

# Assignment 2: Results/Description

## Problem 1

Let  $\{f_i(\mathbf{x})\}$  be the system of nonlinear equations for which we wish to find a root. Then we may introduce  $\mathbf{f}(\mathbf{x})$  comprised of components  $f_i$  that satisfies

$$\mathbf{f}(\mathbf{x}) = \mathbf{0}.$$

Let  $\boldsymbol{\xi}$  be the root of  $\mathbf{f}(\mathbf{x})$ . Then one may Taylor expand around  $\boldsymbol{\xi}$  so that

$$\begin{aligned} 0 &= f_i(\boldsymbol{\xi}) \\ &= f_i(\mathbf{x}) + \sum_j (\xi_j - x_j) \partial_j f_i(\mathbf{x}) + \mathcal{O}(|\boldsymbol{\xi} - \mathbf{x}|^2) \\ \Rightarrow \mathbf{0} &\approx \mathbf{f}(\mathbf{x}) + \mathbf{J}(\mathbf{x})(\boldsymbol{\xi} - \mathbf{x}) \end{aligned}$$

for  $J_{ij} = \partial_j f_i(\mathbf{x})$ . Here we assume we may neglect the quadratic and higher order terms, assuming  $|\boldsymbol{\xi} - \mathbf{x}|$  is small. Therefore, rearranging this gives

$$\boldsymbol{\xi} \approx \mathbf{x} - \mathbf{J}^{-1}(\mathbf{x})\mathbf{f}(\mathbf{x}),$$

assuming  $\mathbf{J}$  is invertible. Using iteration to make this approximation converge to the true value of  $\boldsymbol{\xi}$ , this means

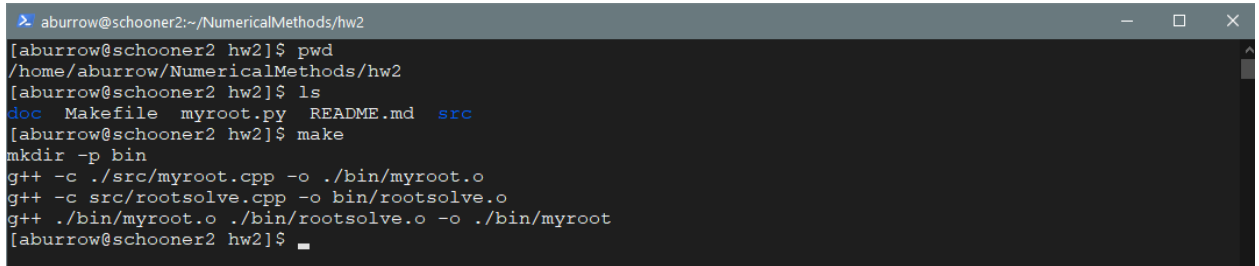
$$\mathbf{x}_{k+1} = \mathbf{x}_k - \mathbf{J}^{-1}(\mathbf{x}_k)\mathbf{f}(\mathbf{x}_k).$$

for iteration index  $k$ .

## Problem 2

(a)

I've written this program to solve the system of equations, which is the attached “./bin/myroot”. This should compile with `make` as shown below.

A terminal window with a dark background and light text. The prompt is 'aburrow@schooner2:~/NumericalMethods/hw2'. The user enters 'pwd' and the output is '/home/aburrow/NumericalMethods/hw2'. Then 'ls' is entered, showing 'doc Makefile myroot.py README.md src'. Finally, 'make' is entered, and the output shows the creation of a 'bin' directory and the compilation of 'myroot.o' and 'rootsolve.o' into 'myroot' using g++.

```
aburrow@schooner2:~/NumericalMethods/hw2
[aburrow@schooner2 hw2]$ pwd
/home/aburrow/NumericalMethods/hw2
[aburrow@schooner2 hw2]$ ls
doc  Makefile  myroot.py  README.md  src
[aburrow@schooner2 hw2]$ make
mkdir -p bin
g++ -c ./src/myroot.cpp -o ./bin/myroot.o
g++ -c src/rootsolve.cpp -o bin/rootsolve.o
g++ ./bin/myroot.o ./bin/rootsolve.o -o ./bin/myroot
[aburrow@schooner2 hw2]$
```

This program makes use of the Newton-Raphson method derived in Problem 1. Here I have analytically found  $\mathbf{J}(\mathbf{x})$  with  $\mathbf{x} = (x, y)^T$  to be

$$\mathbf{J}(\mathbf{x}) = \begin{pmatrix} \cos(x + y) & \cos(x + y) \\ -\sin(x - y) & \sin(x - y) \end{pmatrix}$$

and for the sake of computation speed in the program I use the inverse,

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

The main problem here is determining the initial approximation  $\mathbf{x}_0$ .  $x$  and  $y$  need to be such that one is between 0 and  $\pi$ , while the other is between  $-\pi$  and 0, at least I have found through basic testing. This is because in the iteration step, if  $\mathbf{x}_0$  is not on the same scale as outputs to the periodic sin and cos functions, it attempts to overcorrect and fails to converge to a single result.

In addition, there are 2 roots that this program finds with initial conditions within these bounds. With  $\mathbf{x}_0 = (-1, 1)^\top$  for example, where  $|x|, |y| < \pi/2$ , the program yields two roots which are effectively  $x = \pi/4$  and  $y = -\pi/4$ :

```

aburrow@schooner2:~/NumericalMethods/hw2
[aburrow@schooner2 hw2]$ pwd
/home/aburrow/NumericalMethods/hw2
[aburrow@schooner2 hw2]$ ls
bin  config  doc  Makefile  myroot.py  README.md  src
[aburrow@schooner2 hw2]$ ./bin/myroot
Reading from parameter file: ./config/params
Solving for the root with X0 = [-1, 1]...
-0.77117122281985706, 0.77117122281985706
-0.78540200412904826, 0.78540200412904826
-0.78539816339744828, 0.78539816339744828
-0.78539816339744828, 0.78539816339744828
Found root with 4 iterations
root: -0.78539816339744828, 0.78539816339744828
[aburrow@schooner2 hw2]$

```

When  $\pi/2 < |x|, |y| < \pi$ , where in this case  $\mathbf{x}_0 = (-2, 2)^\top$ , the other two roots ( $x = 3\pi/4$  and  $y = -3\pi/4$ ) are found:

```

aburrow@schooner2:~/NumericalMethods/hw2
[aburrow@schooner2 hw2]$ pwd
/home/aburrow/NumericalMethods/hw2
[aburrow@schooner2 hw2]$ ls
bin  config  doc  Makefile  myroot.py  README.md  src
[aburrow@schooner2 hw2]$ ./bin/myroot
Reading from parameter file: ./config/params
Solving for the root with X0 = [-2, 2]...
-2.4318455772253085, 2.4318455772253085
-2.3556118776621502, 2.3556118776621502
-2.3561944904560255, 2.3561944904560255
-2.3561944901923448, 2.3561944901923448
-2.3561944901923448, 2.3561944901923448
Found root with 5 iterations
root: -2.3561944901923448, 2.3561944901923448
[aburrow@schooner2 hw2]$

```

Numerically, we find these roots to be

$$\begin{pmatrix} -0.78539816339744828 \\ 0.78539816339744828 \end{pmatrix}$$

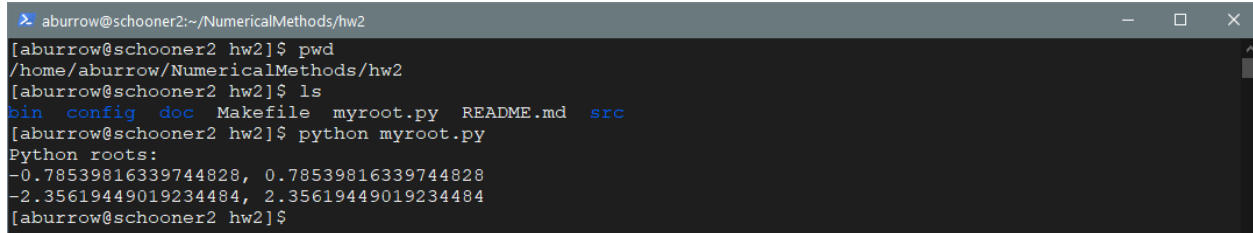
and

$$\begin{pmatrix} -2.3561944901923448 \\ 2.3561944901923448 \end{pmatrix}.$$

Each estimate of the root is shown, and a final value is reached when the difference between the new and old estimates are machine zero.

(b)

I solved this problem in Python as well (“./myroot.py”), using SciPy’s `scipy.optimize.fsolve` function. Using the same two sets of initial conditions, it was able to solve the same roots to the same (double) precision:

A terminal window with a dark background and light text. The window title is 'aburrow@schooner2:~/NumericalMethods/hw2'. The terminal shows the following commands and output:

```
[aburrow@schooner2 hw2]$ pwd
/home/aburrow/NumericalMethods/hw2
[aburrow@schooner2 hw2]$ ls
bin  config  doc  Makefile  myroot.py  README.md  src
[aburrow@schooner2 hw2]$ python myroot.py
Python roots:
-0.78539816339744828, 0.78539816339744828
-2.35619449019234484, 2.35619449019234484
[aburrow@schooner2 hw2]$
```

Typically Python can be used to perform the same calculations for these simple problems (as we saw, it only takes 5 iterations to find a decent result). This is true especially when you are able to vectorize the problem so that one may take advantage of Numpy’s C-optimized functionality. However its flexibility makes it quite burdensome to perform heavy calculations with more complicated problems.