10 Exercises

September 13, 2020

0.1 Exercise 10.1 (search)

We want to find the largest and smallest values in a long list of numbers. Implement two algorithms, based on:

1. Iterating over the list entries; and
2. First applying a built-in sort operation to the list.

Encapsulate each algorithm in a function. To create lists of numbers for testing use, for example:

```
x = np.random.rand(1000)
```

0.1.1 Solution

We first create the list of random numbers

[0]:
```python
import numpy as np
x = np.random.rand(1000)
```

Approach 1

[2]:
```python
def min_max1(x):
    if x[0] > x[1]:
        x_max = x[0]
        x_min = x[1]
    else:
        x_max = x[1]
        x_min = x[0]
    for i in range(len(x)):
        if x[i] > x_max:
            x_max = x[i]
        if x[i] < x_min:
            x_min = x[i]
    return  x_min, x_max

print(min_max1(x))
```

```
(0.0007777799014705433, 0.9994129005589826)
```

Approach 2

[3]:
```python
def min_max2(x):
    def quicksort(A, lo=0, hi=None):
        "Sort A and return sorted array"

        # Initialise data the first time function is called
        if hi is None:
            A = A.copy()
            hi = len(A) - 1

        # Sort
        if lo < hi:
            p = partition(A, lo,  hi)
            quicksort(A, lo, p - 1)
            quicksort(A, p + 1, hi)
        return A

    def partition(A, lo, hi):
        "Partitioning function for use in quicksort"
        pivot = A[hi]
        i = lo
        for j in range(lo,  hi):
            if A[j] <= pivot:
                A[i], A[j] = A[j], A[i]
                i += 1
        A[i], A[hi] = A[hi], A[i]
        return i

    y = quicksort(x)
    x_min = y[0]
    x_max = y[len(y)-1]

    return x_min, x_max


print(min_max2(x))
```

```
(0.0007777799014705433, 0.9994129005589826)
```

[0]:
```python
assert min_max1(x) == min_max2(x)
```

In practice, we would use the the NumPy function:

[5]:
```python
print(np.min(x), np.max(x))
```

```
0.0007777799014705433 0.9994129005589826
```

[6]:
```
%time min_max1(x)
%time min_max2(x)
%time (np.min(x), np.max(x))
```

```
CPU times: user 561 µs, sys: 124 µs, total: 685 µs
Wall time: 759 µs
CPU times: user 8.7 ms, sys: 934 µs, total: 9.64 ms
Wall time: 10.1 ms
CPU times: user 892 µs, sys: 0 ns, total: 892 µs
Wall time: 654 µs
```

[6]:
```
(0.0007777799014705433, 0.9994129005589826)
```

0.2 Exercise 10.2 (Newton's method for root finding)

0.2.1 Background

Newton's method can be used to find a root x of a function $f(x)$ such that

$$f(x) = 0$$

A Taylor series expansion of f about x_i reads:

$$f(x_{i+1}) = f(x_i) + f'\big|_{x_i} (x_{i+1} - x_i) + O((x_{i+1} - x_i)^2)$$

If we neglect the higher-order terms and set $f(x_{i+1})$ to zero, we have Newton's method:

$$x_{i+1} = -\frac{f(x_i)}{f'(x_i)} + x_i \tag{1}$$

$$x_i \leftarrow x_{i+1} \tag{2}$$

In Newton's method, the above is applied iteratively until $|f(x_{i+1})|$ is below a tolerance value.

0.2.2 Task

Develop an implementation of Newton's method, with the following three functions in your implementation:

```python
def newton(f, df, x0, tol, max_it):
    # Implement here

    return x1  # return root
```

where `x0` is the initial guess, `tol` is the stopping tolerance, `max_it` is the maximum number of iterations, and

```python
def f(x):
    # Evaluate function at x and return value


def df(x):
    # Evaluate df/dx at x and return value
```

Your implementation should raise an exception if the maximum number of iterations (`max_it`) is exceeded.

Use your program to find the roots of:

$$f(x) = \tan(x) - 2x$$

between $-\pi/2$ and $\pi/2$. Plot $f(x)$ and $f'(x)$ on the same graph, and show the roots computed by Newton's method.

Newton's method can be sensitive to the starting value. Make sure you find the root around $x = 1.2$. What happens if you start at $x = 0.9$? It may help to add a print statement in the iteration loop, showing x and f at each iteration.

0.2.3 Extension (optional)

For a complicated function we might not know how to compute the derivative, or it may be very complicated to evaluate. Write a function that computes the *numerical derivative* of $f(x)$ by evaluating $(f(x+dx) - f(x-dx))/(2dx)$, where dx is small. How should you choose dx?

0.2.4 Solution

We first implement a Newton solver function:

0.3 Exercise 10.2 (Newton's method for root finding)

0.3.1 Background

Newton's method can be used to find a root x of a function $f(x)$ such that

$$f(x) = 0$$

A Taylor series expansion of f about x_i reads:

$$f(x_{i+1}) = f(x_i) + \left. f' \right|_{x_i} (x_{i+1} - x_i) + O((x_{i+1} - x_i)^2)$$

If we neglect the higher-order terms and set $f(x_{i+1})$ to zero, we have Newton's method:

$$x_{i+1} = -\frac{f(x_i)}{f'(x_i)} + x_i \tag{3}$$

$$x_i \leftarrow x_{i+1} \tag{4}$$

In Newton's method, the above is applied iteratively until $|f(x_{i+1})|$ is below a tolerance value.

We now provide implementations of `f` and `df`, and find the roots:

[0]:
```
f = lambda x: np.tan(x) - (2 * x)
Df = lambda x: (1/np.cos(x))**2 - 2
```

[0]:
```
def newton(f,Df,x0,tol=1e-8, max_it=20):
    '''Approximate solution of f(x)=0 by Newton's method.

    Parameters
```

```
    ----------
    f : function
        Function for which we are searching for a solution f(x)=0.
    Df : function
        Derivative of f(x).
    x0 : number
        Initial guess for a solution f(x)=0.
    tol : number
        Stopping criteria is abs(f(x)) < tol.
    max_it : integer
        Maximum number of iterations of Newton's method.

    Returns
    -------
    xn : number
        Implement Newton's method: compute the linear approximation
        of f(x) at xn and find x intercept by the formula
            x = xn - f(xn)/Df(xn)
        Continue until abs(f(xn)) < tol and return xn.
        If Df(xn) == 0, return None. If the number of iterations
        exceeds max_it, then return None.

    Examples
    --------
    >>> f = lambda x: x**2 - x - 1
    >>> Df = lambda x: 2*x - 1
    >>> newton(f,Df,1,1e-8,10)
    Found solution after 5 iterations.
    1.618033988749989
    '''
    xn = x0
    for n in range(0,max_it):
        fxn = f(xn)
        if abs(fxn.all()) < tol:
            print('Found solution after',n,'iterations.')
            return xn
        Dfxn = Df(xn)
        if Dfxn.all() == 0:
            print('Zero derivative. No solution found.')
            return None
        xn = xn - fxn/Dfxn
    print('Exceeded maximum iterations. No solution found.')
    return None
```

[23]:

```
import scipy
from scipy import optimize
x = np.linspace(-np.pi/2, np.pi/2, 10)
```

```python
r0 = newton(f, Df, x, tol=1e-8, max_it=20)
r1 = scipy.optimize.newton(f, x, tol=1e-08, maxiter=20)
print(r0)
print(r1)
```

```
Found solution after 4 iterations.
[-1.57079633e+00 -1.16556119e+00  1.72835786e+02 -9.26442286e-23
  0.00000000e+00  0.00000000e+00  9.26442286e-23 -1.72835786e+02
  1.16556119e+00  1.57079633e+00]
[-1.57081388e+00 -1.16556119e+00  4.37016320e-25  0.00000000e+00
  0.00000000e+00  0.00000000e+00  0.00000000e+00 -4.37824113e-25
  1.16556119e+00  1.57081388e+00]
```

We can visualise the result:

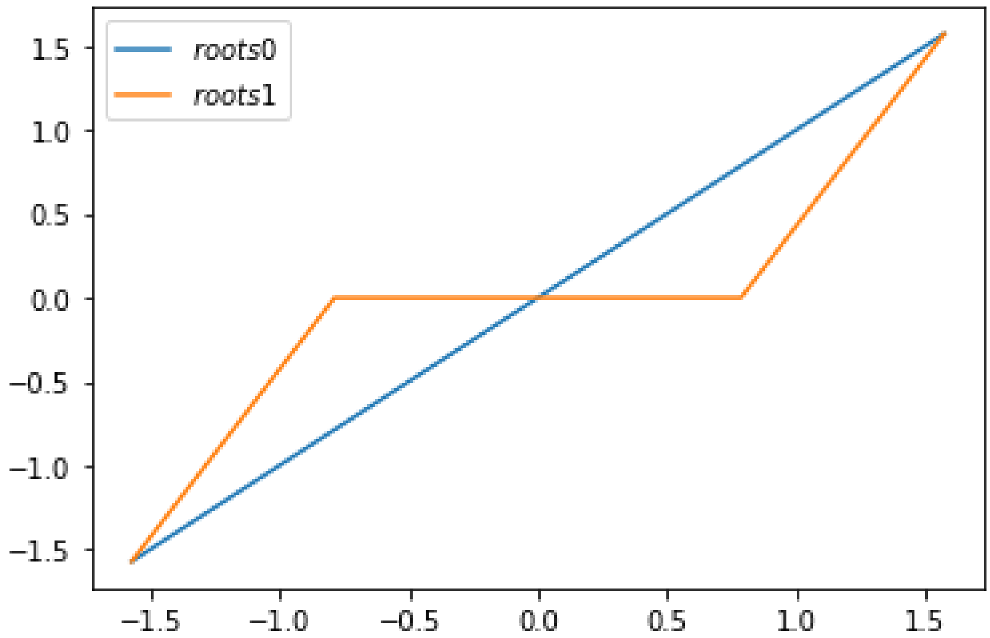
[24]:
```python
%matplotlib inline
import matplotlib.pyplot as plt

# Plot f and df/dx
x = np.linspace(-np.pi/2, np.pi/2, 5)
r0 = newton(f, Df, x, tol=1e-8, max_it=20)
r1 = scipy.optimize.newton(f, x, tol=1e-08, maxiter=20)
#plt.plot(x, f(x), label='$f(x)$')
#plt.plot(x, Df(x), label="$f^{\prime}(x)$")

# Add location of roots to plot
plt.plot(x, r0, label='$roots 0$')
plt.plot(x, r1, label='$roots 1$')

plt.legend()
plt.show()
```

```
Found solution after 0 iterations.
```



For the extension, we can replace the function `df(x)` with a new version

[0]:
```
def df(x):
    # Try changing dx to 1e-15 or smaller
    dx = 1e-9
    return (f(x + dx) - f(x - dx)) / (2*dx)
```

[30]:
```
# Find roots near -1.2, 0.1, and 1.2
xroots = np.array((newton(f, df, -1.2),
                   newton(f, df, 0.1),
                   newton(f, df, 1.2)))
xroots
#assert np.isclose(xroots, [-1.16556119e+00, 2.08575213e-10, 1.16556119e+00]).
↪all()
```

```
Exceeded maximum iterations. No solution found.
Found solution after 3 iterations.
Exceeded maximum iterations. No solution found.
```

[30]:
```
array([None, 0.0, None], dtype=object)
```

In practice, we could use the Newton function `scipy.optimize.newton` from SciPy (http://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.newton.html) rather than implementing our own function.

0.4 Exercise 10.3 (optional, low pass image filter)

Images files can be loaded and displayed with Matplotlib. An imported image is stored as a three-dimensional NumPy array of floats. The shape of the array is `[0:nx, 0:ny, 0:3]`. where `nx` is the number of pixels in the x-direction, `ny` is the number of pixels in the y-direction, and the third axis is for the colour component (RGB: red, green and blue) intensity. See http://matplotlib.org/users/image_tutorial.html for more background.

Below we fetch an image and display it:

[0]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import matplotlib.image as mpimg

# Import image
img = mpimg.imread('https://raw.githubusercontent.com/matplotlib/matplotlib.
↪github.com/master/_images/stinkbug.png')

# Check type and shape
print(type(img))
print("Image array shape: {}".format(img.shape))

# Display image
plt.imshow(img);
```

The task is to write a *function* that applies a particular low-pass filter algorithm to an image array and returns the filtered image. With this particular filter, the value of a pixel in the filtered image is equal to the average value of the four neighbouring pixels in the original image. For the `[i, j, :]` pixel, the neighbours are `[i, j+1, :]`, `[i, j-1, :]`, `[i+1, j, :]` and `[i-1, j, :]`.

Run the filter algorithm multiple times on the above image to explore the effect of the filter.

Hint: To create a NumPy array of zeros, `B`, with the same shape as array `A`, use:

```
import numpy as np
B = np.zeros_like(A)
```

[0]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import matplotlib.image as mpimg

# Import image
img = mpimg.imread('https://raw.githubusercontent.com/matplotlib/matplotlib.
↪github.com/master/_images/stinkbug.png')

# Check type and shape
print(type(img))
print("Image array shape: {}".format(img.shape))

# Display image
```

```
plt.imshow(img);
```

[0]:
```
fft_img = np.zeros_like(img,dtype=complex)
for ichannel in range(fft_img.shape[2]):
    fft_img[:,:,ichannel] = np.fft.fftshift(np.fft.fft2(img[:,:,ichannel]))
```

[0]:
```
def filter_circle(TFcircleIN,fft_img_channel):
    temp = np.zeros(fft_img_channel.shape[:2],dtype=complex)
    temp[TFcircleIN] = fft_img_channel[TFcircleIN]
    return(temp)

fft_img_filtered_IN = []
fft_img_filtered_OUT = []
## for each channel, pass filter
for ichannel in range(fft_img.shape[2]):
    fft_img_channel  = fft_img[:,:,ichannel]
    ## circle IN
    temp = filter_circle(TFcircleIN,fft_img_channel)
    fft_img_filtered_IN.append(temp)
    ## circle OUT
    temp = filter_circle(TFcircleOUT,fft_img_channel)
    fft_img_filtered_OUT.append(temp)

fft_img_filtered_IN = np.array(fft_img_filtered_IN)
fft_img_filtered_IN = np.transpose(fft_img_filtered_IN,(1,2,0))
fft_img_filtered_OUT = np.array(fft_img_filtered_OUT)
fft_img_filtered_OUT = np.transpose(fft_img_filtered_OUT,(1,2,0))
```

[0]:
```
abs_fft_img              = np.abs(fft_img)
abs_fft_img_filtered_IN  = np.abs(fft_img_filtered_IN)
abs_fft_img_filtered_OUT = np.abs(fft_img_filtered_OUT)
```

[0]:
```
def inv_FFT_all_channel(fft_img):
    img_reco = []
    for ichannel in range(fft_img.shape[2]):
        img_reco.append(np.fft.ifft2(np.fft.ifftshift(fft_img[:,:,ichannel])))
    img_reco = np.array(img_reco)
    img_reco = np.transpose(img_reco,(1,2,0))
    return(img_reco)

img_reco              = inv_FFT_all_channel(fft_img)
img_reco_filtered_IN  = inv_FFT_all_channel(fft_img_filtered_IN)
img_reco_filtered_OUT = inv_FFT_all_channel(fft_img_filtered_OUT)

fig = plt.figure(figsize=(25,18))
ax  = fig.add_subplot(1,3,1)
ax.imshow(np.abs(img_reco))
```

```
ax.set_title("original image")

ax  = fig.add_subplot(1,3,2)
ax.imshow(np.abs(img_reco_filtered_IN))
ax.set_title("low pass filter image")


ax  = fig.add_subplot(1,3,3)
ax.imshow(np.abs(img_reco_filtered_OUT))
ax.set_title("high pass filtered image")
plt.show()
```

0.5 Reference

https://fairyonice.github.io/Low-and-High-pass-filtering-experiments.html