Lab #3

Computational Physics I (**Phys381**)
R. Ouyed
Due Feb. 13, 2014 (at the end of class)

[**Total: 100 marks including 20 marks for the report**]

- The latexed report is worth **20%** of the total mark. Only well documented and neatly presented reports will be worth that much! It is not sufficient to give only numerical results and show plots, you should also discuss your results. Include complete figure captions, introduction and conclusion sections.
- Your report must be in a *two-column format*.
- Your report should include your *Fortran code* and *Gnuplot scripts* in an Appendix using the *verbatim* command.
- If applicable, animations should be shown to the teacher and/or TA before handing in the lab report. The animation should be well documented and contains necessary information (student name, assignment number, run time etc ...). Basically, the information should be included in each frame before they are put together.
- You must name your report using the names (last names only): *student1-student2-phys381-lab#.pdf*.
- **Procedure for Handing in your lab report (see instructions on phys381 wbsite)**:

  1) Set permission to your PDF report as 644. It means:
  chmod 644 *student1-student2-phys381-lab#.pdf*
  2) cp -a *student1-student2-phys381-lab#.pdf* **/home/ambrish/phys381/labs/lab#**
  3) Copy a second time to ensure that your exam copied correctly. If you are prompted as to whether or not you would like to replace the existing file, then you report has been successfully submitted.

- **You must check with your TAs (Ambrish or Zach) that your report was received and is readable BEFORE you leave the lab.**

1 Part A: Handling arrays and matrices

[Total: 10 Marks]

In Fortran 90, arrays can be "allocated" the memory they require at runtime and when they are no longer required they can be deallocated and the memory freed. This "Dynamic Allocation" procedure requires three separate stages, **declaration**, **allocation** and **deallocation**.

- [**4 Marks**] Explain what is performed by the lines indexed by the symbol **"!! *** !!"** in the code below.

```
program phys381-arrays
implicit none
integer ::  N, M, L
integer :: i, j, k
real, allocatable, dimension(:,:,:) :: x,y,z     !! *** !!

open(12, file="input.dat")
read(12,*)  N
read(12,*)  M
read(12,*)  L
allocate(x(N,M,L))      !! *** !!
allocate(y(N,M,L))
allocate(z(N,M,L))

do k = 1, L
   do j=1, M
      do i=1, M
           x(i,j,k)=float(i)*float(j)*float(k)     !! *** !!
           y(i,j,k)=float(i)*float(j)*float(k)
           z(i,j,k)=float(i)*float(j)*float(k)
           write(6,'(3f10.6,1x)') x(i,j,k), y(i,j,k), z(i,j,k)     !! *** !!
    enddo
  enddo
enddo
deallocate(x)  !! *** !!
deallocate(y)
deallocate(z)
close(12)
end program phys381-arrays
```

- **[2 Marks]** How would you modify the input file *input.dat* to read in the values of *N, M and L* from the same line instead of separate lines as given in the version above.
- **[2 Marks]** How would you modify the FORMAT in *write(6,'(3f10.6,1x)')* to display N as a an f8.5, M as an f10.6 and L as an f9.5. At the same time, N separated from M by 2 white spaces and M from L by 4 white spaces.

1.1 Passing arrays through the argument list

Allocatable arrays behave exactly the same in your code as explicitly declared arrays and can be passed through argument lists[1] to subroutines and functions. *However, besides passing the names of the arrays the subroutine also needs the size of the arrays.*

[2 Marks] Explain what is performed by the lines indexed by the symbol **"!! *** !!"** in the code below.

```
Program phys381-pass-array
implictit none
integer, parameter ::  N=3,M=3     ‘‘!! *** !!"
real, dimension(N,M) :: x, y, z
real, dimension(N) ::  W

x(1:,1:) = 2.0    ‘‘!! *** !!"
y(1:,1:) = 3.0
z(1:,1:) = 4.0
W(1:) = 5.0

call PassInfo(N,M,x,y,z,W)

Contains
subroutine PassInfo(ismax,jsmax,xi,yi,zi,Wi)
implicit none
integer, intent(in) :: ismax,jsmax       ‘‘!! *** !!"
real, dimension(1:ismax,1:jsmax) :: xi, yi, zi
real, dimension(1:ismax):: Wi

write(6,*) xi(1:2,1:1)
write(6,*) yi(1:2,1:3)    ‘‘!! *** !!"
.
.  skipped lines
```

[1] If a procedure has a dummy argument that is an allocatable then the best practise is to put all of the subroutines in a module and to use this module in the main program. With the module, the interface is automatically known.

```
.
Do i=1, ismax
write(14,*)  xi(i,:)        ‘‘!! *** !!"
End Do
Close(14)
end subroutine PassInfo
end program phys381-pass-array
```

2 Part B: Modules, Functions and Subroutines

[Total: 70 Marks]

2.1 General Questions

[20 Marks]

- **[5 Marks]** What is the difference between a *function* and a *subroutine*? What is the difference between an internal and external subroutine? When would you use external subroutines?
- **[5 Marks]** Read the following code and describe the use of **common blocks**. Also, do you think that the order and type of variables in a **common block** must be the same wherever that block is used?

```
program ouyedcode
implicit none
integer :: a,b
common/block_1/a,b
integer c,n
print*,'Give integers a and b'; read*,a,b
print*,'How many strange operations should be performed?'; read*,n
call strangeoperation(n,c)
print*,'The strange operations have resulted in the value ',c
end program ouyedcode

subroutine strangeoperation(n,strange)
implicit none
integer :: a,b
common/block_1/a,b
```

```
integer i,n,strange
strange=0
do i=1,n
  strange=strange+(strange-a)**2+a*b*strange
  if (abs(strange) > 100000) strange=strange/1000+10*i
enddo
end subroutine strangeoperation
```

- **[5 Marks]** What is a module and how would you call a module from your main code?
- **[5 Marks]** Are the following statements about modules **True** or **False**?

  → By saying implicit none at the beginning of the module, you don't have to say it at the beginning of the subroutines within the module.

  → Modules, unlike program blocks, cannot have code that is not contained within a procedure.

  → Modules can hold data, declared, as in a common block (see example above), before the contains statement.

2.2 Subroutines and Functions

[20 Marks]

- **[10 Marks]** Write a main program and internal subroutine that returns, as its first argument, the sum of two real numbers. Do not forget the use of INTENT. *test it.*
- **[10 Marks]** Ad to the main program above an internal function that returns the sum of two real numbers supplied as arguments. *test it.*

2.3 Modules

[30 Marks]

- **[10 Marks]** Copy the code you developed above to a new one. Give it a different name so that you have a backup code – describe what **commands** you used to perform this task.
- **[10 Marks]** Now rewrite your code by encapsulating your subroutine and function into ONE module. Compile, run and check that you get similar results as in the previous code.

- **[5 Marks]** Now let's play an interesting game: In the previous version of the code, try to call the subroutine that sums 2 numbers with one more argument than you have assigned to it. For example if you have **call sum2(output,x,y)** in your main program replace it with **Call sum2(output,x,y,z)** where z is the extra variable (you must define it of course). **Keep your subroutines defined without the extra variable !**. Compile and run it. What do you notice?

- **[5 Marks]** Do the same thing but now in your modular version of the code. Compile and run it. What do you conclude?

A Compile Programs with Modules

Since we will not be covering **Makefiles** in this course, here are some instructions on how to compile codes that refer to external subroutines and to modules.

Normally, your programs and modules are stored in different files, all with filename suffix **.f90**. When you compile your program, **all involved modules must also be compiled**. For example, if your program is stored in a file **main.f90** and you expect to use a few modules stored in files **compute.f90**, **convert.f90** and **constants.f90**, then the following command will compile your program and all three modules, and make an executable file **a.out**

gfortran compute.f90 convert.f90 constants.f90 main.f90

If you do NOT want the executable to be called **a.out**, you can do the following:

gfortran compute.f90 convert.f90 constants.f90 main.f90 -o main.exe

In this case, **-o main.exe** instructs the Fortran compiler to generate an executable **main.exe** instead of **a.out**.

Different compilers may have different "personalities." This means the way of compiling modules and your programs may be different from compilers to compilers. If you have several modules, say A, B, C, D and E, and C uses A, D uses B, and E uses A, C and D, then the safest way to compile your program is the following command:

gfortran A.f90 B.f90 C.f90 D.f90 E.f90 main.f90

That is, list those modules that do not use any other modules first, followed by those modules that only use those listed modules, followed by your main program.