rule
R = 2
x = np.linspace(-R, R, 100)
plt.plot(x, f(x, R), color='k')

plt.xlabel('$\lambda$')
plt.ylabel('$p(\lambda)$')
plt.show()
```



[21]:
```
## Now generate 1000 samples of 4x4 matrices
n = 4
n_samples = 100_000

samples = np.array([random_gue(n) for _ in range(n_samples)])
```

[22]:
```
# compute the mean Hij
mu = samples.mean(axis=0)

# check it's close to 0
np.allclose(mu, 0, atol=1e-2)
```

[22]: True

[23]:
```
# compute Hij Hmn*
xi = np.einsum('bij,bmn->ijmn', samples, samples.conj())/n_samples

# check it is equal to delta_im delta_jn / n
delta = np.eye(n)
expected = 1/n * np.einsum('im,jn->ijmn', delta, delta)
```

```
np.allclose(expected, xi, atol=1e-2)
```

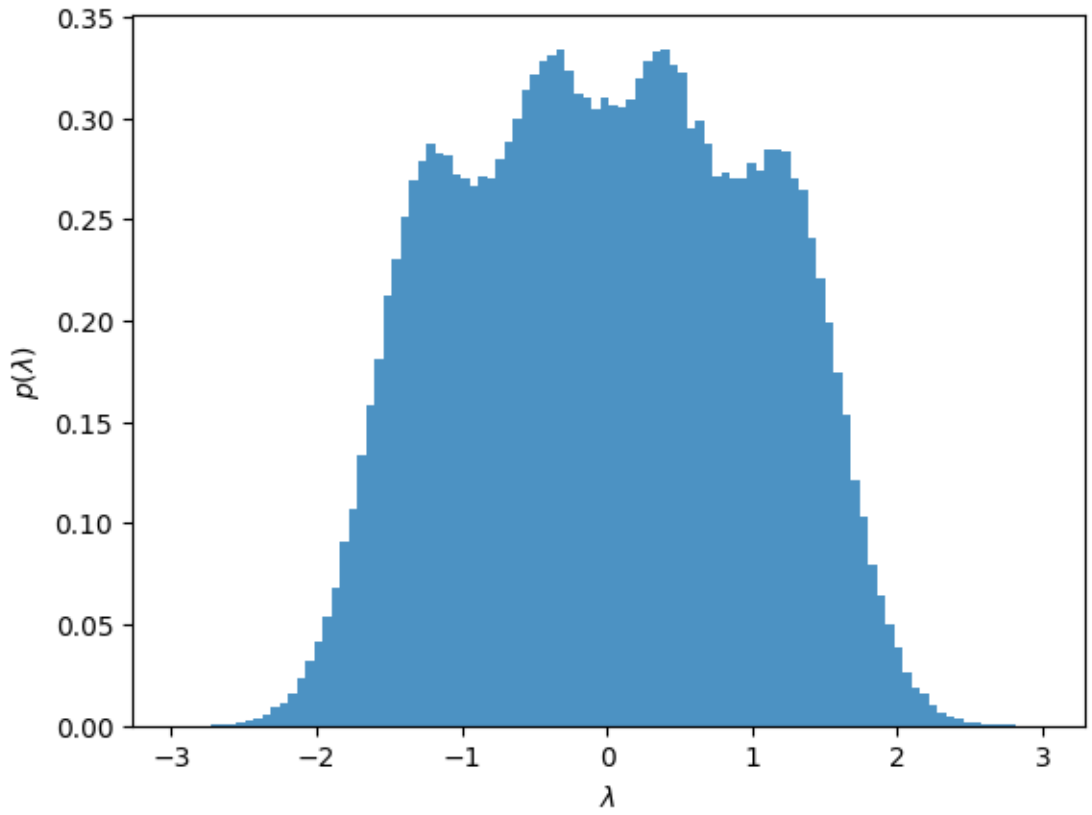
[23]: True

0.1.4 Bonus part: plot the distribution of the level-spacings

[24]:
```
def p2(s):
    # approximate distribution for the level spacings
    return 32./(np.pi**2) * s**2 * np.exp(-4./np.pi * s**2)
```

[25]:
```
spectrum = np.linalg.eigvalsh(samples)
plt.hist(spectrum.ravel(), bins=100, density=True, alpha=0.8)
plt.xlabel('$\lambda$')
plt.ylabel('$p(\lambda)$')
```

[25]: Text(0, 0.5, '$p(\\lambda)$')



[26]:
```
# compute the level-spacings and plot their distribution
spacings = (spectrum[:, 1:]-spectrum[:, :-1]).ravel()
spacings = spacings/spacings.mean()
```

```
plt.hist(spacings, bins=100, density=True, alpha=0.8)
x = np.linspace(spacings.min(),spacings.max(), 100)
plt.plot(x, p2(x), color='k')
plt.xlabel('$s$')
plt.ylabel('$p(s)$')
```

[26]: Text(0, 0.5, '$p(s)$')