Fall 2021: Computational Science I

Instructor: Mohammad Sarraf Joshaghani
(m.sarraf.j@rice.edu)

Module 5: FORTRAN programming

brief intro to fortran 90

why? we want to also discuss using **blas** and **lapack** in c. these are important linear algebra packages that are written in fortran.

some important distinctions between fortran and c

- function arguments are automatically passed by reference.
- indexing starts from 1, not 0, as in c.
- matrices are stored in column major format, while in c, matrices are stored in row major format.

the important parts of a fortran program are:

- **functions**: they have a return value
- **subroutines**: they do not have a return value.
- **modules**: they are collections of subroutines and functions.

fortran program and module structure

here is the basic structure of a **program**:

```
PROGRAM my_program
! include modules
USE module1
! declarations
! instructions
END PROGRAM my_program
```

and the basic structure for a **module**:

```
MODULE module1
! modules
! declarations
! instructions
END MODULE module1
```

syntax for subroutines

```
SUBROUTINE name_sub(arg1, arg2)
! modules
! declarations
! instructions
END SUBROUTINE name_sub
```

some more general comments on syntax

- capital and lowercase letters are considered the same.
- lines are ended with a "new line" and not a semicolon.

a subroutine is called as follows:

```
CALL name_sub(arg1, arg2)
```

syntax for functions

```
FUNCTION name_func(arg1, arg2 ) RESULT (foo)
! modules
! declarations
! instructions (need to return foo)
RETURN
END FUNCTION name_func
```

the function is called as:

```
var = name_func(arg1, arg2)
```

where `var` and `foo` have the same type.

variable declaration

available types: INTEGER, REAL, COMPLEX, LOGICAL, CHARACTER

```
! this is necessary for historical reasons
IMPLICIT NONE
LOGICAL :: bool
INTEGER :: n
! equivalent to a float
REAL (KIND = 4) :: f
! equivalent to a double
REAL (KIND = 8) :: g
! an array of dimension 4 by 3 by 2
INTEGER , DIMENSION (4 ,3 ,2) :: array_int_3D
```

the IMPLICIT NONE line is important. this means that we do NOT want fortran to implicitly decide our variable types. for example, variables with names beginning with "i, j, k, l, m, n" would automatically be of type INTEGER (this is not good!!)

specifying inputs and outputs

the attributes `IN`, `OUT`, and `INOUT` specify if a parameters is an input, and output, or both, respectively.

```
SUBROUTINE my_sub (v1, v2, v3)
IMPLICIT NONE

INTEGER, INTENT(IN) :: v1
INTEGER, INTENT(INOUT) :: v2
INTEGER, INTENT(OUT) :: v3

! instructions
END SUBROUTINE my_sub
```

a parameter that is declared with `IN` **cannot** be modified by the subroutine or function.

loops and conditional statements

an **if-else statement**:

```
IF ( condition1 ) THEN
! Instructions
ELSE IF ( condition2 ) THEN
! instructions
ELSE
! instructions
END IF
```

a **for loop**:

```
DO i = 1 , n
! instructions
END DO
```

a **while loop**:

```
DO WHILE ( condition )
! instructions
END DO
```

reading and writing in fortran

this is done with the `READ` and `WRITE` functions.

typical usage might be:

```
WRITE(*,*) "hello world!"
```

which would output the string `"hello world"` to the screen. the first input in the `WRITE` function corresponds to *where* you want to write to, and the second input corresponds to the *format specifier*.

an asterisk in the first input means that you are going to write to the terminal.

an asterisk in the second input means "list-directed io," where the program simply tries to figure out the best way to display whatever follows the `WRITE` function.

the second input can be an format specifier that should follow certain rules, see: https://pages.mtu.edu/~shene/COURSES/cs201/NOTES/chap05/format.html

compiling fortran programs

this can be done in the same way as we compiled c programs, but we want to use a fortran compiler. for example, if the compiler is called `gfortran`, we can create an executable called `main` as:

```
gfortran -o main my_program.f90
```

we can also do this in two steps, where we go from source file to object file and then object file to executable:

```
gfortran -c my_program.f90
gfortran my_program.o -o main
```

using fortran functions and subroutine within c code

to do this, there are some things to keep in mind.

- the name of the fortran function/subroutine must be supplemented with an underscore at the end of it when being called in c code, i.e. `my_fortran_function` in fortran would need to be called `my_fortran_function_` in the c source file.
- arguments (i.e. function parameters) *must* be passed by reference.
- function outputs are passed by *copy*.

compiling c and fortran programs together

you can do this in two steps, where we generate object files from the fortran and c source files:

```
gcc -c main.c
gfortran -c fortran_functions.f90
```

these two commands will generate object files `main.o` and `fortran_functions.o`, which can be combined into a single executable as:

```
gcc -o main main.o fortran_functions.o -lgfortran
```

where we need to include the `-lgfortran` flag.

blas and lapack

blas: " basic linear algebra package."

see http://www.netlib.org/blas/

lapack: "linear algebra package."

see http://www.netlib.org/lapack/.

these libraries are important implementations of cutting edge linear algebra operations. they have been around for a while (and are written in fortran), but they are continually updated.

more about blas

blas contains implementations for "low level" linear algebra operations that are needed for more sophisticated software, like lapack. these operations fall into one of three categories:

blas **level 1**: scalar, vector, and vector-vector operations
(think scalar multiplication and dot products).

blas **level 2**: matrix-vector operations.

blas **level 3**: matrix-matrix operations

more about lapack

lapack contains implementations of things you would want to do with matrices, like compute eigenvalues and eigenvectors

$$\boldsymbol{A}\boldsymbol{x} = \lambda \boldsymbol{x}$$

and solve linear systems

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}.$$

there are many ways to solve these types of problems. you can take into account the structure of the problem, like if $\boldsymbol{A} = \boldsymbol{A}^T$, in which case, there might be more efficient algorithms than the "default" choice.

example: LU factorization

an LU factorization of a matrix $\boldsymbol{A}$ is a factorization

$$\boldsymbol{A} = \boldsymbol{L}\boldsymbol{U}$$

where $\boldsymbol{L}$ is unit lower triangular and $\boldsymbol{U}$ is upper triangular.

this factorization can be directly built from the process of Gaussian elimination, which converts a matrix to upper triangular form.

note that if you want to solve a linear system $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$, you can equivalently solve

$$\boldsymbol{L}\boldsymbol{y} = \boldsymbol{b}$$
$$\boldsymbol{U}\boldsymbol{x} = \boldsymbol{y}$$

why is it "better" to solve these two linear systems?

installing blas and lapack

you can install these fortran libraries on your ubuntu machine like:

```
sudo apt-get install libblas-dev liblapack-dev
```

to link these libraries with a c program that might be using functions from them, you would compile as follows:

```
gcc -c main.c
gcc main.o -llapack -lblas -lgfortran
```