

Fortran Modernisation Workshop

Exercises

Numerical Algorithms Group

This exercise will involve modernising a legacy Fortran 77 code¹ to modern Fortran. The code solves the one dimensional heat diffusion equation:

$$\frac{\partial H}{\partial t} - \kappa \frac{\partial^2 H}{\partial x^2} = f(x) \quad (1)$$

where κ is the heat coefficient. Equation (1) describes the distribution of heat between $x_{\min} = 0$ and $x_{\max} = 1$ and uses the following explicit finite difference scheme to integrate in time:

$$H_i^{(n+1)} = H_i^{(n)} + \text{CFL} \{ H_{i-1}^{(n)} - 2H_i^{(n)} + H_{i+1}^{(n)} \} + \Delta t f(x_i) \quad (2)$$

$$\text{where CFL} = \kappa \frac{\Delta t}{\Delta x^2} \quad (3)$$

Equation (3) is known as the Courant-Friedrichs-Lewy coefficient which must satisfy the condition $\text{CFL} > 0.5$ for (2) to be stable. Don't worry if you do not know what all this means - the focus of the exercise is on Fortran programming.

The exercises are split into three parts: section one, section two and optional parallel exercises. Firstly, set up Git:

```
git config --global user.name "firstname lastname"
git config --global user.email firstname.lastname@address.com
cd fmw_exercises
git init
git add .
git commit -m "initial version"
```

Replace the *firstname* and *lastname* with your name :-). The Git commands will version control your code so you can see its revision history. Please put all your source code in the `src/` directory.

¹https://people.sc.fsu.edu/~jburkardt/f77_src/fd1d_heat_explicit/fd1d_heat_explicit.html

Optional: if you have one, you can add a remote repository (GitHub² or Bitbucket³) to your local repository using the following commands and replacing *username* and *repo* with your upstream username and repository name, respectively:

```
git remote add origin git@bitbucket.org:username/repo.git
git push origin master # after making changes to local repo
```

which you should do at the end of the workshop as the Linux machines will be cleaned 5 days after the workshop. However, you do not need a remote repository for the workshop exercises and then copying them at the end of the workshop. Day one exercises will include modernising an existing Fortran 77 code `fd1d_heat_explicit.f90` which will be referred to the main program code.

²<https://github.com>

³www.bitbucket.org

Section One

1. Create a module `types_mod` and put it in the file `src/types_mod.f90` which contains the following numeric kind types:

```
1 use, intrinsic :: iso_fortran_env
2 integer, parameter :: SP = REAL32
3 integer, parameter :: DP = REAL64
4 integer, parameter :: SI = INT32
5 integer, parameter :: DI = INT64
```

Listing 1: Kind parameters

using the following module template:

```
1 module types_mod
2   use, intrinsic :: iso_fortran_env
3
4   implicit none
5   ! everything is private unless otherwise stated
6   private
7   public :: SP
8
9   integer, parameter :: SP = REAL32
10  ! put other kind parameters here from Listing 1
11 contains
12
13 end module types_mod
```

Listing 2: Module template

2. Compile the module using the command in the `src/` directory:

```
nagfor -c types_mod.f90
```

3. Use the NAG compiler unifying precision feature to change double precision to `real(kind=DP)` using the command below. The output is a temporary file shown in italics which is then moved to the original file - remember to close the main source file `fd1d_heat_explicit.f90` in any editor:

```
nagfor =unifyprecision -pp_module=types_mod -pp_name=DP \  
  fd1d_heat_explicit.f90 -o fd1d_heat_explicit.f90_prs  
mv fd1d_heat_explicit.f90_prs fd1d_heat_explicit.f90
```

The above command will also add the DP kind after literal constants, e.g. `0.5` will become `0.5_DP`

4. In the main program code, use the NAG compiler polishing feature to:

- (a) change relational operators, e.g. `.LE.` to `<=` using the flag `-relational=F90+`
- (b) lowercase keywords, e.g. `PROGRAM` to `program` using the flag `-kwcase=L`
- (c) set the leftmost margin to zero using the flag `-margin=0`
- (d) indentation to two space, using the flag `-indent=2`
- (e) add double colons for variable declarations, e.g. `integer i` to `integer :: i` using the flag `-dcolon_in_decls=Insert`
- (f) change old-style string declarations to modern Fortran string declarations, e.g. `character*11 :: str` to `character(len=11) :: str` using the NAG compiler flag `-character_decl=Keywords`

by using the command below. The output is a temporary file shown in italics which is then moved to the original file - remember to close the file `fd1d_heat_explicit.f90` in any editor:

```
nagfor =polish -relational=F90+ -kwcase=L -margin=0 -indent=2 \
  -dcolon_in_decls=Insert -character_decl=Keywords \
  fd1d_heat_explicit.f90 -o fd1d_heat_explicit.f90_pol
mv fd1d_heat_explicit.f90_pol fd1d_heat_explicit.f90
```

5. In the main program code, change how parameters are declared, e.g.

```
integer t_num
parameter ( t_num = 201 ) ! change to
integer, parameter :: T_NUM = 201
```

And the same for the variable `X_NUM`

6. In the functions and subroutines, use the `intent` keyword for all dummy arguments and properly scope the dummy arguments, e.g. `intent(in)` (read-only), `intent(out)` (write-only) or `intent(inout)` (read and write)

7. In the main program code, use dynamic memory allocation for the arrays `h(1:X_NUM)`, `h_new(1:X_NUM)`, `hmat(1:X_NUM,1:T_NUM)`, `t(1:T_NUM)`, `x(1:X_NUM)`. Check the status of the dynamic memory allocation

8. At the end of the main program code, deallocate the arrays declared in step 7

9. In the functions and subroutines, remove the size of the array and use assumed shaped arrays as dummy arguments:

- (a) For the function `func(j, x_num, x)` remove the `x_num` argument and replace the argument declaration to `real(kind=DP), dimension(:) :: x`. Make sure the calling of `func()` reflect this change

- (b) For the subroutine `fd1d_heat_explicit()` remove the `x_num` argument and ensure all arrays are declared as assumed shaped arrays. Use the automatic arrays feature in Fortran to declare the array `f(:)` as `real(kind=DP) :: f(size(x))`
- (c) For the subroutine `r8mat_write()` declare the arguments `m` and `n` as local variables and assign them to `size(table(:, :), 1)` and `size(table(:, :), 2)`, respectively. Declare the argument `table(:, :)` as an assumed shaped array
- (d) For the subroutine `r8vec_linspace()` remove the argument `n` and declare the argument `a(:)` as an assumed shaped array
- (e) For the subroutine `r8vec_write()` remove the argument `n` and declared the argument `x(:)` as an assumed shaped array

Use the `size()` intrinsic function to get array dimensions.

10. Compile both the main program and the created Fortran module:

```
nagfor -c types_mod.f90
nagfor -c -I. fd1d_heat_explicit.f90
nagfor fd1d_heat_explicit.o types_mod.o -o fd1d_heat_explicit.exe
./fd1d_heat_explicit.exe
```

11. To test whether your code runs correctly execute:

```
diff h_test01.txt h_test01.txt_bak
```

If the command outputs difference, then the refactoring introduced a bug.

12. Type `git diff fd1d_heat_explicit.f90` to see the refactored code. Stage and commit the changes by typing:

```
git add fd1d_heat_explicit.f90
git add types_mod.f90
git commit -m "refactored Fortran 77 into modern Fortran"
```

The following exercises will modularise the code and use the module template in Listing 2 for creating additional modules.

13. Create a module `rhs_mod` and put it in the file `src/rhs_mod.f90` and **move** the Fortran function `func()` (from `fd1d_heat_explicit.f90`) into `rhs_mod` and declare it public. In the main program code, insert the line `use rhs_mod`

14. Create a module `cfl_mod` and put it in the file `src/cfl_mod.f90` and **move** the subroutine `fd1d_heat_explicit_cfl()` into `cfl_mod` and declare it public. In the main program code, insert the line `use cfl_mod`

15. Create a module `io_mod` and put it in the file `src/io_mod.f90` and **move** the subroutines `r8mat_write()`, `r8vec_linspace()` and `r8vec_write()` into `io_mod` and declare them public. In the main program code, insert the line `use io_mod`

16. Create a module `solver_mod` and put it in the file `src/solver_mod.f90` and **move** the subroutine `fd1d_heat_explicit()` into `solver_mod` and declare it public. In the main program code, insert the line `use solver_mod`

17. Compile the recently created modules in the order shown:

```
nagfor -c -I. rhs_mod.f90
nagfor -c -I. cfl_mod.f90
nagfor -c -I. io_mod.f90
nagfor -c -I. solver_mod.f90
nagfor -c -I. fd1d_heat_explicit.f90
nagfor fd1d_heat_explicit.o types_mod.o rhs_mod.o cfl_mod.o io_mod.o \
      solver_mod.o -o fd1d_heat_explicit.exe
./fd1d_heat_explicit.exe
```

18. To test whether your code runs correctly execute:

```
diff h_test01.txt h_test01.txt_bak
```

If the command outputs difference, then the refactoring introduced a bug.

19. Add the newly created module files into Git and stage the modified main program for a another Git commit:

```
git add rhs_mod.f90 cfl_mod.f90 io_mod.f90 solver_mod.f90
git add fd1d_heat_explicit.f90
git commit -m "modularised RHS, CFL, IO and Solver"
```

Section Two

1. Write a Makefile for the Fortran code produced on day one in the same directory as the source code (src/) using the dependency graph in Figure 1

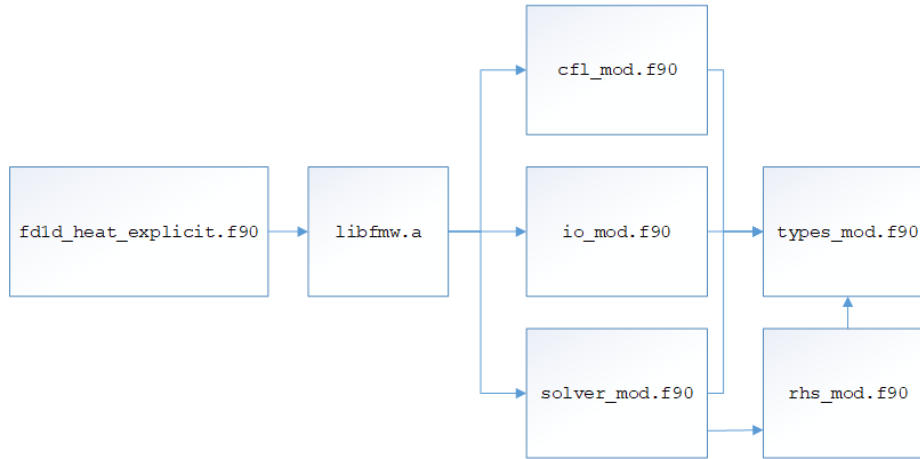


Figure 1: Dependency graph for Makefile

You can use the command:

```
nagfor =depend -otype=make *.f90 -ideclient | sed s#$(pwd)/##
```

to create the Makefile dependencies. The sed command will remove absolute paths and the -ideclient flag removes the header. Then create the rule for compilation:

```
%.o %.mod: %.f90
    nagfor -c -I. $<
```

Also add the link flags to create the executable. Remember to precede the command with the tab character.

2. Add a clean target which cleans the build:

```
.PHONY: clean

clean:
    rm -f *.mod *.o *.png *.exe *.a
```

3. Create a static library containing all the module object files:

```
ar rcs libfmw.a cfl_mod.o io_mod.o rhs_mod.o solver_mod.o types_mod.o
```

In the Makefile, use the static library `libfmw.a` in the link stage to create the final executable. The link line for linking `libfmw.a` is `-L. -lfmw`

- (a) After creating your Makefile, type `make -n` to see what commands will be executed without executing your commands, which is useful for debugging
- (b) Then type `make` to build your code
- (c) After creating the Makefile, add it to your repository using `git add Makefile`
- (d) Use the Linux command:

```
nm libfmw.a
```

which shows the symbols listed in the library just created. The symbol type field (second field) shows T for the symbol being **defined** in the library and U being **undefined** but being called by the library.

4. This task will cover Git in a bit more detail.

- (a) Type `git status` which will list the status of all the files. Notice that the object files (`*.o`), Fortran module files (`*.mod`) and executable files (`*.exe`) are listed as untracked files. These files need not be version controlled as they can be recreated
- (b) Create the file `.gitignore` in the root directory (`fmw_exercises/`). This configuration file will specify which files **not to version control**, e.g. object files, executable files or any file that can be recreated. Add the following extensions in the ignore file:

```
*.o
*.mod
*.exe
*.nc
*.dat
*.a
*.mp4
doc/
```

- (c) The `.gitignore` file also needs to be version controlled using `git add .gitignore`
- (d) Browse the commit history of all the Fortran files created using `git log`

5. The following exercises will involve using the NetCDF API by writing the `x(:)`, `t(:)` and `hmat(:, :)` variables in one file with meta-data. Please refer to the NetCDF documentation⁴ for the details of the API. Use the following process when creating NetCDF files for writing:

- `NF90_CREATE()` to create the file and enter define mode

⁴ <http://www.nag.co.uk/market/training/fortran-workshop/netcdf-f90.pdf>

- `NF90_DEF_DIM()` to create the x and t dimensions and name them x and t , respectively;
- `NF90_DEF_VAR()` to create $x(:)$ with variable name x -range, $t(:)$ with variable name t -range and $table(:, :)$ with variable name $solution$
- `NF90_PUT_ATT()` to put global and dimension attributes
- `NF90_ENDDEF()` to end define mode and to enter data mode
- `NF90_PUT_VAR()` to write the data to the file
- `NF90_CLOSE()` to close the file

(a) Open the file `io_mod.f90` and add the line `use netcdf`

(b) Open the main program code `fd1d_heat_explicit.f90` and pass the arguments $x(:)$ and $t(:)$ into the subroutine call `r8mat_write()` and change the file name from `h_test01.txt` to `h_test01.nc` - the file extension `.nc` is used to denote NetCDF files

(c) Comment out the two `r8vec_write()` subroutine calls as the $x(:)$ and $t(:)$ arrays will be written to a single NetCDF file using `r8mat_write()`

(d) Edit the subroutine `r8mat_write` and add the dummy arguments:

```
real(kind=DP), dimension(:), intent(in) :: x
real(kind=DP), dimension(:), intent(in) :: t
```

(e) When in define mode, add the following meta data using `NF90_GLOBAL` for `varid` argument:

- i. `"purpose" = "Fortran workshop"`
- ii. `"name" = "<Your name>"`
- iii. `"institution" = "<Your university>"`

(f) Add units meta-data for $x(:)$ which is metres, $t(:)$ which is seconds, and $table(:, :)$ which is Celsius

(g) In the subroutine `r8mat_write()` write the one-dimensional arrays $x(:)$ and $t(:)$ and the two-dimensional array $table(:, :)$ into a NetCDF file

(h) To include NetCDF Fortran bindings for compilation, add the flag in your Makefile:

```
-I/usr/local/netcdf/include
```

(i) To do the final link, add the link flag:

```
-L/usr/local/netcdf/lib -lnetcdff -lnetcdf
```

Note that the ordering of the flags is crucial (`netcdff` calls `netcdf` so this ordering is required). The library `netcdff` contains the Fortran bindings and `netcdf` is the actual implementation in the C language

(j) After executing your code, you can view the contents of the NetCDF file using:

```
ncdump h_test01.nc | less # or the command
ncks h_test01.nc | less # this gives much more information
```

(k) To verify whether your code works correctly, compare the created NetCDF file with the correct NetCDF file:

```
ncdiff --overwrite h_test01.nc h_test01.nc.valid -o diff.nc
ncwa -y max --overwrite diff.nc out.nc
ncdump out.nc | grep "solution ="
```

The last command should show 0 for the solution NetCDF variable

6. The following exercises will allow you to visualise the solution at every 10 time steps using the PLplot visualisation library. Please refer the PLplot documentation⁵ for further information. The visualisation will be done in the main program code (fd1d_heat_explicit.f90) using the following sequence of subroutine calls from within the main time loop:

- PLSFNAM() to set the output file name
 - PLSDEV() to set the output device to use. Set this to "pngcairo" which will save the images in the portable network graphics (PNG) format
 - PLINIT() to initialise PLplot
 - PLENV() to set the x - and y -range
 - PLLAB() to set the x and y labels, and the title of the graph
 - PLLINE() to set the x and y values which will be represented by the arrays $x(:)$ and $h_new(:)$, respectively
 - PLEND() to finalise PLplot
- (a) In the main program code, add the line `use plplot` so that PLplot features can be used
- (b) In the main time loop create an IF branch which is executed at every 10 time steps using the Fortran intrinsic function `mod()`
- (c) Create a string for the filename which includes the time step, e.g. `image001.png`
- (d) From the above list of PLplot subroutine calls, create the PNG file of the current time step
- (e) To compile the code to include the PLplot Fortran bindings, add the flag:

```
-I/usr/local/plplot/lib/fortran/modules/plplot
```

The include path contains the Fortran module files for PLplot

⁵<http://www.nag.co.uk/market/training/fortran-workshop/plplot-5.11.1.pdf>

(f) To link the code to the PLplot libraries, use the link flags

```
-L/usr/local/plplot/lib -lplplotfortran -lplplot
```

Note that the ordering of the link flags is crucial (plplotfortran calls plplot). The library plplotfortran contains the Fortran bindings and plplot is the actual implementation in the C language

(g) Then execute your code to create the PNG images of the solution at different time steps $n = 10, 20, \dots, 200$, e.g. `fd1d_heat_explicit_00010.png`

(h) Create a movie file with the list of images created using:

```
ffmpeg -f image2 -i fd1d_heat_explicit_%.png fd1d_heat_explicit.mp4
```

and view it using any video player by copying the movie `fd1d_heat_explicit.mp4` to your laptop/desktop

7. Ensure the Makefile is updated to include the compile and link flags for NetCDF and PLplot

Remember to commit everything and push your changes to your upstream repository:

```
git add -A # this adds all changes
```

```
git commit -m "completed workshop exercises"
```

```
git push origin master # push your changes upstream
```