AMS 209: Homework 3 Report

Due: *October 30, 2017*

Instructor: Dongwook Lee

Panos Karagiannis

Contents

Question 1

Answer:
We build a website using Sphinx. The url can be found at:
`https://people.ucsc.edu/`∼ `pkaragia/`.

Question 2

Answer:

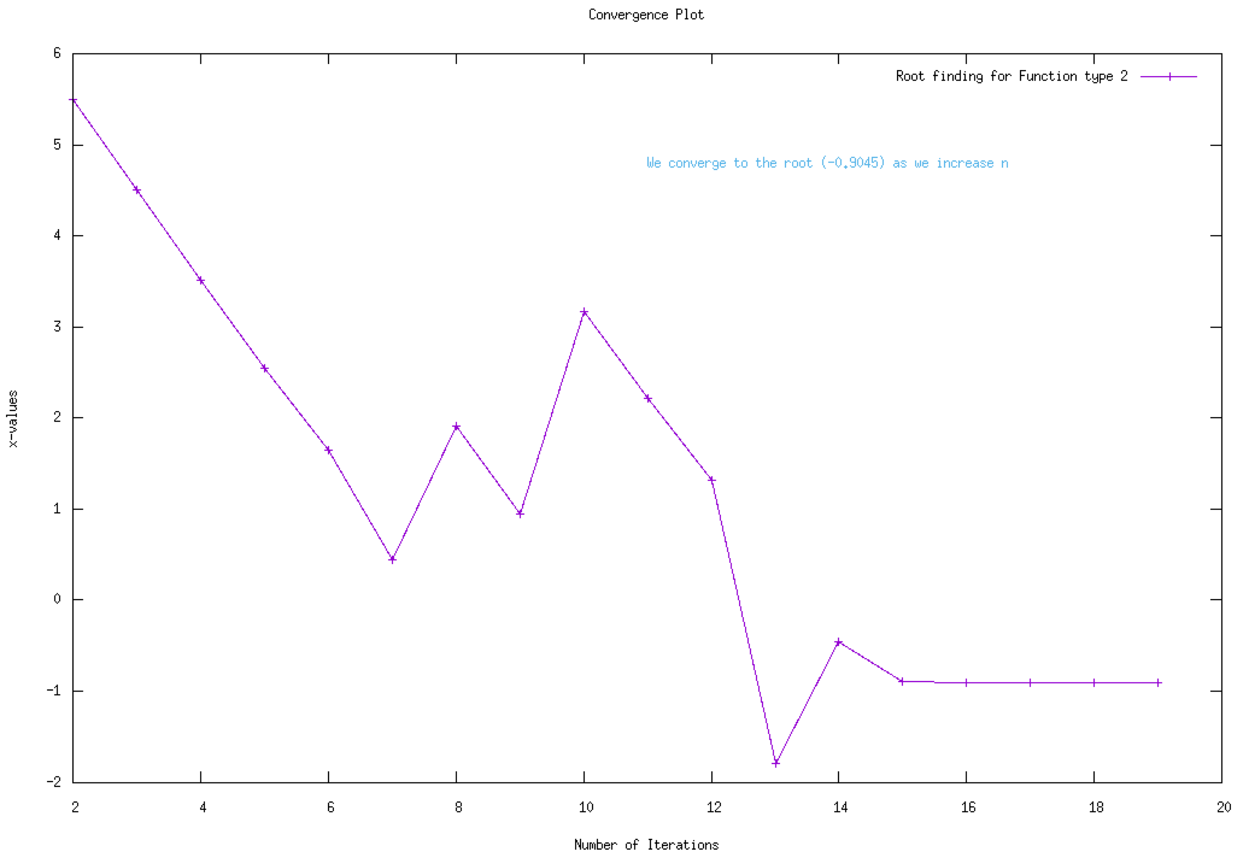
(Q. 1) For this exercise we simply have to import the `ftntype` variable from the `setup_module.F90` and print that (using an if statement) along with `xInit` variable in the `RootFinder.F90` file.

(Q. 2) Here we need to insert print statements in the `setup_module.F90` for all the variables included in the file (except `xInit` which is printed in question 1)

(Q. 3) We produce a plot for function *type 2* and we observe how the root converges to the actual solution which is approximately -.0904563. To plot the data we use `gnuplot` and a script `plot.gnu`:

```
gnuplot -p plot.gnu
```

Question 2 continued on next page. . .



(Q. 4) *No* , we do not need to recompile the code everytime we change the init parameters because we read the input when we run the executable. This is also verified by the fact that the rootFinder.init file appears nowhere in the Makefile.

(Q. 5) We define a new function to be $e^{2x+3} - e^x$. Clearly the root occurs when $x = -3$. Starting with initial parameters `xInit = -2.7`, after six iterations *newton's* method gets close to the root of the function.

The second function we define is $x^2 + 4x + 4$. Clearly $x^2 + 4x + 4 = (x + 2)^2$, so the root occurs when $x = -2$. Starting with initial parameters `xInit = -0.4` after 29 iterations, *newton's* method, gets close to the root of the function.

(Q. 6) If we change the `definition.h` file then we would need to recompile the code in order to observe the effect of the changes we made. Without recompiling, we see that the program *does* work properly since the header file is not accessed at runtime.

(Q. 7) No different behavior is seen after we delete `newton` and `modified_newton`. This is because we have pre-assigned values to these variables in our code.

(Q. 8) If we use make debug, the `FFLAGS_DEBUG` defined in the makefile, will be used during compilation. This will allow us to catch potential bugs in our code.

Question 3

Answer:
For this question I followed closely the instructions given in the exercise. The only point where my solution deviates from the exercise is that I decided not to use the `pi_errorCheck.F90` file since computing the error requires only a single line of code. Hence, I included this functionality in the file `pi_module.F90` inside the function `pi_summation`. To compile and run the program use the makefile:

```
make clean && make && ./pi_approx.exe
```

Question 4

Answer:

(Q. 1) In this exercise, we initialize an array `x` but then fail to deallocate it:

```
gfortran -g -Wall -Wextra -Wimplicit-interface -fPIC -fmax-errors=1
    -fcheck=all -fbacktrace buggy_code_1.f90 -o myprog1
```

We then use *valgrind* by typing:

```
valgrind --leak-check=full --dsymutil=yes --track-origins=yes ./myprog1
    2> output_1
```

We see the error occuring on line 37 of the `buggy_code_1.f90`.

(Q. 2) Again we use the same array `x(1:10)` as before, but now we initialize some values of the array but not `x(8)`. Then we attempt to print the array `x` and *valgrind* finds out that a value in the array has not been initialized. Similarly we use:

```
gfortran -g -Wall -Wextra -Wimplicit-interface -fPIC -fmax-errors=1
    -fcheck=all -fbacktrace buggy_code_2.f90 -o myprog2
```

We then use *valgrind* by typing:

```
valgrind --leak-check=full --dsymutil=yes --track-origins=yes ./myprog2
    2> output_2
```

(Q. 3) In this case we simply attempt to access the array `x` after it has been freed. Again we use similar commands to compile and use valgrind:

Question 4 continued on next page...

```
gfortran -g -Wall -Wextra -Wimplicit-interface -fPIC -fmax-errors=1
        -fcheck=all -fbacktrace buggy_code_3.f90 -o myprog3
```

We then use *valgrind* by typing:

```
valgrind --leak-check=full --dsymutil=yes --track-origins=yes ./myprog3
                              2> output_3
```

(Q. 4) We notice that valgrind fails to produce lines of error when we allocate an array `x` and then set `x(1) = x(5)` :

```
gfortran -g -Wall -Wextra -Wimplicit-interface -fPIC -fmax-errors=1
        -fcheck=all -fbacktrace buggy_code_3.f90 -o myprog4
```

To use valgrind we type:

```
valgrind --leak-check=full --dsymutil=yes --track-origins=yes ./myprog3
                              2> output_4
```