The Quaternions

David Arnold

May 9, 2002

1 Introduction

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,
>
> $$i^2 = j^2 = k^2 = ijk = -1, \tag{1}$$
>
> 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 (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`

In this activity, we will learn how to declare a new type, called a *structure*, and we will use this to define a quaternion type. Let's begin.

2 Declaring and Using Structures

F provides a facility that allows the suer to define new data types. These new datatypes are called *derived types*, which are used to define a structure. Suppose for example, that you wish to collect information about customers. You might wish to record the following data about each customer in your business's database.

1. Name
2. Phone Number

Thus, you need to craft a structure for each customer with two components,[1] name and phone number. In addition, phone numbers usually have two components: the area code and the phone number, as in (707) 476-4222. So, a phone type might be defined as follows.

```
type, public :: phone_type
  integer :: area_code, number
end type phone_type
```

In F, this declaration must be made in a module. We can now define a derived type for our customer as follows.

```
type, public :: customer
  character(len=40) :: name
  type(phone_type) :: phone
end type customer
```

Note that we are nesting structures inside of structures, a technique that is both allowed and encouraged in F. We can now declare a customer with the following command.

```
type(customer) :: david
```

The variable `david` has type `customer`, with two components, `name` and `phone`. Two questions require immediate attention.

[1] We are keeping the example simple. Other components besides name and phone number might include address, last item purchased, etc.

1. How do we assign values to our customer variable `david`, and once assigned,

2. how do we access the data in `david`?

In each case, the % symbol allows us to access any component of a derived type. For example, suppose that we want to assign "David Arnold" to the `name` component of the customer variable `david`. This is easily done with this command.

```
david % name = ‘‘David Arnold’’
```

The spaces before and after the % symbol are not required, but they do ease readability and we encourage their use.

Now, David's phone number is (707) 476-4222, so these commands will populate the `phone` component of the customer variable `david`.

```
david % phone % area_code = 707
david % phone % number = 4764222
```

Here is a small program verifying these observations.

```
module database_needs

  type, public :: phone_type
     integer :: area_code, number
  end type phone_type

  type, public :: customer
     character(len=40) :: name
     type(phone_type) :: phone
  end type customer

end module database_needs

program database
  use database_needs
  type(customer) :: david
  david % name = "David Arnold"
  david % phone % area_code = 707
  david % phone % number = 4764222
  print *, "The customer's name is: ", david % name
  print *, "The customer's area code is : ", david % phone % area_code
  print *, "The customer's phone number is: ", david % phone % number
end program database
```

Here is the resulting output.

```
 The customer's name is: David Arnold
 The customer's area code is :  707
 The customer's phone number is:  4764222
```

2.1 Populating Structures With Data

As with arrays, F also provides constructors for structures. Thus, we can populate the customer variable `david` as follows.

```
  david=customer(‘‘David Arnold’’, phone_type(707,4764222))
```

Clearly, this is more concise, making the code more compact.

```
program database
  use database_needs
  type(customer) :: david
  david=customer("David Arnold", phone_type(707,4764222))
  print *, "The customer's name is: ", david % name
  print *, "The customer's area code is : ", david % phone % area_code
  print *, "The customer's phone number is: ", david % phone % number
end program database
```

Note that no changes to the module are necessary. The output of this program is identical to the first.

3 Defining a Quaternion Structure

To begin our activity on the quaternions, let's begin by defining a quaternion type, after which we will want to write a subroutine that will print the quaternion to the screen in a nicely formatted manner. Recall that a quaternion has the form

$$a + bi + cj + dk, \tag{2}$$

where a, b, c, and d are arbitrary real numbers. Certainly, the following type definition seems natural.

```
type, public :: quaternion
 real :: a
 real :: b
 real :: c
 real :: d
end type quaternion
```

Of course, this can be written more concisely.

```
type, public :: quaternion
 real :: a, b, c, d
end type quaternion
```

Obviously, it is advantageous to code the quaternion type in the second form, as this definition takes up much less room in our source code. However, there is something to be said about the first definition. It also has its merits, as it is makes clear the fact that the quaternion has four separate components. We will leave it to our readers to decide which definition is best.

We now write a routine to print the quaternion, adopting a strategy of printing the quaternion as an ordered 4-tuple. We place this routine in the module and declare it public, which allows its use in any program unit that "uses" the module.

```
module Quaternions_needs

public :: quaternion_print

type, public :: quaternion
real :: a
real :: b
real :: c
real :: d
end type quaternion

contains

  subroutine quaternion_print(q)
    type(quaternion), intent(in) :: q
    real :: ap, bp, cp, dp
    ap = q % a
    bp = q % b
    cp = q % c
    dp = q % d
    print "(a1,4f12.6,a1)", "(",ap,bp,cp,dp,")"
  end subroutine quaternion_print

end module Quaternions_needs

program Quaternions
  use Quaternions_needs
  type(quaternion) :: v
  v=quaternion(1,2,3,4)
  call quaternion_print(v)
end program Quaternions
```

We use a structure constructor in the main program to initialize the quaternion. The output of the program follows.

```
(    1.000000    2.000000    3.000000    4.000000)
```

Now that we have the quaternion type defined, it's time to do a little quaternion arithmetic.

4 Overloading Operators

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

$$(a + bi + cj + dk) + (e + fi + gj + hk) = (a + e) + (b + f)i + (c + g)j + (d + h)k. \tag{3}$$

It is a simple matter to write a function that adds two quaternions.

```
  function quat_add(x,y) result (res)
    type(quaternion), intent(in) :: x, y
```

```
    type(quaternion) :: res
    res % a = x % a + y % a
    res % b = x % b + y % b
    res % c = x % c + y % c
    res % d = x % d + y % d
  end function quat_add
```

With this routine, we can define quaternions `u`, `v`, and `w`, assign data to `u` and `v`, then store the sum of `u` and `v` is `w` with this command.

```
w=quat_add(u,v)
```

However, readers will agree that it would be much nicer if we could do this with a simpler command.

```
w = u + v
```

This would be much more natural, but the difficulty lies in the fact that this operator (+) does not handle operands that are quaternions. Fortunately, F allows us to *overload* the operator (+), giving it multiple meanings, depending upon the type of operands used with the operator.

We first declare the operator (+) to be public. If we don't plan to call the `quat_add` routine directly from the main program, then there is no need to declare that it is public.

```
public :: operator(+)
private :: quat_add
```

Here's where we use the interface concept once again.

```
interface operator(+)
   module procedure quat_add
end interface
```

With this interface, when we use the operator (+) with two quaternions, the private module function `quat_add` is used to do the arithmetic.

Finally, we initialize two quaternions, then add and print them in our main procedure. To make sure that we haven't completely destroyed the ability of the operator (+) to do what it normally does, we also print the sum of two integers as a test. Here is the code "in toto."

```
module Quaternions_needs

  public :: quaternion_print
  public :: operator(+)
  private :: quat_add

  type, public :: quaternion
     real :: a
     real :: b
     real :: c
     real :: d
  end type quaternion

  interface operator(+)
```

```
     module procedure quat_add
  end interface

contains

  subroutine quaternion_print(q)
    type(quaternion), intent(in) :: q
    real :: ap, bp, cp, dp
    ap = q % a
    bp = q % b
    cp = q % c
    dp = q % d
    print "(a1,4f12.6,a1)", "(",ap,bp,cp,dp,")"
    print *
  end subroutine quaternion_print

  function quat_add(x,y) result (res)
    type(quaternion), intent(in) :: x, y
    type(quaternion) :: res
    res % a = x % a + y % a
    res % b = x % b + y % b
    res % c = x % c + y % c
    res % d = x % d + y % d
  end function quat_add

end module Quaternions_needs

program Quaternions
  use Quaternions_needs
  type(quaternion) :: u, v, w
  u=quaternion(1,2,3,4)
  v=quaternion(5,6,7,8)
  w=u+v
  call quaternion_print(w)
  print *, 2+3
end program Quaternions
```

The output shows that our code is performing as expected.

```
(    6.000000    8.000000   10.000000   12.000000)

 5
```

5 The Program

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

$$(a + bi + cj + dk) - (e + fi + gj + hk) = (a - e) + (b - f)i + (c - g)j + (d - h)k \tag{4}$$

Add a function `quat_subt` to the module and use it to overload the operator $(-)$.

Scalar multiplication was easily established by Sir William.

$$\alpha(a + bi + cj + dk) = (\alpha a) + (\alpha b)i + (\alpha c)j + (\alpha d)k \tag{5}$$

Add a function `quat_scalar_mult` to the module and use it to overload the operator $(\times)$.

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.

$$ij = k = -ji, \qquad jk = i = -kj, \qquad ki = j = -ik \tag{6}$$

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 (6) 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 $u = u_1 + u_2i + u_3j + u_4k$ and $v = v_1 + v_2i + v_3j + v_4k$ is

$$\begin{aligned} uv = {} & (u_1v_1 - u_2v_2 - u_3v_3 - u_4v_4) + (u_1v_2 + u_2v_1 + u_3v_4 - u_4v_3)j \\ & + (u_1v_3 + u_3v_1 + u_4v_2 - u_2v_4)j + (u_1v_4 + u_4v_1 + u_2v_3 - u_3v_2)k. \end{aligned}$$

Add a function `quat_mult` to the module and append it to the previously overloaded operator $(\times)$.

When you studied the complex numbers, before making the definition of division, you paused to develop the *complex conjugate*. In a similar manner, if $u = u_1 + u_2i + u_3j + u_4k$, then the conjugate of this quaternion is the quaternion

$$\overline{u} = \overline{u_1 + u_2i + u_3j + u_4k} = u_1 - u_2i - u_3j - u_4k. \tag{7}$$

There's a little twist that we must add if we overload an existing *intrinsic function*. Our intent here is to overload the intrinsic function `conjg`, which is used to take the complex conjugate of a complex number. The first rule is that we must use an intrinsic statement in the module containing the definition of the extension of an existing intrinsic function.

```
intrinsic :: conjg
```

Next, we intend to use the `conjg` function call in our main programming unit, so we must declare it `public` in the module.

```
public :: conjg
```

The interface strategy is the same, but this time we are not dealing with an operator, so the look and feel is a bit different.

```
interface conjg
   module procedure quat_conjg
end interface
```

The code for the function `quat_conjg` is quite easy to write.

```
function quat_conjg(x) result (res)
  type(quaternion), intent(in) :: x
  type(quaternion) :: res
  res % a = x % a
  res % b = -(x % b)
  res % c = -(x % c)
  res % d = -(x % d)
end function quat_conjg
```

Add these code fragments to your module.

Next, it's again a straightforward calculation to show that if $u = u_1 + u_2 i + u_3 j + u_4 k$, then the norm or magnitude of the quaternion is

$$\text{norm}(u) = u\overline{u} = u_1^2 + u_2^2 + u_3^2 + u_4^2. \tag{8}$$

Note the similarity between this and the magnitude of the complex number $z = a + bi$.

$$\text{norm}(z) = z\overline{z} = a^2 + b^2. \tag{9}$$

If you are thinking geometrically, then the norm of a complex number is the length of the vector representing the complex number. In Figure 1, the complex number $z = a + bi$ is represented as a vector in the Argand plane (complex plane). By the Pythagorean Theorem, the square of the length of the complex number is $z\overline{z} = a^2 + b^2$. Thus, the length of the complex number, usually denoted by $|z|$, is $|z| = \sqrt{a^2 + b^2}$.

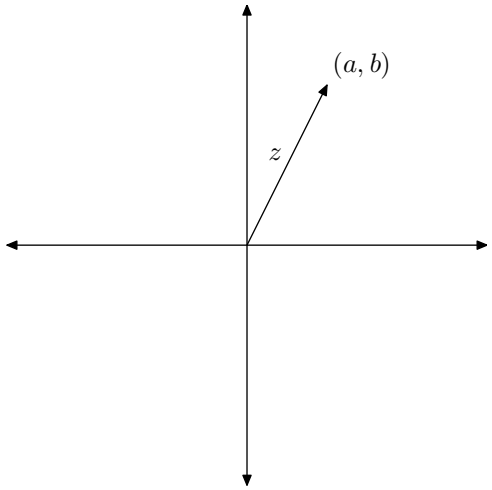

Figure 1: The complex number $z = a + bi$ represented as a vector in the complex plane.

In Fortran, the intrinsic function `abs` is used to find the magnitude of a number. If the number z is complex, then `abs(z)` returns the magnitude of the complex number, or the length of the vector representing the complex number in the complex plane. Your assignment is to overload the intrinsic function `abs` so that it returns the magnitude of the quaternion passed to it. Add the appropriate code to your module.

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

$$v \cdot \frac{\overline{v}}{v\overline{v}} = 1. \tag{10}$$

Thus,

$$v^{-1} = \frac{\overline{v}}{v\overline{v}}. \tag{11}$$

Division is now easily defined.

$$\frac{u}{v} = uv^{-1}. \tag{12}$$

Add a function `quat_div` to your module that will take the quotient of two quaternions. Overload the operator (/).

To test the routines you've written for quaternion arithmetic, let

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

In your main program unit, print each of the following to the screen.

1. $u + v$
2. $u - v$
3. $2u$
4. uv
5. conjg(u)
6. abs(u)
7. u/v

Your grade on the program will depend upon the correctness of these answers, so please check your solutions with others before submitting your program.

6 The Grading Rubric

The following rules will apply for this program, after which we will discuss and adjust the rubric during class.

1. (30 points) Will be awarded for adequate comments. Comments should include:
   (a) A description of the program's purpose.
   (b) Name, date, version or revision number.
   (c) A complete dictionary of all variables and parameters used in the program.
   (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.

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?

7 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.

8 Managing Files and Folders

Each of you has been given personal space on the sci-math server to store your work. Typically, 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 this folder, create a new folder call FortranPrograms. Note that you must **never** use spaces in filenames. In the Windows operating system, filenames are not case-sensitive, which is exactly opposite what happens in Unix and Linux, where filenames are case-sensitive.

In your FortranPrograms folder, create another folder called Program13. It is in this folder that you are to place the source code and executables for this current project. Please name your program `program13.f90`. When you receive your next project, create a new folder called Program14 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. Use the Windows Explorer to copy the files on your disk into the proper folder (H:/FortranPrograms/Program13). Do not copy executables to the school drive. Rather, copy your source, then compile on the school computer to produce the executable.

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

9 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.