

Computational Astrophysics

Fortran 90 notes

1 Programs, compilers & terminology

A fortran 90 program is one (or more) files containing *source code*, which is a sequence of commands such as

```
program test
  print *, 'Hello World!'
end program test
```

These files can be given any name ending in `.f90`, and can be written by any text editor you want, as long as the output is plain text (and not for example a Word file, or an rtf file). For example, the above program could be saved in a file called `mycode.f90`. The name after the `program` commands in the source code does not have to match the filename.

A *compiler* reads in the source code and writes it out into a machine-executable file. The compiler available on the Linux machines is `g95`, and the compiler available on the Suns is `f90`. In this section the examples are given using `g95`, but if you are using a Sun simply replace this by `f90`

The following command *compiles* the source code, i.e. it creates a file which can be *executed* to run your program:

```
g95 mycode.f90
# replace mycode.f90 by whatever your source code file is called
```

`a.out` is the default name given by the compiler to the executable file, but is not a very interesting name. Instead you can specify the `-o` option when you compile the code, in the following way, in order to give it a more explicit name:

```
g95 mycode.f90 -o myprog
# replace myprog by whatever you want to call the executable file
```

The next step is to run the program. To do this, type the name of the executable file preceded by `./`

For the above example where the file is called `myprog`, use

```
./myprog
```

If you decided not to give the executable file a name other than `a.out`, then use

```
./a.out
```

mycode.f90 and myprog are just examples of names you can use. In practice you could use for example question1.f90 for the source code file, and q1 for the executable file. In this case you would write your source code in question1.f90, and then run the following commands to compile and run the program:

```
g95 question1.f90 -o q1      # To compile
./q1                        # To run
```

Once you write more complex programs, you may have the main program in a file and a subroutine in another. To compile this type of program, simply list all the .f90 files needed in the compiling command:

```
g95 mycode.f90 subroutine.f90 -o myprog
```

2 Integer, real and double precision variables

As seen in the lectures, to declare an integer variable named `count` for example, you should write the following at the top of the program

```
integer :: i
```

Similarly to declare a real variable named `distance` for example you should write

```
real :: distance
```

And finally, if you wish to declare a double precision variable named `alpha`, you should write

```
real*8 :: alpha
```

or even better,

```
double precision :: alpha
```

In all programs, it is very important to be *consistent* with variable types. An `integer` number always has to consist of a number with no decimal places, such as 1 or 5, and a `real` number always has to contain a period, *even if it has no decimal places other than 0*. For example the correct way to write number five as a real number is 5. or 5.e0 (not 5 which is an integer). Finally, `double precision` numbers have to contain a period and have to be followed by `d0`. Here are examples of integers, reals, and double precision variables:

```
Integer : 0  -2  3  19  1234  -332
Real    : 0.  0.e0  3.9  1.223  9.  8.2e0  3.4e8  2.e8
Double  : 0.d0  2.d8  4.3321222d9
```

Therefore the following statements are wrong, assuming the variable types declared above:

```
i=2.0          # Should be i=2
distance=0     # Should be distance=0. or 0.e0
alpha=2        # Should be alpha=2.d0
alpha=2.021    # Should be alpha=2.021d0
```

In a more general sense, expressions should not mix different types. For example the following expressions are wrong:

```
i/2.2          # Should be i/2
distance**2/98 # Should be distance**2./98.
alpha*distance # This is not correct, you can't mix types
alpha**2.      # Should be alpha**2.d0
```

If you do want to calculate `alpha*distance`, you have to use either

```
distance*real(alpha)
alpha*dble(distance)
```

In the top case, `real()` converts `alpha` to a real variable. This expression is overall real, meaning that if for example I want to multiply it by two, I need to use `distance*real(alpha)*2`.

In the bottom case, `dble()` converts `distance` to a double precision number. The expression is overall double precision, meaning that if I want to multiply it by two, I need to use `alpha*dble(distance)*2.d0`

Similarly if you do want to calculate `i/2.2` rather than `i/2`, then you can convert `i` to a real number by using `real(i)/2.2`. If you want `i/2.2` to be a double precision number, then use `dble(i)/2.2d0`.

Thomas Robitaille
12 February 2006