

The Cube Roots of Unity

David Arnold

February 25, 2003

Abstract

Newton's Method can find complex roots of polynomials as easily as it can find the real roots. In this activity, we use Newton's Method to examine the *basin of attraction* for one of the cube roots of unity, which leads to an amazing fractal structure.

Introduction

We will be discussing solutions of $z^3 = 1$, commonly called the *cube roots of unity*. This equation has one real solution, $z = 1$, but it also has two complex solutions, which we will determine shortly. In order to calculate these complex solutions, we first need to develop a relation called *Euler's Identity*.

Euler's Identity

If we define $\sqrt{-1} = i$, it is a simple matter to deduce that $i^2 = -1$.¹ Once this is established, it is a small step to deduce that

$$i^3 = -i \quad \text{and} \quad i^4 = 1.$$

After these introductory calculations, we quickly see that the pattern repeats itself in blocks of four. That is,

$$\begin{aligned} i^5 &= i & i^9 &= i \\ i^6 &= -1 & i^{10} &= -1 \\ i^7 &= -i & i^{11} &= -i \\ i^8 &= 1 & i^{12} &= 1, \end{aligned} \tag{1}$$

etc.

Every calculus student, and sometimes a lucky trigonometry student, is familiar with the infinite series representation for e^x , $\cos x$, and $\sin x$. Indeed,

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \frac{x^5}{5!} + \cdots \tag{2}$$

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots \tag{3}$$

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots \tag{4}$$

¹For a more careful development of the theory of complex numbers, you might want to consider taking a course in complex variables.

We will find an expression for $e^{i\theta}$ by substituting $i\theta$ into the infinite series (2).

$$e^{i\theta} = 1 + i\theta + \frac{(i\theta)^2}{2!} + \frac{(i\theta)^3}{3!} + \frac{(i\theta)^4}{4!} + \frac{(i\theta)^5}{5!} + \frac{(i\theta)^6}{6!} + \frac{(i\theta)^7}{7!} + \frac{(i\theta)^8}{8!} + \dots \quad (5)$$

Using (1), this becomes

$$e^{i\theta} = 1 + i\theta - \frac{\theta^2}{2!} - \frac{i\theta^3}{3!} + \frac{\theta^4}{4!} + \frac{i\theta^5}{5!} - \frac{\theta^6}{6!} - \frac{i\theta^7}{7!} + \frac{\theta^8}{8!} + \dots \quad (6)$$

Separating this into real and imaginary parts,

$$e^{i\theta} = \left(1 - \frac{\theta^2}{2!} + \frac{\theta^4}{4!} - \frac{\theta^6}{6!} + \frac{\theta^8}{8!} - \dots\right) + i\left(\theta - \frac{\theta^3}{3!} + \frac{\theta^5}{5!} - \frac{\theta^7}{7!} + \dots\right). \quad (7)$$

However, by (3) and (4),

$$\begin{aligned} \cos \theta &= 1 - \frac{\theta^2}{2!} + \frac{\theta^4}{4!} - \frac{\theta^6}{6!} + \frac{\theta^8}{8!} - \dots \\ \sin \theta &= \theta - \frac{\theta^3}{3!} + \frac{\theta^5}{5!} - \frac{\theta^7}{7!} + \dots \end{aligned}$$

Therefore, equation (7) becomes the world famous *Euler's Identity*.

$$e^{i\theta} = \cos \theta + i \sin \theta \quad (8)$$

The Geometry of Complex Numbers

The complex number $a + bi$ can be represented by a vector in the complex plane, as shown in Figure 1. The magnitude of the complex number (length of the vector) is denoted by r , and by the

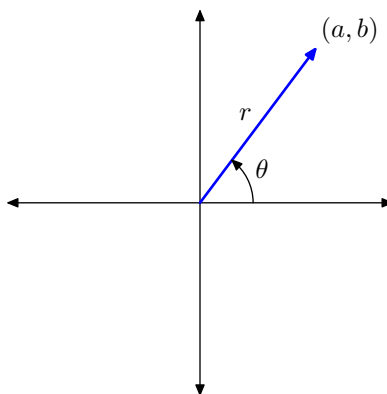


Figure 1: Complex numbers as vectors in the complex plane.

Pythagorean Theorem, is easily seen to equal

$$r = \sqrt{a^2 + b^2}. \quad (9)$$

The argument of the complex number is the angle made with the positive horizontal axis (counterclockwise is the positive direction).

Now, suppose that we start with an arbitrary complex number

$$z = a + bi,$$

then factor out its magnitude as follows.

$$z = \sqrt{a^2 + b^2} \left(\frac{a}{\sqrt{a^2 + b^2}} + i \frac{b}{\sqrt{a^2 + b^2}} \right) \quad (10)$$

Referring again to Figure 1, note that

$$\begin{aligned} \cos \theta &= \frac{a}{\sqrt{a^2 + b^2}} \\ \sin \theta &= \frac{b}{\sqrt{a^2 + b^2}}. \end{aligned} \quad (11)$$

Thus, substituting (9) and (11) into equation (10), we arrive at the result

$$z = re^{i\theta}, \quad (12)$$

where r is the magnitude of the complex number and θ is the angle made by the vector in Figure 1 with the positive x -axis. This form is called the *trigonometric form* of the complex number.

The Cube Roots of Unity

Let us return now to the problem of finding all solutions of the equation

$$z^3 = 1. \quad (13)$$

It is easy to sketch the vector representing the number 1 in the complex plane (see Figure 2). Note

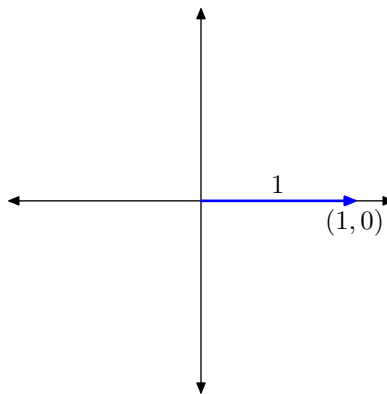


Figure 2: The vector representing the complex number 1.

that the length is $r = 1$ and we can choose any multiple of 2π for the angle made with the positive x -axis. Let's set $\theta = 2k\pi$, where k is some arbitrary integer. Hence, written in trigonometric form,

$$1 = 1e^{i2k\pi}. \quad (14)$$

Now, the z in equation (13) is unknown. We don't know its length and we don't know its argument. Therefore, as both r and θ are unknown, let

$$z = re^{i\theta}. \quad (15)$$

Substitute (15) and (14) into the left and right sides of equation (13), respectively, obtaining

$$(re^{i\theta})^3 = 1e^{i2k\pi}. \quad (16)$$

Using the laws of exponents, this becomes

$$\begin{aligned} r^3(e^{i\theta})^3 &= 1e^{i2k\pi} \\ r^3(e^{i3\theta}) &= 1e^{i2k\pi} \end{aligned} \quad (17)$$

Comparing left and right hand sides of this last equation, we note that

$$r^3 = 1 \quad \text{and} \quad 3\theta = 2k\pi,$$

or, equivalently,

$$r = 1 \quad \text{and} \quad \theta = \frac{2k\pi}{3}.$$

Substituting these values of r and θ into (15), the solutions of equation (13), the cube roots of unity, are given by

$$z = 1e^{i2k\pi/3}. \quad (18)$$

Letting $k = 0, 1,$ and $2,$ we get three solutions,

$$\begin{aligned} z_0 &= 1e^{i0}, \\ z_1 &= 1e^{i2\pi/3}, \text{ and} \\ z_2 &= 1e^{i4\pi/3}. \end{aligned}$$

Note that each of these roots of unity have magnitude or length 1. Therefore, the vectors representing these roots of unity must lie on the unit circle. Finally, note that the angles divide the unit circle into three equal sectors, at $\theta = 0, 2\pi/3,$ and $4\pi/3,$ as shown in Figure 3.

Newton Again

Recall that to use Newton's method, we start with an initial guess, say $x_0,$ then we iterate according to the equation

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}, \quad (19)$$

which produces a sequence of numbers that hopefully converge to a root of the function $f.$

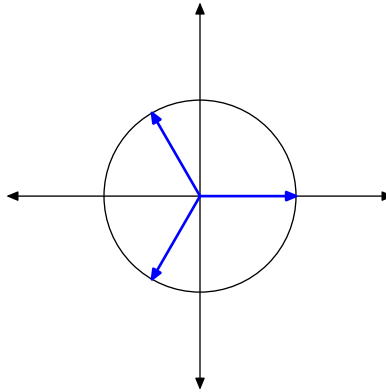


Figure 3: The three cube roots of unity.

If f is a function of a complex variable z , Newton's Method works equally well. We start with an initial guess z_0 , which is complex, then we iterate according to the equation

$$z_{n+1} = z_n - \frac{f(z_n)}{f'(z_n)}. \quad (20)$$

In our case, we wish to find the cube roots of unity, so we seek roots of the function

$$f(z) = z^3 - 1,$$

having first derivative

$$f'(z) = 3z^2.$$

The code for this algorithm is all too familiar.

```

module newton_mod

  public :: f, fp

contains

  function f(z) result (w)
    complex, intent(in) :: z
    complex :: w
    w=z**3-1
  end function f

  function fp(z) result (w)
    complex, intent(in) :: z
    complex :: w
    w=3*z**2
  end function fp

```

```

end module newton_mod

program newton
  use newton_mod
  integer, parameter :: NUM_ITERATIONS=10
  complex :: z
  integer :: i
  print *, "Enter an initial guess in the form (x,y): "
  read *, z
  do i=1, NUM_ITERATIONS
    print *, z
    z=z-f(z)/fp(z)
  end do
end program newton

```

There is little new here that bears explaining other than the syntax for declaring a complex variable. For example,

```
complex :: z
```

is used to declare z as a complex variable.

After saving this file as `newton.f90` and compiling with `F -o newton newton.f90`, running the program with an initial guess $(-2, -2)$ yields the following output.

```

Enter an initial guess in the form (x,y):
(-2,-2)
(-2.0000000,-2.0000000)
(-1.3333333,-1.3750000)
(-0.8916841,-1.0074898)
(-0.6168304,-0.8544437)
(-0.5057055,-0.8545201)
(-0.4998346,-0.8660440)
(-0.5000000,-0.8660254)
(-0.5000000,-0.8660254)
(-0.5000000,-0.8660254)
(-0.5000000,-0.8660254)

```

Note that using $(-2, -2)$ as the initial guess results in a sequence of complex numbers that converge to the complex number $-1/2 - i\sqrt{3}/2$, which in trigonometric form is our friend $1e^{i4\pi/3}$.

Try running the program again with different initial guesses. For example, try $2 + 2i$, which you enter at the prompt as $(2, 2)$. Experiment with other initial guesses until you find initial guesses that converge to each of the three roots pictured in Figure 3.

Can you explain the following run?

```

Enter an initial guess in the form (x,y):
(0,0)
(0.0000000E+00,0.0000000E+00)
*** Arithmetic exception: - aborting

```

Why does this happen? How can you adjust your code to prevent this from happening with this initial guess?

Basin of Attraction

In the previous section, we discovered that an initial guess will converge to one of three different roots of unity. In this next exercise, rather than iterate a single guess, we are going to iterate each point in a grid of points in the complex plane. Those initial guesses that produce a sequence of complex numbers that converge to 1 will be colored white, otherwise they will be painted black.

For example, start by superimposing an eight by eight grid of initial guesses on the complex domain

$$D = \{(x, y) : -2 \leq x \leq 2 \text{ and } -2 \leq y \leq 2\}, \quad (21)$$

as shown in Figure 4. The idea is to iterate each of the initial guesses shown in the grid in Figure 4.

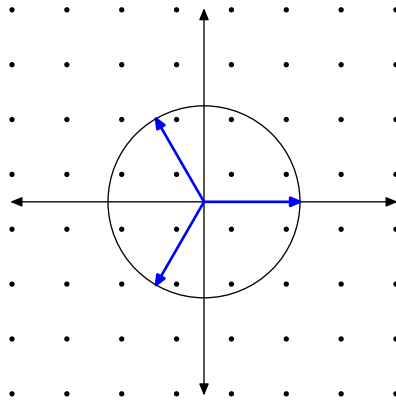


Figure 4: An 8×8 grid of initial guesses.

Those that produce a sequence of complex numbers that converge to 1 will be colored white, the others will be painted black.

The Program

Our module will be almost identical to the module used in program `newton`.

```
module cuberoot_mod

  integer, public, parameter :: MAX_ITERATIONS=25
  public :: f, fp

contains

  function f(z) result (w)
    complex, intent(in) :: z
    complex :: w
    w=z**3-1
  end function f

  function fp(z) result (w)
```

```

    complex, intent(in) :: z
    complex :: w
    w=3*z**2
end function fp

```

```
end module cuberoot_mod
```

Note again how we have defined the function $f(z) = z^3 - 1$ and its derivative $f'(z) = 3z^2$ exactly as we did before. However, there is one different line in this module.

```
integer, public, parameter :: MAX_ITERATIONS=25
```

The parameter `MAX_ITERATIONS` is set to 25. Each initial guess will be iterated 25 times using Newton's Method. The last iterate will be compared with the root 1. If we are within a certain distance of this root, our initial guess will be colored white, otherwise we will paint it black.

Our main program begins in the usual manner.

```

program cuberoot
  use cuberoot_mod
  integer, parameter :: SIZE = 512
  real, parameter :: TOL=1.0e-3
  real :: a, b, c, d, x, y, dx, dy
  complex :: z
  integer :: i, j, iter

```

We “use” the module `cuberoot_mod`, then proceed to declare the type of parameters and variables.

- Rather than using an 8×8 grid of initial guesses, we intend to use a 512×512 grid. This will provide us with a final image having dimensions 512 pixels wide and 512 pixels high.
- The real parameter `TOL` will be used to check if our sequence is within a certain distance of the root 1.
- The variables `a` and `b` contain the minimum and maximum x -values of the grid. Similarly, `c` and `d` contain the minimum and maximum y -values of the grid.
- The variables `x` and `y` will contain the real and imaginary parts of the complex number $z = x + iy$.
- The variables `dx` and `dy` contain the incremental distances between consecutive points in the grid, in the x and y -directions, respectively.
- The variable `z` contains the complex number $z = x + iy$.
- The integer variables `i` and `j` are counters which are used to help cycle through the grid of initial guesses. They will also index the matrix containing the final color of each grid point.
- The integer variable `iter` will track how many times we iterate Newton's equation (20).

The next line of code introduces a new data type called an *array*.

```
integer, dimension(SIZE,SIZE) :: M
```

This syntax declares `M` to be a two dimensional integer array. `M` is a 512×512 matrix whose entries are zeros and ones, one for each initial guess that produces a sequence that converges to 1, zero otherwise. It is this matrix that holds the color of each pixel of our final picture.

Next, we initialize variables.

```
a=-2.0
b=2.0
c=-2.0
d=2.0
dx=(b-a)/(SIZE-1)
dy=(d-c)/(SIZE-1)
```

Some of this initialization is obvious, some not.

- Clearly, `a=-2.0`, `b=2.0`, `c=-2.0`, and `d=2.0` initializes the grid defined in (21) and shown in Figure 4.
- The statement `dx=(b-a)/(SIZE-1)` is a bit tricky. Suppose that we wish to plot 8 points on the interval $[a, b]$, with the first point at a and the final point at b . A little careful thinking reveals that the distance between consecutive points is found by dividing the total distance between a and b by 7, not 8. Think of 8 equally spaced pickets on a picket fence. How do you calculate the distance between them? A diagram will help you sort this out. Note that `dy` is calculated in a similar manner.

Next, we begin the loops that will iterate each point in our grid using Newton's method. But first we set $y = c$ in preparation (our intent is to first use the gridpoints along the bottom row in Figure 4).

```
y=c
rows: do i=1,SIZE
  x=a
  columns: do j=1, SIZE
    z=cplx(x,y)
```

Much explanation is needed here.

- We're starting across the bottom row in Figure 4. Hence, we start with $y = c$. There are `SIZE=512` rows, hence we start the iteration with `rows: do i=1, SIZE`. Note that we have placed a *label* at the start of this `do` loop, which is useful for keeping track of which `do` loop we are currently residing in.
- We want to start at the left end of each grid row in Figure 4, so each time we move up to a new row, we want to set $x = a$ before moving across the new row.
- Another loop begins with `columns: do j=1, SIZE`. This loop has the label `columns` to remind us that we are moving across the row, a column at a time while we are in this inner loop.
- The next line sets the initial guess to $z = x + iy$ with the syntax `z=cplx(x,y)`. It's unfortunate that we couldn't simply do this with the syntax `z=(x,y)`, but that won't work. I haven't

got a satisfactory answer yet from the Fortran gurus as to why this is necessary. After all, in the previous program we were able to enter our guess at the command prompt with $(-2, -2)$, if you recall. Oh well, that's Fortran for ya!

Now that we have our initial guess, we will iterate exactly `MAX_ITERATIONS` times in the next `do` loop.

```
iterate: do iter = 1, MAX_ITERATIONS
    z=z-f(z)/fp(z)
end do iterate
```

After this iteration loop terminates, we want to check to see if our last iterate is close to 1.

```
if (abs(z-1)<TOL) then
    M(i,j)=1
else
    M(i,j)=0
end if
```

If the distance between the last iterate and 1 is less than our prescribed tolerance, then we color the pixel with color 1, otherwise we color the pixel with color zero. Note that the companion matrix `M` is used to record the color of the current gridpoint. The matrix `M` will be used to draw the final picture.

We now want to move over a column to use the grid point immediately to the right of the current grid point, so we increment `x`.

```
x=x+dx
end do columns
```

When the columns `do` loop terminates, we need to increment `y` to move up to the next row.

```
y=y+dy
end do rows
```

Of course, once this last loop completes its cycles, we are done, and matrix `M` will contain the appropriate color to paint each pixel.

Saving Data to a File

We will use Matlab and its Image Processing Toolbox to plot the image produced by the data in matrix `M`. However, if we hope to transfer the data in matrix `M` to Matlab, we will first need to save its contents to a file. So, just how does one write to a file using the F programming language?

First, open a channel to the file with the following command.

```
open(unit=11,file="cuberoot.dat",action="write",status="replace")
```

Some explanation is warranted here.

- You are basically free to use any unit number you please (up to a point). Your system reserves some unit numbers to contact the screen, keyboard, or other devices attached to your system. You are pretty safe using unit numbers above 9. In this case, I've opened a channel to the file system via unit number 11. Henceforth, I will use this number to write to the file.

- You must state the name of the file to which you wish to write the data contained in matrix M. We've used `file="cuberoot.dat"`, but you are free to choose any file name you wish.
- You must declare the "action" you desire. Options are "read", "write", and "readwrite," to name a few. We have no intention of trying to read the contents of the file (we only want to write to the file), so we've set `action="write"`. It must not be "read" if the status is "new" or "replace." More on that in a moment.
- The status determines the fate of your file. Options are "old," "new," "replace," and "scratch." Use `status="old"` when you want to read a file. Use `status="new"` if the file doesn't exist and you wish to create it. It is an error to use "new" if the file already exists. That brings us to `status="replace"`, my personal favorite. With this option, if the file doesn't exist, it is created. If the file exists, it is first destroyed, then recreated anew without any data contained in the new file, a great way to start over and perfect for the task at hand.

Now that we have a channel open to `cuberoot.dat` via unit number 11, we refer to the unit number and not the filename in any subsequent write statements.

```
do j=1,SIZE
  do i=1,SIZE
    write(unit=11,fmt=*) M(i,j)
  end do
end do
```

We simply write a double loop that stores each entry of the matrix M on an individual line in the file `cuberoot.dat`. In the F programming language, any time you execute a write statement, you automatically issue a new linefeed.

Note the order of the do loops. First we fix `j=1`, then iterate over `i`, printing entries of `M(i,j)` one to a line in the file. This means we that we are moving down the first *column* of matrix M as we print entries to the file. Next, we fix `j=2`, then iterate over `i` again, which means we are moving down the second *column* of matrix M as we save entries to the file, one line at a time. We say that the matrix M is saved to the file in *column order*.

Once the writing is completed, we must close the channel to the file.

```
close(11)
```

This is extremely important. Never leave files open. You are in danger of corrupting your data if you do so.

We're done. End the program.

```
end program cuberoot
```

Using Matlab to Draw the Image

Open Matlab. Change to the directory containing your program `cuberoot.f90` and the corresponding data saved in the file `cuberoot.dat`.

```
cd /home/username/fortran/assign6/
```

Import the data into Matlab.

```
importdata("cuberoot.dat");
```

Reshape the matrix (otherwise the data is in one long column).

```
A=reshape(A,512,512);
```

Plot the result with the Image Processing Toolbox command.²

```
imshow(A)
```

This last command should produce the image shown in Figure 5.

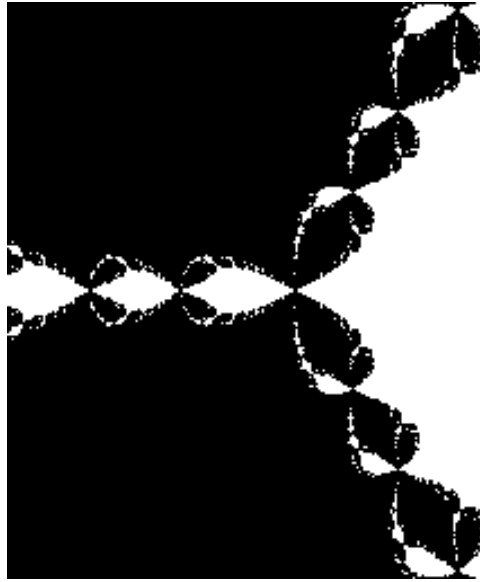


Figure 5: Basin of attraction.

The Assignment

Adjust your program as follows.

1. If an initial guess converges to 1, set $M(i, j)=1$.
2. If an initial guess converges to $-1/2 + \sqrt{3}/2$, set $M(i, j)=2$.
3. If an initial guess converges to $-1/2 - \sqrt{3}/2$, set $M(i, j)=3$.

Use the following Matlab commands to plot the result.

²If you don't have the Image Processing Toolbox on your home system, you can try `image(A)` instead.

```
A=importdata('cuberoot.dat');
A=reshape(A,512,512);
imshow(A)
map=[1,0,0;0,1,0;0,0,1]
colormap(map)
shg
```

Obtain a printout of the resulting image.

The Grading Rubric

The following rules define the rubric I will use to grade your program.

1. (30 points) Will be awarded for adequate comments. Comments should include:
 - (a) A description of the program's purpose.
 - (b) Your name, the date of submission, and the version or revision number.
 - (c) A complete dictionary of all variables and parameters used in the program.³
 - (d) Interprogram comments should proceed any code snippets explained by the comments. These should be adequately sprinkled throughout your code.
2. (50 points) Will be awarded if the program works and does what it was asked to do.
3. (10 points) Will be awarded for good program style. This includes good indentation practices, etc. Proper use of Emacs should insure good programming style.
4. (10 points) Will be awarded for creativity and extra effort. Did you just do the bare minimum? Or did you stretch and reach a little higher? Did you put something cute or clever into your program that nobody else seemed to think of?

Penalties

Each program that is assigned during the term will have a due date. On that date, the program must be on the instructor's desk before the start of class. Penalties will be assessed as follows.

1. (10 points) There will be a 10 point deduction for any program that is handed in after the class has begun.
2. (20 points) There will be a 20 point deduction per class period. That is, if you hand the program in one class period late, there is an automatic 20 point deduction. Two class periods warrants a 40 point deduction, etc. To be clear, if the program is in the instructor's hands before the beginning of the next class, that is a 20 point deduction. If the program is in the instructor's hands before the start of the second class period past the due date, that is a 40 point deduction, etc.

³See <http://www.netlib.org/lapack/single/sgesv.f> for an example of production quality code. This is an excellent example of how professionals comment their code.

Managing Files and Folders

Each of you has been given personal space on the sci-math server to store your work. In Windows, this space is mapped to the drive letter H. If you open the Windows Explorer (the file manager, not the internet browser), you can see that the drive letter has been mapped to your login name. In Linux, your personal space is in your home directory, typically `/home/username`.

In this folder, create a new folder called `fortran`. Note that you must **never** use spaces in filenames. Also note that in the Windows operating system, filenames are not case-sensitive, which is exactly opposite to what happens in Unix and Linux, where filenames are case-sensitive.

In your `/home/username/fortran` folder, create another folder called `assign6`. It is in this folder (`/home/username/fortran/assign6`) that you are to place the source code and executables for this current project. When you receive your next project, create a new folder called `assign7` to hold that project, etc.

If you work at home, I still want you to place copies of your work in the space reserved for you on our system. Simply copy your home files onto a floppy disk and bring them with you to school. In Windows, use the Explorer to copy the files on your disk into the proper folder on your H: drive. In Linux, you have to mount your floppy. Open a terminal window and enter this command.

```
mount /mnt/floppy
```

Make an `assign5` directory.

```
cd /home/username/fortran
mkdir assign6
```

Change to the new directory.

```
cd assign6
```

Copy the file `cuberoot.f90` from your floppy to the current directory with the following command.⁴

```
cp /mnt/floppy/cuberoot.f90 .
```

Note that the dot represents the current directory. Hence, this command copies the file `cuberoot.f90` from your floppy to the current directory.

Finally, you have to unmount your floppy before removing it from the computer. This is the only safe way to make sure all filesystems are closed on the floppy and prevent the corruption of files.

```
umount /mnt/floppy
```

It is now safe to remove the floppy from the disk drive.

If everyone follows these simple rules, I can easily access your work from my office machine for purposes of assigning a grade.

Caveat

On this project, if you stop by my office with hardcopy of your program before the due date of this assignment, I will give a quick glance and critique of your source code. Somewhat like receiving a grade on a draft before submitting your final draft for assessment.

⁴You may have to adjust the path to `cuberoot.f90` if it is in some sort of subfolder on your floppy.

Appendix

We repeat the entire program here as a courtesy.

```
module cuberoot_mod
  integer, public, parameter :: MAX_ITERATIONS=25
  public :: f, fp
contains
  function f(z) result (w)
    complex, intent(in) :: z
    complex :: w
    w=z**3-1
  end function f
  function fp(z) result (w)
    complex, intent(in) :: z
    complex :: w
    w=3*z**2
  end function fp
end module cuberoot_mod
```

```
program cuberoot
  use cuberoot_mod
  integer, parameter :: SIZE = 256
  real, parameter :: TOL=1.0e-3
  real :: a, b, c, d, x, y, dx, dy
  complex :: z
  integer :: i, j, iter
  integer, dimension(SIZE,SIZE) :: M
  a=-2.0
  b=2.0
  c=-2.0
  d=2.0
  dx=(b-a)/(SIZE-1)
  dy=(d-c)/(SIZE-1)
  y=c
  rows: do i=1,SIZE
    x=a
    columns: do j=1, SIZE
      z=cmplx(x,y)
      iterate: do iter = 1, MAX_ITERATIONS
!         if (abs(z)<1.0e-3) then
!           exit iterate
!         end if
        z=z-f(z)/fp(z)
      end do iterate
      if (abs(z-1)<TOL) then
        M(i,j)=1
      end if
    end do
  end do
```

```
        else
            M(i,j)=0
        end if
        x=x+dx
    end do columns
    y=y+dy
end do rows

open(unit=11,file="cuberoot.dat",action="write",status="replace")
do j=1,SIZE
    do i=1,SIZE
        write(unit=11,fmt=*) M(i,j)
    end do
end do
close(11)

end program cuberoot
```