Fortran Modernisation Workshop Exercises

Numerical Algorithms Group

May 31, 2018

This exercise will involve modernising a legacy Fortran 77 code[1] to modern Fortran. The code solves the one dimensional heat diffusion equation:

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

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

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

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

Equation (3) is known as the Courant-Friedrichs-Lewy coefficient which must satisfy the condition 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 two parts: day one and day two. 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. Git will be covered in the second day. Please put all your source code in the `src/` directory.

[1] https://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 (GitHub[2] or Bitbucket[3]) 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. Day two exercises will involve the following topics:

1. Makefiles for Fortran codes;
2. Git source code version control system;
3. Fortran Documenter tool;
4. NetCDF file format for arrays;
5. In-memory visualisation using PLplot;
6. Unit testing with pFUnit;
7. Verification using CamFort.

[2] https://github.com
[3] www.bitbucket.org

Day One Exercises

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 contains
11
12 end module Types_mod
```

Listing 2: Module template

2. 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`

3. 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
```

4. 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`

5. 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)

6. 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

7. At the end of the main program code, deallocate the arrays declared in step 6

8. 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.

9. 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
```

10. 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.

11. 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.

12. 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`

13. 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`

14. 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`

15. 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`

16. 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
```

17. 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.

18. 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"
```

Day Two Exercises

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

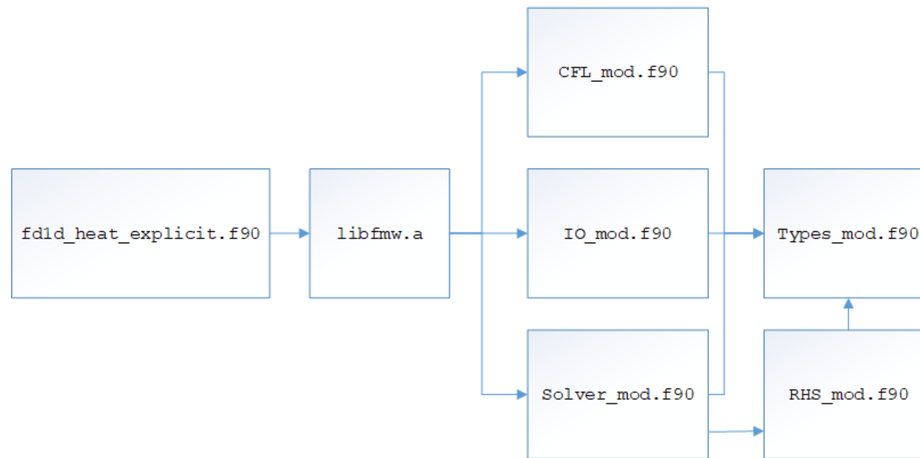

Figure 1: Dependency graph for Makefile

You can use the command:

```
makedepf90 *.f90
```

to create the Makefile dependencies and then create the commands

2. Add a `clean` target which cleans the build:

```
.PHONY: clean

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

Remember to precede the commands with the tab

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 set of questions will involve using Fortran Documenter to document your source code. Edit the template Fortran Documenter configuration file `fmw.md` in the root directory (`fmw_exercises/`) and set the following options:

Option	Value
project:	1D heat equation
src_dir:	./src
output_dir:	./doc
summary:	Fortran Workshop
author:	Your name
docmark:	!
predocmark:	>
display:	public protected private
source:	true
graph:	true
search:	true
version:	1.1.1
coloured_edges:	true

Note that the values for the `display` option must be on separate lines. After the last option in the above table, leave an empty space and create `@Note` and `@Bug` sections and add any text that describes your code.

6. On line 1 of the file `fd1d_heat_explicit.f90` add the following documentation which includes a LaTeX equation:

```
!> <Your name>, <Your affiliation>
!> Solves the one dimensional heat diffusion equation
!> \( \frac{\partial H}{\partial t}
!>     - \kappa\frac{\partial^{2} H}{\partial x^{2}} = f(x) \)
```

7. On line 1 of the module file `CFL_mod.f90` add the following comments:

```
!> this module calculates the CFL number
```

In the same module file and for the subroutine `fd1d_heat_explicit_cfl( )`, add the following comments before the appropriate code lines:

```
!> calculates the CFL number
!> \( \text{CFL} = \kappa\frac{\Delta t}{\Delta x^2} \)
subroutine fd1d_heat_explicit_cfl( )

!> calculated CFL number
real(KIND=DP), intent(inout) :: cfl
!> the heat constant \( \kappa \)
```

```
real(KIND=DP), intent(in) :: k
!> t_max upper bound of t-axis
real(KIND=DP), intent(in) :: t_max
!> lower bound of t-axis
real(KIND=DP), intent(in) :: t_min
!> number of intervals in t-axis
integer(KIND=SI), intent(in) :: t_num
!> upper bound of x-axis
real(KIND=DP), intent(in) :: x_max
!> x_min lower bound of x-axis
real(KIND=DP), intent(in) :: x_min
!> number of intervals in x-axis
integer(KIND=SI), intent(in) :: x_num
```

Run the FORD command `ford fmw.md` and then load the file `doc/index.html` in any Web browser to browse the source code documentation by copying the directory `doc/` to your laptop. To copy `doc/` in Linux, use the command from your laptop:

```
scp -r username@fortran-training.hpc.cam.ac.uk:~/fmw_exercises/doc .
```

where *username* is the username you have been provided.

(a) Click on `Program` tab (top right) which displays the module dependency graph as well as the call graph. The local variables and the source code is also displayed;

(b) The text also contains the LaTeX equation:

$$\frac{\partial H}{\partial t} - \kappa \frac{\partial^2 H}{\partial x^2} = f(x) \tag{4}$$

(c) Have a browse around the other links to familiarise yourself with Fortran Documenter;

(d) After the workshop has ended, do the same for the remaining module files (i.e. `IO_mod.f90`, `RHS_mod.f90`, `Solver_mod.f90`, `Types_mod.f90`)

(e) Commit the changed files in Git:

```
git add fd1d_heat_explicit.f90 CFL_mod.f90 fmw.md
git commit -m "added FORD documentation in source code"
```

8. 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 documentation[4] 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
- `NF90_DEF_VAR( )` to create the `x(:)` (variable name `x-range`), `t(:)` (variable name `t-range`) and `table(:, :)` (variable name `solution`) variables
- `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-4.6.1/include
```

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

```
-L/usr/local/netcdf-4.6.1/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

9. The following exercises will allow you to visualise the solution at every 10 time steps using the PLplot visualisation library. Please refer the PLplot documentation[5] 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-5.13.0/lib/fortran/modules/plplot
```

The include path contains the Fortran module files for PLplot

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

[5] `http://www.nag.co.uk/market/training/fortran-workshop/plplot-5.11.1.pdf`

```
-L/usr/local/plplot-5.13.0/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 $n = 10, 20, \ldots, 200$, 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

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

11. This exercise will involve creating a pFUnit test pseudocode. This exercise will only test the `fd1d_heat_explicit_cfl( )` subroutine.

(a) Create the following test driver code which is in pseudo Fortran and name the file `testCFL.pf` in the `src/` directory. This will test the `fd1d_heat_explicit_cfl( )` subroutine:

```
@test
subroutine testCFL( )
  use pFUnit_mod
  use CFL_mod
  use Types_mod

  integer, parameter :: t_num = 201
  integer, parameter :: x_num = 21
  real(KIND=DP) :: k, x_min, x_max, t_min, t_max
  real(KIND=DP) :: cfl, cfl_exact, tol

  tol = 0.0000001_DP
  cfl_exact = 0.32_DP
  k = 0.002_DP

  x_min = 0.0_DP
  x_max = 1.0_DP
  t_min = 0.0_DP
  t_max = 80.0_DP

  call fd1d_heat_explicit_cfl( k, t_num, t_min, t_max, &
```

```
                                    x_num, x_min, x_max, cfl )

  @assertEqual( cfl, cfl_exact, tol )
end subroutine testCFL
```

(b) Create the test configuration file `testSuites.inc` which will tell pFUnit which tests to execute. In this case, this will only execute the `testCFL` test above.

```
ADD_TEST_SUITE(testCFL_suite)
```

(c) Preprocess the pseudo Fortran test driver code to produce Fortran code (namely, `testCFL.F90`), type:

```
${PFUNIT}/bin/pFUnitParser.py testCFL.pf testCFL.F90 -I.
```

where `$PFUNIT` is the environment variable which points to the installation directory of pFUnit. Note that the Fortran code must have the `.F90` extension as it still needs to be preprocessed

(d) Then compile the created Fortran test driver code:

```
nagfor -I${PFUNIT}/mod -I. -c testCFL.F90
```

(e) Then create the final test driver executable (`tests.exe`):

```
nagfor -I${PFUNIT}/mod ${PFUNIT}/include/driver.F90 \
       CFL_mod.o testCFL.o -L${PFUNIT}/lib -lpfunit -I. -o tests.exe
```

Note that `CFL_mod.f90` must be compiled before the above command is executed

(f) This command will create the `tests.exe` binary executable which needs to be executed and will print the result of the test, which is either a pass or fail, and the time it took to complete the test

(g) Change the value `cfl_exact` to `0.34_DP` in the pseudo Fortran code and repeat steps (c), (d), (e) and (f) which should fail the test

(h) Add the pFUnit commands listed above in the Makefile and call the target `pfunit`

12. This exercise will involve using CamFort.

(a) Use the CamFort command to obtain the stencil specification:

```
camfort stencils-infer Solver_mod.f90
```

(b) From the stencil specification obtained in the previous step, annotate the code (in file `Solver_mod.f90`):

```
h_new(j) = h(j) + dt * f(j) + cfl * (h(j-1) - 2.0_DP * h(j) + h(j+1))
```

Remember to prefix every specification with `!= <specification>`

(c) Then use the CamFort command to verify the stencil specification:

```
camfort stencils-check Solver_mod.f90
```

which should print `Correct` for every specification.

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
```