Assignment 2

Brett Bonine
ASTR 5900: Numerical Methods

February 22, 2021

Problem 1

Derive the Newton-Raphson method for systems of non-linear equations. You can just follow my in-class derivation, but consider x andf (x) as vectors. ˇSirca Horvat explain how to handle the derivative.

Answer: The procedure for finding the roots of a function f via the Newton-Raphson method can be summarized as an interative process that starts with an intial guess, computes the slope of the line tangent to f at that guess, and then repeats the process taking the solution of the tangent line to f as the new guess. For the the scalar case discussed in class, we know that we can Taylor expand the function f according to

$$f(x) = f(x_n) + (x_{n+1} - x_n)f'(x_n) + O(x_{n+1} - x_0)^2 \tag{1}$$

Setting (1) to zero (as is the case for a root) we find, to first order,

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} \tag{2}$$

We'll now explore the the case where $\vec{x}$ and $f(\vec{x})$ are vectors instead of scalars. Following the derivation outlined in Sirca & Horvat, we'll begin by Taylor expanding $\mathbf{f}$ in the vicinity of the root $\vec{\xi}$. In this case, (1) becomes

$$0 = f_i(\vec{x}) + \sum_j \frac{\partial f_i(\vec{x})}{\partial x_j} + O(|\vec{\xi} - x|^2) \tag{3}$$

This can be expressed in terms of the Jacobian $J(\vec{x})$ as

$$0 = f_i(\vec{x}) + J(\vec{x})(\vec{\xi} - x) + ... \tag{4}$$

The Jacobian depends on the various partial derivatives for the different components of our vector. This gives us a recurssion relation that serves as our analogue to (2):

$$\boxed{\vec{x_{k+1}} = \vec{x_k} - J^{-1}(\vec{x_k}).(\vec{\xi} - x) + ...} \tag{5}$$

Problem 2

a) In your chosen compiled language, write a code to solve the equations:

$$sin(x+y) = 0$$
$$cos(x-y) = 0$$

Answer: To solve this system of non-linear equations, we'll follow the vectorized prescription of Newton-Raphson derived above. The Jacboian can be expressed analytically in terms of the derivatives of our system of equations. In accordance with equation (5), we expect solutions of the form:

$$\begin{pmatrix} x_{k+1} \\ y_{k+1} \end{pmatrix} = \begin{pmatrix} x_k \\ y_k \end{pmatrix} - \begin{pmatrix} cos(x_k+y_k) & cos(x_k+y_k) \\ -sin(x_k+y_k) & sin(x_k,+y_k) \end{pmatrix}^{-1} \begin{pmatrix} sin(x_k+y,) \\ cos(x_k-y_k) \end{pmatrix} \tag{6}$$

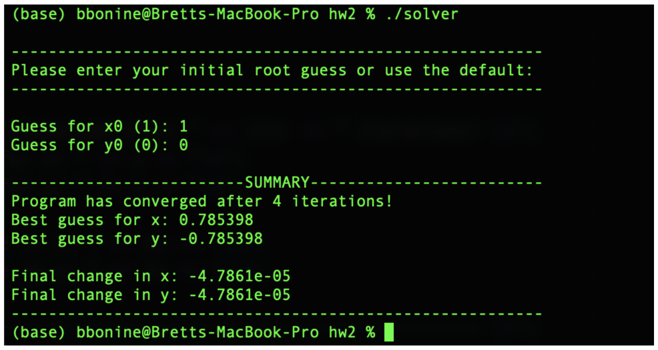
The c++ algorithm **solver.cpp** attempts to solve this system as follows:

- Input a guess for x_0, y_0
- Evaluate separately the values of x_{k+1}, y_{k+1}
- Compare the magnitude of the product term in (6) to the current value of x_k and y_k. If the magnitude of the term is less than some threshold, the program terminates and returns the final guess for the roots.

As suggested in the second point, this algorithm is not vectorized; the entries for x_{k+1}, y_{k+1} are calculated separately according to the matrix multiplication in Fig. 1. As a demonstration of the algorithm, an sample output for a user-specified values of x_0 and y_0 is shown below in Fig. 2.

$$\begin{pmatrix} \cos(x+y) & \cos(x+y) \\ -\sin(x-y) & \sin(x-y) \end{pmatrix}^{-1} \begin{pmatrix} \sin(x+y) \\ \cos(x-y) \end{pmatrix} = \begin{pmatrix} \frac{\sin(x+y)\sin(x-y)-\cos(x-y)\cos(x+y)}{2\cos(x+y)\sin(x-y)} \\ \frac{\sin(x+y)\sin(x-y)+\cos(x-y)\cos(x+y)}{2\cos(x+y)\sin(x-y)} \end{pmatrix}$$

Figure 1: Output from the online matrix multiplication tool on *Symbolab*. The two components on the right were saved as separate functions for use in evaluating x_{k+1} and y_{k+1}, separately.



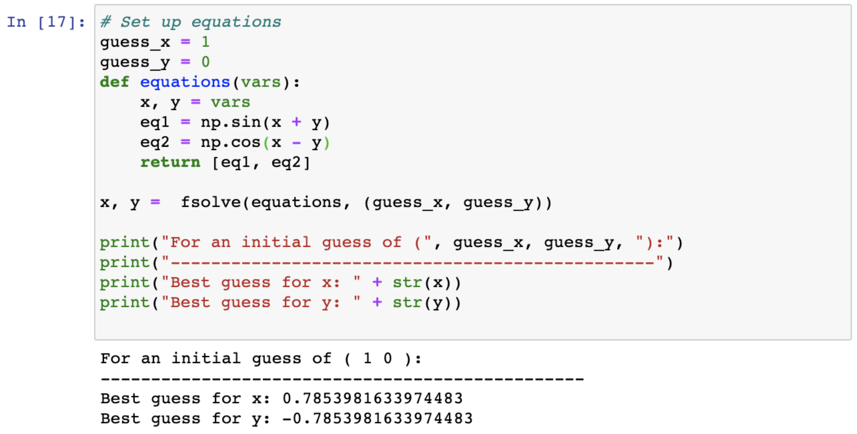
```
(base) bbonine@Bretts-MacBook-Pro hw2 % ./solver

--------------------------------------------------------
Please enter your initial root guess or use the default:
--------------------------------------------------------

Guess for x0 (1): 1
Guess for y0 (0): 0

------------------------SUMMARY-------------------------
Program has converged after 4 iterations!
Best guess for x: 0.785398
Best guess for y: -0.785398

Final change in x: -4.7861e-05
Final change in y: -4.7861e-05
--------------------------------------------------------
(base) bbonine@Bretts-MacBook-Pro hw2 %
```

Figure 2: Raw output for an initial guess input by the user. This demonstrates convergence for an initial guess of $x_0 = 1$, $y_0 = 0$ after only four iterations. As a safeguard against non-convergence, the program will terminate if no solution is reached within 1000 iterations.

The initial guess of (1,0) in Fig.2 was chosen by trial and error. We initially assume a guess of (1,1) since the two equations suggest x=y will likely be solutions, but the $sin(x-y)$ terms evident in the denominator of Fig 1. lead the algorithm to return NaNs for both values. This is likely due to a limitation of the Newton-Raphson method for our particular set of equations (perhaps due to the derivatives for the slope of the tangent line? Another numerical estimate of the roots would probably be required to get around this issue).

b) Check your result using a canned solver, that is something from Mathematica or Python or Mathlab.

Answer: As a test of the performance of the c++ algorithm, we evaluate the same root guesses in the Python script **prob_2b.py** using the scipy method *fsolve*.As shown in Fig 3, our results are consistent with the prepackaged solver.



```
In [17]: # Set up equations
         guess_x = 1
         guess_y = 0
         def equations(vars):
             x, y = vars
             eq1 = np.sin(x + y)
             eq2 = np.cos(x - y)
             return [eq1, eq2]

         x, y =  fsolve(equations, (guess_x, guess_y))

         print("For an initial guess of (", guess_x, guess_y, "):")
         print("-----------------------------------------------")
         print("Best guess for x: " + str(x))
         print("Best guess for y: " + str(y))

         For an initial guess of ( 1 0 ):
         ------------------------------------------------
         Best guess for x: 0.7853981633974483
         Best guess for y: -0.7853981633974483
```

Figure 3: Output from a canned solver for the same initial guess as in Fig.2. This demonstrates the accuracy of our algorithm compared to existing solvers.

It's worth noting that this prepackaged solver does not struggle finding roots with an initial guess of (x, y = -x) like the c++ algorithm we described above. According to the documentation of *fsolve*, this prepackaged solver makes use of the 'modified Powell' method. Perhaps this technique avoids the issue with diverging denominator when x = -y.