Tutorial_2 Notebook

March 19, 2019

0.1 Optimisation and Minimization

Notebook for ***Tutorial 2****, Phys3112 S1 2019*

Try running the cells in this notebook and understanding them.

```
In [1]: import scipy.optimize
        import numpy as np
        import random
        from matplotlib import pyplot as plt
```

1 1-D Brute force solver

This is a simple brute force solver! We are going to find the minimum of a function by randomly guessing until we have found an acceptable point. Our function takes only one input (x) so this is a One Dimensional input problem. We also will only have one output.

```
In [2]: # Define a function that we are going to try and mimize
        # Feel free to change this function around!
        def func_1(x):
            return np.abs(-x**4 + 5*x**3 - 2*x**2 + 17*x - 10)
```

```
In [3]: # We will stop our search after finding this output or lower!
        threshold = 0.001
        # Our range of input values
        input_min = 0
        input_max = 10
        searching = True
        # Number of iterations we have tried (just so we can count them!)
        n_iter = 0
        # Number of iterations before we give up (important, so we don't have an infinite loop
        max_iter = 10000000

        # Our searching loop
        while searching:
            n_iter += 1
            # Get a random x to guess, within our range
            x = random.uniform(input_min, input_max)
            # Check if our guess gives us a 'good enough' value
```

```
            if func_1(x) < threshold:
                # Yay! We have an acceptable input.
                searching = False
                success = True
            if n_iter >= max_iter:
                searching = False
                success = False

        if success:
            print('Acceptable solution found after ' + str(n_iter) + ' iterations.')
            print('Input of ' + str(x) + ' gives output of ' + str(func_1(x)))
        else:
             print('We were unable to find a suitable input!')
```

```
Acceptable solution found after 106580 iterations.
Input of 0.5773795637732904 gives output of 1.847198885052137e-05
```

Homework:

Modify the above code to return the best guess for x it had tried, if it didn't manage to find one that met the threshold!

1.1 2-D Gradient Descent

This code is a SUPER BASIC gradient descent. It 'looks around' it's current position, and steps to the best looking option. The purpose of this code is to show you the basic structure of a descent method - it is a pretty terrible implementation.

The quality of implementation you can use depends on weather or not your function is differentiable, and how smooth the output regime is.

```
In [12]: # Define a new function that we will minimize. This one takes two inputs.
         def func_2(x):
             # x is a variable with two values; x[0] is the first, and x[1] is the second.
             return(x[0]**2 + 4*x[1]**2 - x[0]*x[1] - 4*x[0] + 3)
```

```
In [16]: init_guess = [5, 100]
         jump_size = 0.001
         max_iter = 1000000
         n_iter = 0

         search = True

         x = init_guess
         while search:
             n_iter += 1

             # Generate our new guesses
             left_x = np.asarray((x[0] - jump_size, x[1]))
             right_x = np.asarray((x[0] + jump_size, x[1]))
```

```
            up_x = np.asarray((x[0], x[1]  + jump_size))
            down_x = np.asarray((x[0], x[1]  - jump_size))

            # Jump to the best guess.
            guesses = np.asarray((x, left_x, right_x, up_x, down_x))
            results = np.asarray((func_2(x), func_2(left_x), func_2(right_x), func_2(up_x), fu
            if np.argmin(results) == 0:
                # The current guess is our best position!
                search = False
                success = True
            else:
                x = guesses[np.argmin(results)]
            if n_iter >= max_iter:
                search = False
                success = False

        if success:
            print('Local minimum found after ' + str(n_iter) + ' iterations.')
            print('Input of ' + str(x) + ' gives output of ' + str(func_2(x)))
        else:
            print('We were unable to find the local minimum!')
            print('Best guess: an input of ' + str(x) + ' gives output of ' + str(func_2(x)))
```

```
Local minimum found after 102601 iterations.
Input of [2.133 0.267] gives output of -1.26666600000034
```

Homework:
Have a look at the wikipedia implementation of a gradient descent - it uses the actual function derivative and 'jumps' in proportion to the size of the gradient. Write your own func_2 (and derivative) and implement the wikipedia descent!

1.2 Writing a SCIPY Wrapper

Writing a scipy wrapper for a curve fitting function. Here we will implement the scipy.optimize.leasts_quares method to find the curve parameters that best fit our data. Although we could use curve_fit, doing things this way allows us to specify the equation of our curve arbitrarily - we don't have to fit to one of the common functions (eg exponential, log, lorentzian) available in the curve_fit library.

We will be making use of the **kwargs variable, which allows us to pass variables through nested functions without needing to explicitly pack and unpack them at every level.

BEFORE going through this section, be sure to read the least_squares man page

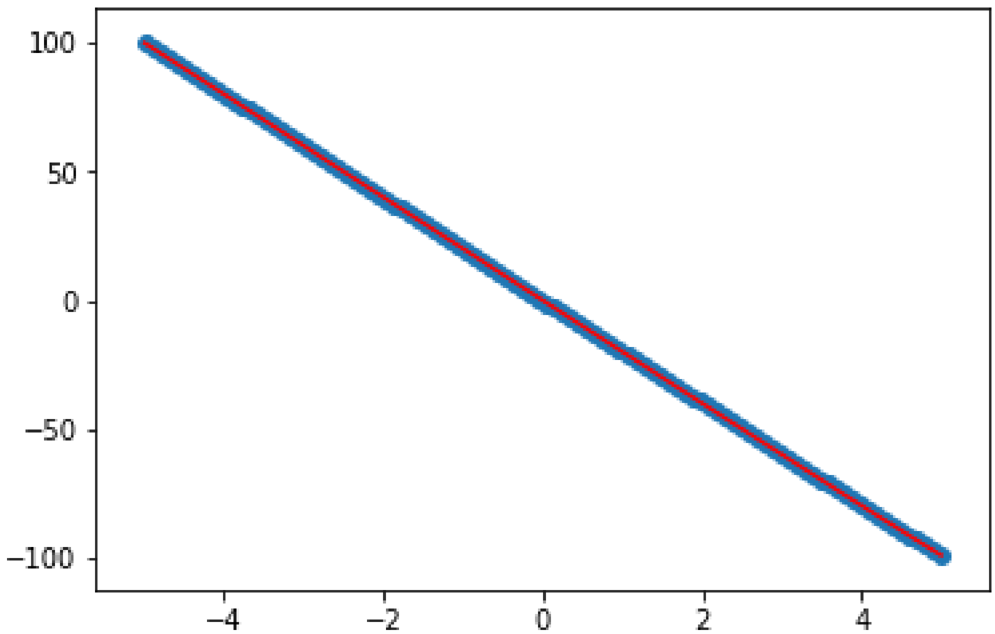
```
In [21]: ## Defining our curve equation (can be as janky as you like, as long as it will NEVER
         def curve(x, b):
             y = x*b - b**2*x + np.exp(x-b)
             return (y)
```

In [18]:
```python
## Definining our SCIPY wrapper for the curve function
def curve_wrapper(b_guess, **kwargs):
    # kwargs should contain our x and y DATA, which we would use to evaluate how good
    x = kwargs.get('x')
    y = kwargs.get('y')

    # Calculate the residuals; difference between our guess-curve's output and the ac
    residuals = curve(x, b_guess) - y
    return residuals
```

In [23]:
```python
## Implementing a SCIPY fitting routine
# Generate some x and y data first
b_true = 2
x = np.linspace(-5,5,100)
y = curve(x, b_true)
# Get scipy to guess what our value of 'b' was, given only the x and y data
# Generate the kwargs dictionary to be passed through the optimisation function.
kwargs = {
    "x":x,
    "y":y
}
b_guess = scipy.optimize.least_squares(curve_wrapper, x0=1, bounds=(-100,100), kwargs=
print('Guess for b is ' + str(b_guess))

# Plot the output
plt.figure()
plt.plot(x, curve(x, b_guess), 'r')
plt.scatter(x,y)
```

```
Guess for b is [5.]
```

Out[23]:
```
<matplotlib.collections.PathCollection at 0x1ef86183ef0>
```



Homework:
Add some random noise to the initial data and see how the curve fit performs!