4.5 Subfunctions in Matlab

Up to this point, our programs have been fairly simple, usually taking on a single task, such as solving a quadratic equation. As programs become more complex and perform multiple tasks, modern programmers break programs into a series of self contained modules. In this manner, various parts of the program can be written separately, tested, and once stable, can be integrated into the program as a whole.

In older programming languages, these modules were known as subroutines. In modern, object oriented languages such as C++ and Java, the modules are objects, created from classes, that contain their own variables and methods for manipulating data and performing tasks. These objects can then be reused and assembled to create sophisticated programs.

To attack a complex programming task in Matlab, we must first break the task up into manageable units. Each of these units or modules are then coded as **functions** and tested until we are certain they perform the simple task for which they are designed. Once satisfied that the individual functions are performing as designed, we must then integrate them into a complete and functioning program.

In this section we will discuss two ways that we can accomplish this modular design.

1. We can write each of the modules as functions in separate files. We then write a master script file which calls each of the function M-files as needed.
2. We can write a primary function in place of the script file. We can then incorporate the modules as *subfunctions* directly in this primary function file. That is, we write one self contained file.

Actually there is also a third way to proceed. We can write a primary file in place of a script file. However, by signalling the end of this primary function with the Matlab keyword **end**, we signal Matlab that out subfunctions (each of which must also use the keyword **end**) are *nested functions*. A primary function written in this manner emulates some of the behavior of an object in an object oriented language, with its own variables and methods (subfunctions). We'll see in a later section that the rules for variable scope will change significantly when using nested functions.

We will investigate each of these options in some detail. In the next section, we investigate the first of these options, calling a function M-file from within the body of a script M-file.

---

[1] Copyrighted material. See: http://msenux.redwoods.edu/Math4Textbook/

Calling Functions From Script Files

We are going to tackle a programmng project that will perform rational arithmetic for the user of the program. Options will include reducing, adding, subtracting, multiplying, and dividing fractions input by the user of our program.

Contemplating the design of such a program, here are some possible smaller tasks that need implmentation.

1. Present the user with a menu of choices: reduction, addition, subtraction, multiplication, and division.
2. Routines will need to be written for each possible user choice on the menu.
3. The program will need to query the user for a menu choice, then react in the appropriate manner.

Note that this design is preliminary. In the process of writing a complex program, difficulties with the original design will surely arise. When that occurs, we will need to make modifications of the original design, which in turn will create the need for design of new modules not considered in the first design.

However, one clear place that we can start is on the design of a menu of choices for the user of our program. Remember, the menu should query what action the user wants to perform, reduction, addition, subtraction, or division. With this thought in mind, we write the following lines and save the script as **rationalArithmetic.m**.

```
clc
fprintf('Rational Arithmetic Menu:\n\n')
fprintf('1). Reduce a fraction to lowest terms.\n')
fprintf('2). Add two fractions.\n')
fprintf('3). Subtract two fractions.\n')
fprintf('4). Multiply two fractions.\n')
fprintf('5). Divide two fractions.\n')
fprintf('6). Exit program.\n\n')
n=input('Enter number of your choice: ');
```

Go to the Matlab prompt in the command window and execute the script by typing the following.

```
>> rationalArithmetic
```

The script clears the command window, puts up a menu of choices, then queries the user for a response. The script accepts the user response from the keyboard and saves it in the variable n. Because we haven't programmed it to do anything else yet, the script takes no further action and simply exits.

```
Rational Arithmetic Menu:

1). Reduce a fraction to lowest terms.
2). Add two fractions.
3). Subtract two fractions.
4). Multiply two fractions.
5). Divide two fractions.
6). Exit program.

Enter number of your choice: 1
```

Now, let's think modularly. Putting up a menu is a task that will be repeated over and over as the user employs our program to perform rational arithmetic. So, instead of placing the code in the script, let's put it into a function. Enter the following lines in the editor and save the file as **rationalMenu**.

```
function n=rationalMenu
clc
fprintf('Rational Arithmetic Menu:\n\n')
fprintf('1). Reduce a fraction to lowest terms.\n')
fprintf('2). Add two fractions.\n')
fprintf('3). Subtract two fractions.\n')
fprintf('4). Multiply two fractions.\n')
fprintf('5). Divide two fractions.\n')
fprintf('6). Exit program.\n\n')
n=input('Enter number of your choice: ');
```

Test the function **rationalMenu** to make sure that it functions properly. Enter the following command at the Matlab prompt in the command window.

```
n=rationalMenu
```

The response feels the same. A menu is presented and the user responds by entering a number to indicate their menu choice.

```
Rational Arithmetic Menu:

1). Reduce a fraction to lowest terms.
2). Add two fractions.
3). Subtract two fractions.
4). Multiply two fractions.
5). Divide two fractions.
6). Exit program.

Enter number of your choice: 3
n =

     3
```

Note that Matlab stores the user response in the variable n in the command window workspace (base workspace).

At this point, users can replace all current lines in the Matlab script file **rationalArithmetic** with a single line, namely **n=rationalMenu**. Running the script file **rationalArithmetic** will call and execute the function **rationalMenu** with the expected result.

However, we'd like to add a bit more functionality to the script. Specifically, we'd like to add a loop that continually replays the menu until the user makes the choice: 6) Exit program, whereupon, the script should exit. Open the script M-file **rationalArithmetic** in the editor and replace all existing lines with the lines that follow. Resave the script as **rationalArithmetic**.

```
finished=false;
while (~finished)
    n=rationalMenu;
    switch n
        case 1
        case 2
        case 3
        case 4
        case 5
        case 6
            finished=true;
    end
end
```

Run the script at the Matlab prompt in the command window.

```
>> rationalArithmetic
```

In response to the prompt, try entering choices 1 through 5. Note that in each case, the menu refreshes and you are prompted for another choice. Selecting choice #6 exits the script and returns you to the Matlab command prompt. When the user makes the choice to exit the program, the variable **finished** is assgined the value **true** and the **while** loop terminates.

Hopefully, readers can now ascertain where we're heading with this structure. We code specific individual tasks as functions and save them as function M-files in the same directory containing the script **rationalArithmetic**. We then can access these functions by calling them from the script. Once the program is fully implemented, we'll have a number of function M-files collected in a directory that are driven from a script residing in the same directory.

We will pursue this design and structure no further (functions and a master script file), as we will now consider how we can implement a similar design with subfunctions.

Subfunctions

To use subfunctions to accomplish our programming task, we make a simple but subtle change to our structure. We take the script file and change it into a function by adjusting the first line so that it contains the keyword **function**.

```
function rationalArithmetic
finished=false;
while (~finished)
    n=rationalMenu;
    switch n
        case 1
        case 2
        case 3
        case 4
        case 5
        case 6
            finished=true;
    end
end
```

Execute this function by typing the following at the Matlab command prompt.

```
rationalArithmetic
```

You should note no change in the behavior of the program.

So, where's the advantage? The advantage lies in the fact that instead of creating *external* function M-files that are called from with a script, we can actually include the code for the **rationalMenu** function as a *subfunction* inside the primary function. Open the editor, enter the following lines, and save the file as **rationalArithmetic.m**.

```
function rationalArithmetic
finished=false;
while (~finished)
    n=rationalMenu;
    switch n
        case 1
        case 2
        case 3
        case 4
        case 5
        case 6
            finished=true;
    end
end

function n=rationalMenu
clc
fprintf('Rational Arithmetic Menu:\n\n')
fprintf('1). Reduce a fraction to lowest terms.\n')
fprintf('2). Add two fractions.\n')
fprintf('3). Subtract two fractions.\n')
fprintf('4). Multiply two fractions.\n')
fprintf('5). Divide two fractions.\n')
fprintf('6). Exit program.\n\n')
n=input('Enter number of your choice: ');
```

Readers should note no difference in performance when they enter the command **ratinalArithmetic** at the Matlab prompt. The savings here is the fact that when

we replace the driving script file with a function M-file, this primary function can contain as many subfunctions as needed to complete the programming task. In this way, the program is *self contained.* One file contains everything needed by the program to perform the task for which it was designed.

Adding Functionality to our Program

In this section, we will write subfunctions that will reduce a fraction to lowest terms. This corresponds to the first choice on the menu of choices produced by the subfunction **rationalMenu**.

How to proceed? We reduce a fraction to lowest terms in two steps. First, we find a greatest common divisor for both numerator and denominator, then divide both numerator and denominator by this greatest common denominator.

For example, to reduce the rational number 336/60 to lowest terms, we proceed as follows.

1. Find the greatest common divisior of 336 and 60, which is $\gcd(336, 60) = 12$.
2. Divide numerator and denominator by the greatest common divisor.

$$\frac{336}{60} = \frac{336 \div 12}{60 \div 12} = \frac{28}{5}$$

Two tasks call for two subfunctions, one to find the gcd and a second to do the actual reduction.

The Euclidean Algorithm. Early in our mathematical education we learn that if you take an nonegative integer a and divide it by a positive integer b, there is an integer quotient q and remainder r such that

$$a = bq + r, \qquad 0 \le r < b.$$

We can use this *Euclidean Algorithm* to find the greatest common divisor of 336 and 60. We proceed as follows.

1. When 376 is divided by 60, the quotient is 5 and the remainder is 36. That is,

$$336 = 5 \cdot 60 + 36.$$

2. When 60 is divided by 36, the quotient is 1 and the remainder is 24. That is,

$$60 = 1 \cdot 36 + 24.$$

3. When 36 is divided by 24, the quotient is 1 and the remainder is 12. That is,

$$36 = 1 \cdot 24 + 12.$$

4. When 24 is divided by 12, the quotient is 2 and the remainder is 0. That is,

$$24 = 2 \cdot 12 + 0.$$

The last nonzero remainder is the greatest common divisor. That is,

$$\gcd(336, 60) = 12.$$

Let's write a Matlab function that will implement the Euclidean Algorithm. Open the Matlab editor and enter the following lines. Save the file as **rationalGCD.m**.

```
function d=rationalGCD(a,b)
while true
    a=mod(a,b);
    if a==0
        d=b;
        return
    end
    b=mod(b,a);
    if b==0
        d=a;
        return
    end
end
```

We must test this function. Enter the following at the Matlab prompt.

```
>> d=rationalGCD(336,60)
d =
    12
```

Note that this is the correct response, matching our hand calculations above.

Our code warrants some comments.

1. Recall that the command **a=mod(a,b)** stores in a the remainder when a is divided by b. If the remainder a is zero, as we saw above, then b is the GCD, which we assign to the output variable d, then issue a **return**. The **return** command exits the function and returns control to the caller, in this case the command window.
2. Because a now contains the remainder from the first division, **b=mod(b,a)** stores in b the remainder when b is divided by a. This is equivalent to the

second step of our hand calculation above. If the remainder b is zero, then a is the GCD, which we assign to the output variable d, then issue a **return**.

3. Iterate until done.

Now that we have a function that will find the greatest common divisor, let's craft a function that will reduce a fraction to lowest terms.

Reducing to Lowest Terms. Our function will take as input the numerator and denominator of the rational number. It will then call **rationalGCD** to find the greatest common divisor of the numerator and denominator. It will then divide both numerator and denominator by the greatest common divisor and output the results. Open the Matlab editor, enter the following lines, then save the file as **rationalReduce.m**.

```
function [num,den]=rationalReduce(num,den)
d=rationalGCD(num,den);
num=num/d;
den=den/d;
```

Let's test this function on the fraction 336/60, which we know reduces to 28/5.

```
>> [num,den]=rationalReduce(336,60)
num =
     28
den =
      5
```

This is the correct response. The calling script of function will need to take responsibility for formatting the output.

Adding Subfunctions to Primary Function. We've tested **rationalGCD** and **rationalReduce** and found them to produce the correct results. We will now add each of these functions as subfunctions to our existing primary function **rationalArithmetic**. Open the file **rationalArithmetic** and add the following lines at the very bottom of the file **rationalArithmetic.m**.

```
function [num,den]=rationalReduce(num,den)
d=rationalGCD(num,den);
num=num/d;
den=den/d;
```

Next, add the following lines at the very bottom of the file **rationalArithmetic.m**.

```
function d=rationalGCD(a,b)
while true
    a=mod(a,b);
    if a==0
        d=b;
        return
    end
    b=mod(b,a);
    if b==0
        d=a;
        return
    end
end
```

Actually, the order of the subfunctions is irrelevant, but collecting them at the bottom of the primary function M-file **rationalArithmetic.m** is a good practice, making them easy to find and separating them from the main body of code. However, the order in which they appear at the bottom of the primary function file **rationalArithmetic.m** makes no difference to execution.

Activating Menu Items. We now need to write some code so that when the user selects the first menu item (Reduce a fraction to lowest terms), two things will happen:

1. The user will be asked to input the numerator and denominator of the fraction she wants reduced.
2. The program will output an equivalent fraction that is reduced to lowest terms.

With these thoughts in mind, add these lines to **rationalArithmetic.m** just after **case 1**.

```
    clc
    fprintf('Reduce a rational number to lowest terms.\n\n')
    num=input('Enter the numerator: ');
    den=input('Enter the denominator: ');
    fprintf('You''ve entered the fraction %d/%d.\n\n',num,den)
    [num,den]=rationalReduce(num,den);
    fprintf('When reduced to lowest terms: %d/%d\n\n',num,den)
    fprintf('Press any key to continue.\n')
    pause
```

Resave the file as **rationalArithmetic.m**. Test the result by entering the command **rationalArithmetic** at the Matlab prompt. The following menu appears.

```
Rational Arithmetic Menu:

1). Reduce a fraction to lowest terms.
2). Add two fractions.
3). Subtract two fractions.
4). Multiply two fractions.
5). Divide two fractions.
6). Exit program.


Enter number of your choice:
```

Enter 1 as your choice and hit the `Enter` key. Enter the following responses to the prompts.

```
You have chosen to reduce a rational number to lowest terms.

Enter the numerator: 336
Enter the denominator: 60
You've entered the fraction 336/60.

When reduced to lowest terms: 28/5

Press any key to continue.
```

Pressing any key returns the user to the main menu where she can make another choice. Selecting 6). Exit the program will exit the program.

A few comments are in order.

1. The **fprintf** commands are self explanatory.
2. The heart of this code is the call to the **rationalReduce** subfunction, which takes the user's input for the numerator and denominator, then returns the reduced form of the numerator and denominator. The ensuing **fprintf** commands format the result.
3. The **pause** command is new, but simply explained. It halts program execution until the user strikes a key.

It should now be a simple matter to complete menu items #4 and #5. You need only multiply two rational numbers as follows,

$$\frac{a}{b} \cdot \frac{c}{d} = \frac{a \cdot c}{b \cdot d},$$

then reduce the result. For the division, you need only invert and multiply,

$$\frac{a}{b} \div \frac{c}{d} = \frac{a}{b} \cdot \frac{d}{c} = \frac{a \cdot d}{b \cdot c},$$

then reduce the result.

To accomplish these tasks in code requires two subfunctions, **rationalMultiply** and **rationalDivide**. You will also have to write code that requests input, performs the requested operation, then formats the output after **case 4** and **case 5**.

Addition and subtraction are a bit trickier.

Finding a Least Common Denominator. To add and subtract fractions, the program will need to find a least common denominator (LCD). For example, to add 5/12 and 3/8, we make equivalent fractions with a commond denominator, then add.

$$\frac{5}{12} + \frac{3}{8} = \frac{10}{24} + \frac{9}{24} = \frac{19}{24}$$

Additionally, you must insure that the answer is reduced to lowest terms.

So, how do we write a subfunction that will produce a least common denominator? The following fact from number theory reveals the result.

$$\mathrm{lcd}(a, b) = \frac{ab}{\gcd(a, b)}$$

Thus,

$$\mathrm{lcd}(8, 12) = \frac{8 \cdot 12}{\gcd(8, 12)} = \frac{96}{4} = 24.$$

Thus, the code for the least common denominator is clear. Open the editor and enter the following lines. Save the file as **rationalLCD.m**.

```
function m=rationalLCD(a,b)
d=rationalGCD(a,b);
m=(a*b)/d;
```

Test the result at the command line.

```
>> m=rationalLCD(8,12)
m =
    24
```

This is the correct response. Now copy the following lines and enter them at the bottom of the primary function **rationalArithmetic.m** as follows.

```
function m=rationalLCD(a,b)
d=ratGCD(a,b);
m=(a*b)/d;
```

Completing rationalArithmetic.m

We'll leave it to our readers to complete the remaining tasks in the primary function **rationalArithmetic**. Here is what remains to be done.

1. You'll need to write subfunctions **rationalAdd** and **rationalSubtract**. Each should take two fractions as input, find an LCD, make equivalent fractions, add (or subtract), reduce, then send the result back to the caller.
2. Write code that requests input, performs the expected operation, then formats the output after **case 2** and **case 3**.

   Try to proceed as we have in the foregoing narrative. First write the standalone function **rationalAdd**. You might consider using the following first line.

```
function [num,den]=rationalAdd(num1,den1,num2,den2)
```

Note that the input calls for two fractions: **num1** and **den1** are the numerator and denominator of the first fraction input by the user, **num2** and **den2** are

the numerator and denominator of the second fraction input by the user. You will want to make calls to **rationalLCD** and **rationalReduce** in the standalone function **rationalAdd**. Once you have the function **rationalAdd** tested and working, paste it as a subfunction at the bottom of the file **rationalArithmetic**.

This is an imporant idea. Write the code for the smaller task. Test it thoroughly. When you gain confidence in the result, then drop it in the larger program and coordinate it with the existing code.

4.5 Exercises

---

1.   Complete the program **rationalArithmetic**. Implement four subfunctions, **rationalAdd**, **rationalSubtract**, **rationalMultiply** and **rationalDivide**, then add code to the corresponding **case** structure that will prompt the user for input, call the appropriate subfunction, then format the output. Test your function thoroughly.

2.   Although Matlab handles the arithmetic of complex numbers quite nicely, in this exercise you will write your own function to perform complex arithmetic. First, a bit of review. Recall that $i = \sqrt{-1}$ and $i^2 = -1$. With these facts, coupled with the knowledge that the usual laws of arithmetic apply equally well to the complex numbers (commutative, associative, distributive, etc.), it is not difficulty to show each of the following results.

i.) $(a + bi) + (c + di) = (a + c) + (b + d)i$

ii.) $(a + bi) - (c + di) = (a + c) - (b + d)i$

iii.) $(a + bi)(c + di) = (ac - bd) - (ad + bc)i$

iv.) $\dfrac{a + bi}{c + di} = \dfrac{(ac + bd) + (bc - ad)i}{c^2 + d^2}$

It is instructive to provide a proof of each of these results. Here is your task. Name your primary function **complexArithmetic**, then perform each of the following tasks.

a.)   Write a subfunction **complexMenu** that will offer the user a choice of four operations, addition, subtraction, multiplication, division, and exiting the program. This routine should ask the user for a numerical response and return the choice to the calling function in the output variable n.

b.)   The primary function should have a **case** structure that responds to the user's choice in n. Each item in the case structure should prompt the user to enter two complex numbers as row vectors, i.e., as **[a,b]** and **[c,d]**, representing $a + bi$ and $c + di$, respectively. This input should be sent as row vectors to the appropriate subfunction, then the returned row vector should be formatted and printed to the screen with **fprintf** in a manner you deem fitting.

c.)   Write four subfunctions, **complexAdd**, **complexSubtract**, **complexMultiply**, and **complexDivide**. Each subfunction should accept two row vectors as

---

[2] Copyrighted material. See: http://msenux.redwoods.edu/IntAlgText/

input and return a row vector as output. For example, the first line of the subfunction **complexDivide** should read as follows.

```
function w=complexDivide(u,v)
```

Note that the function receives the row vectors $[a, b]$ and $[c, d]$ in the arguments **u** and **v**, respectively. The output of **complexDivide** should be returned as a **row vector** in the output argument **w**. Before including your subfunction in the primary function, test it thoroughly. One suggestion would be to turn on rational display with **format rat**, then enter **(2+3i)/(2+i)** at the Matlab prompt and note the result. Now, feed $[2, 3]$ and $[2, 1]$ to your standalone function **complexDivide** and compare the result. Once the subfunction is written and tested, add it as a subfunction to the primary function **complexArithmetic**.

Thoroughly test the primary function **complexArithmetic** for accuracy, insuring that each of the five menu choices function properly and provide accurate output.

3. On November 13, 1843, Sir William Rowan Hamilton presented his first paper on the quaternions to the Royal Irish Academy. It was entitled *On a new Species of Imaginary Quantities connected with a theory of Quaternions* and was published in Volume 2 of the *Proceedings of the Royal Irish Academy*. It is interesting to read about the discovery in Sir William's own words in a letter to his son.

Every morning in the early part of the above-cited month, on my coming down to breakfast, your (then) little brother William Edwin, and yourself, used to ask me, "Well, Papa, can you *multiply* triplets"? Whereto I was always obliged to reply, with a sad shake of the head: "No, I can only *add* and *subtract* them."

But on the 16th day of the same month — which happened to be a Monday, and a Council day of the Royal Irish Academy — I was walking in to attend and preside, and your mother was walking with me, along the Royal Canal, to which she had perhaps driven; and although she talked with me now and then, yet an *under-current* of thought was going on in my mind, which gave at last a *result*, whereof it is not too much to say that I felt at once the importance. An *electric* circuit seemed to *close*; and a spark flashed forth, the herald (as I *foresaw*, *immediately*) of many long years to come of definitely directed thought and work, by *myself* if spared, and at all events on the part of *others*, if I should even be allowed to live long enough distinctly to communicate the discovery. Nor could I resist the impulse — unphilosophical as it may have been — to cut with a knife on a stone

of Brougham Bridge, as we passed it, the fundamental formula with the symbols, i, j, k; namely,

$$i^2 = j^2 = k^2 = ijk = -1, \tag{4.1}$$

which contains the *Solution of the Problem*, but of course, as an inscription, has long since mouldered away. A more durable notice remains, however, on the Council Books of the Academy for that day (October 16th, 1843), which records the fact, that I then asked for and obtained leave to read a Paper on Quaternions, at the *First General Meeting* of the session: which reading took place accordingly, on Monday the 13th of the November following.

This letter and other correspondence by Sir William, along with a collection of his work and papers, can be found at the following URL.

http://www.maths.tcd.ie/pub/HistMath/People/Hamilton/

The quaternions grew out of Hamilton's desire to extend the geometry associated with complex numbers. A complex number has the form $a + bi$, where a and b are arbitrary real numbers. Hamilton first attempted to extend this to another dimension, working with numbers having the triplet form $a + bi + cj$, but soon discoved that another dimension was needed. Thus, the general form of a quaternion is $a + bi + cj + dk$. The discovery of the relationship (**4.1**) initiated Hamilton's journey into the world of quaternions.

To this day, the quaternions remain influential. Students in abstract algebra meet the definition when they begin their work on the theory of groups. The geometry of quaternions enables rotations and reflections in four dimensional space. The role of the quaternions in quantum mechanics and physics is nicely documented at the following URL.

http://world.std.com/~sweetser/quaternions/qindex/qindex.html

Sir William had no difficulty determining how to add two quaternions. In short,

$$(a+bi+cj+dk)+(e+fi+gj+hk) = (a+e)+(b+f)i+(c+g)j+(d+h)k. \tag{4.2}$$

The definition of subtraction established by Sir William is not unexpected.

$$(a+bi+cj+dk)-(e+fi+gj+hk) = (a-e)+(b-f)i+(c-g)j+(d-h)k \tag{4.3}$$

Scalar multiplication was easily established by Sir William.

$$\alpha(a + bi + cj + dk) = (\alpha a) + (\alpha b)i + (\alpha c)j + (\alpha d)k \tag{4.4}$$

We now reach the point where Sir William was confounded, multiplication of two quaternions. His startling breakthrough, captured during the walk with his wife along the Royal Canal, allowed him to make the further developments.

$$ij = k = -ji, \qquad jk = i = -kj, \qquad ki = j = -ik \tag{4.5}$$

Students of vector calculus will recognize the familiar relationships amongst i, j, and k, if they think of i, j, and k as unit vectors along the x, y, and z axes in the usual 3-space orientation used in vector calculus.

Of course, the relationships in (**4.5**) demonstrate that multiplication of the quaternions is *not commutative*. Changing the order of the factors changes the product. In general, if u and v are quaternions, it is not the case that uv equals vu.

Sir William was able to demonstrate that multiplication was an associative operation ($(uv)w = u(vw)$) and that multiplication is distributive with respect to addition ($u(v + w) = uv + uw$). With associativity and the distributive law in hand, a straightforward (albeit messy) calculation shows that the product of the quaternions $u = u_1 + u_2 i + u_3 j + u_4 k$ and $v = v_1 + v_2 i + v_3 j + v_4 k$ is

$$uv = (u_1 v_1 - u_2 v_2 - u_3 v_3 - u_4 v_4) + (u_1 v_2 + u_2 v_1 + u_3 v_4 - u_4 v_3)i$$
$$+ (u_1 v_3 + u_3 v_1 + u_4 v_2 - u_2 v_4)j + (u_1 v_4 + u_4 v_1 + u_2 v_3 - u_3 v_2)k. \tag{4.6}$$

When you studied the complex numbers, before making the definition of division, you paused to develop the *complex conjugate*. In a similar manner, if $u = u_1 + u_2 i + u_3 j + u_4 k$, then the conjugate of this quaternion is the quaternion

$$\overline{u} = \overline{u_1 + u_2 i + u_3 j + u_4 k} = u_1 - u_2 i - u_3 j - u_4 k. \tag{4.7}$$

Note that the conjugate of a quaternion if very similar to the conjugate of a complex number. That is, if z is the complex number $z = a + bi$, then the conjugate of z is $\overline{z} = a - bi$. Geometrically, the conjugate of the complex number z is the reflection of z across the real axis, as shown in **Figure 4.1**.
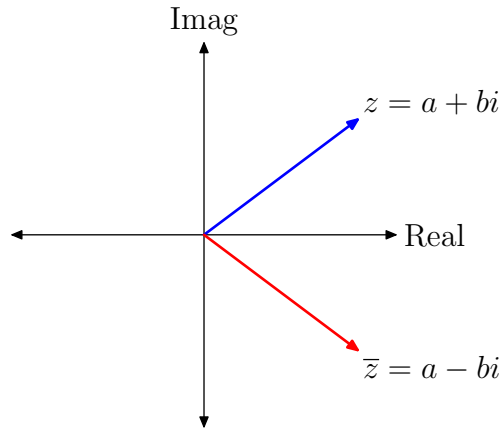

Figure 4.1.   The conjugate is a reflection across the real axis.

Moreover, recall that multiplying a complex number by its conjugate produces the square of length of the complex number. That is,

$$z\bar{z} = (a + bi)(a - bi) = a^2 - b^2 i^2 = a^2 + b^2.$$

Note that in **Figure 4.2**, the Pythagorean Theorem gives the length of the complex number $z = a + bi$ as $\sqrt{a^2 + b^2}$. Hence, $z\bar{z} = a^2 + b^2$ give the square of the length.
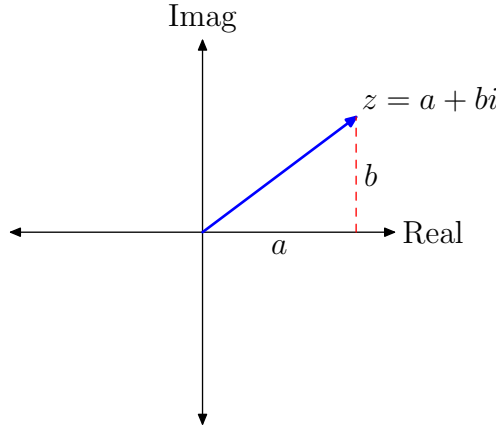

Figure 4.2.   The magnitude of the complex number is $\sqrt{a^2 + b^2}$.

In similar fashion, if v is the quaternion $v = v_1 + v_2 i + v_3 j + v_3 k$, then one can use **equation (4.6)** to show that

$$v\bar{v} = (v_1^2 + v_2^2 + v_3^2 + v_4^2) + 0i + 0j + 0k = v_1^2 + v_2^2 + v_3^2 + v_4^2.$$

Thus, in a sense, $v\bar{v}$ gives the square of the length of the quaternion. That is, $v\bar{v}$ is a scalar, not a quaternion.

We are now in a position where we can divide one quaternion by another. If the quaternion is nonzero, then it has nonzero length and an easy calculation reveals that

$$v \cdot \frac{\bar{v}}{v\bar{v}} = 1.$$

Thus,

$$v^{-1} = \frac{\bar{v}}{v\bar{v}}. \tag{4.8}$$

When writing your program, take note that $v\bar{v}$ is a real number. So, if you use **quaternionMultiply** to compute $v\bar{v}$, you will need to divide $\bar{v}$ by the *first component* of the result returned by **quaternionMultiply**.

Division is now easily defined.

$$\frac{u}{v} = uv^{-1}. \tag{4.9}$$

We begin our programming task. Save a primary function as **quaternionArithmetic**, then perform each of the following tasks.

a.) The primary function should store the quaternions

$$u = 1 + 2i + 3j + 4k \qquad \text{and} \qquad v = 5 + 6i + 7j + 8k$$

as row vectors **u=[1,2,3,4]** and **v=[5,6,7,8]**. It should then use **fprintf** to format and output four results to the screen, $u + v$, $u - v$, uv, and u/v.

b.) Write and implement each of the following subfunctions:

   i.)   **function w=quaternionAdd(u,v)**

   ii.) **function w=quaternionSubtract(u,v)**

   iii.) **function v=quaternionConjugate(u)**

   iv.) **fucntion v=quaternionInverse(u)**

   v.)   **function w=quaternionDivide(u,v)**

   Note that the input and output to each of these routines are quaternions, which should be stored a row vectors of length four. Before adding each subfunction to your primary function, test each function thoroughly for accuracy.

Some hints are in order:

- **quaternionAdd** and **quaternionSubtract** should use **equations (4.2)** and **4.3**, respectively.
- **quaternionMultiply** should use **equation (4.6)**.
- **quaternionConjugate** should use **equation (4.7)**.
- **quaternionInverse** should use **equation (4.8)**, which in turn requires a call of the subfunction **quaternionConjugate** to complete the calculation.
- **quaternionDivide** should use **equation (4.9)**, which in turn will require calls to **quaternionInverse** and **quaternionMultiply** to complete the calculation.

Because there is no user input involved, you should be able to open the editor in cell mode and simply call **quaternionArithmetic** and publish the result to HTML. Include your function **quaternionArithmetic** as a comment. Your grade on this assignment will reflect the accuracy of your answers, so you might want to compare and contrast your answers with your classmates. One way to do this is through the Discussion Board on Blackboard.