Fortran Modernisation Workshop Exercises

F. Spiga, F. Chami and W. Miah

August 31, 2016

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

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

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

$$\mathbf{H}_i^{(n+1)} = \mathbf{H}_i^{(n)} + \mathrm{CFL}\{\mathbf{H}_{i-1}^{(n)} - 2\mathbf{H}_i^{(n)} + \mathbf{H}_{i+1}^{(n)}\} + \Delta t f(x_i) \tag{2}$$

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

Equation (3) is known as the Courant-Friedrichs-Lewy coefficient which must satisfy the condition CFL > 0.5 for the scheme (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 and not Maths.

The exercises are split into two parts: one set for the first day and a second set for the second day. Obtain the exercises by using the commands:

```
cp /usr/local/src/fmw_exercise.tar.gz $HOME
cd $HOME
tar -zxvf fmw_exercise.tar.gz
```

which will extract the exercises to the `fmw_exercise/` directory. Change into the `fmw_exercise/` directory:

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

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

The Git commands will version control your code so you can see its revision history. Git will be covered in the second session.

Day one exercises will include modernising an existing Fortran 77 code. Day two exercises will involve the following topics:

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

Day One Exercises

1. Create a module `Types_mod` and put it in the file `Types_mod.f90` which contains the following numeric data types:

```
use, intrinsic :: iso_fortran_env
integer, parameter :: SP = REAL32
integer, parameter :: DP = REAL64
integer, parameter :: SI = INT32
integer, parameter :: DI = INT64
```

using the following module template:

```
module Types_mod
   use, intrinsic :: iso_fortran_env

   implicit none

   public :: SP

   integer, parameter :: SP = REAL32
contains

end module Types_mod
```

The above

2. In the main program code (file `fd1d_heat_explicit.f90` including all functions and subroutines), include the double colon after the variable type and before the variable name, e.g. from `double precision a` to `double precision ::  a`

3. In the main program code, include the line `use Types_mod` just before the `implicit none` statement. This will allow you to use the constants declared in the `Types_mod` Fortran module

4. In the main program code, use the `kind` keyword in variable declarations, e.g. from `double precision` to `real(kind=DP)` and `integer` to `integer(kind=SI)`

5. In the main program code, change how parameters are declared, e.g. from `parameter ( T_NUM = 201 )` to `integer(kind=SI), parameter ::  T_NUM = 201` and the same for the variable `X_NUM`

6. In the main program code, change how constants are used, e.g. from `0.0D+00` to `0.0_DP`

7. In the functions and subroutines, use the `intent` keyword for dummy arguments

8. 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

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

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

- 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 invocations of `func( )` reflect this change
- For the function `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))`
- For the function `r8mat_write( )` remove the arguments `m` and `n` and assign them to `size( table(:,:), 1 )` and `size( table(:,:), 2 )`, respectively. Declare the argument `table(:,:)` as an assumed shaped array
- For the function `r8vec_linspace( )` remove the argument `n` and declare the argument `a(:)` as an assumed shaped array
- For the function `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.

11. In the main program code, use the modern string declaration statement. For dummy argument declaration:

```
character * ( * ) string    ! to
character(len=*) :: string
```

For string declarations:

```
character * ( 30 ) :: string             ! to
character(len=30)  :: string
```

12. In the main program code, use symbolic relational operators `<, <=, /=, ==, >=, >` instead of `.lt., .le., .ne., .eq., .ge., gt.`

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

```
gfortran -c Types_mod.f90
gfortran -c -I. fd1d_heat_explicit.f90
gfortran fd1d_heat_explicit.o Types_mod.o -o fd1d_heat_explicit.exe
./fd1d_heat_explicit.exe
```

14. 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.

15. 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 further modularise the code and use the module template in question 1 for creating additional modules.

16. Create a module `RHS_mod` and put it in the file `RHS_mod.f90` and put the Fortran function `func()` into `RHS_mod` and declare it public. In the main program code, insert the line `use RHS_mod`

17. Create a module `CFL_mod` and put it in the file `CFL_mod.f90` and put the Fortran function `fd1d_heat_explicit_cfl( )` into `CFL_mod` and declare it public. In the main program code, insert the line `use CFL_mod`

18. Create a module `IO_mod` and put it in the file `IO_mod.f90` and put the Fortran functions `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`

19. Create a module `Solver_mod` and put it in the file `Solver_mod.f90` and put the Fortran function `fd1d_heat_explicit( )` into `Sover_mod` and declare it public. In the main program code, insert the line `use Solver_mod`

20. Compile the recently created modules:

```
gfortran -c RHS_mod.f90
gfortran -c CFL_mod.f90
gfortran -c IO_mod.f90
gfortran -c Solver_mod.f90
gfortran -c -I. fd1d_heat_explicit.f90
gfortran 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
```

21. 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.

22. Add the newly created module files into Git and stage the changed 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 using the dependency graph in Figure 1

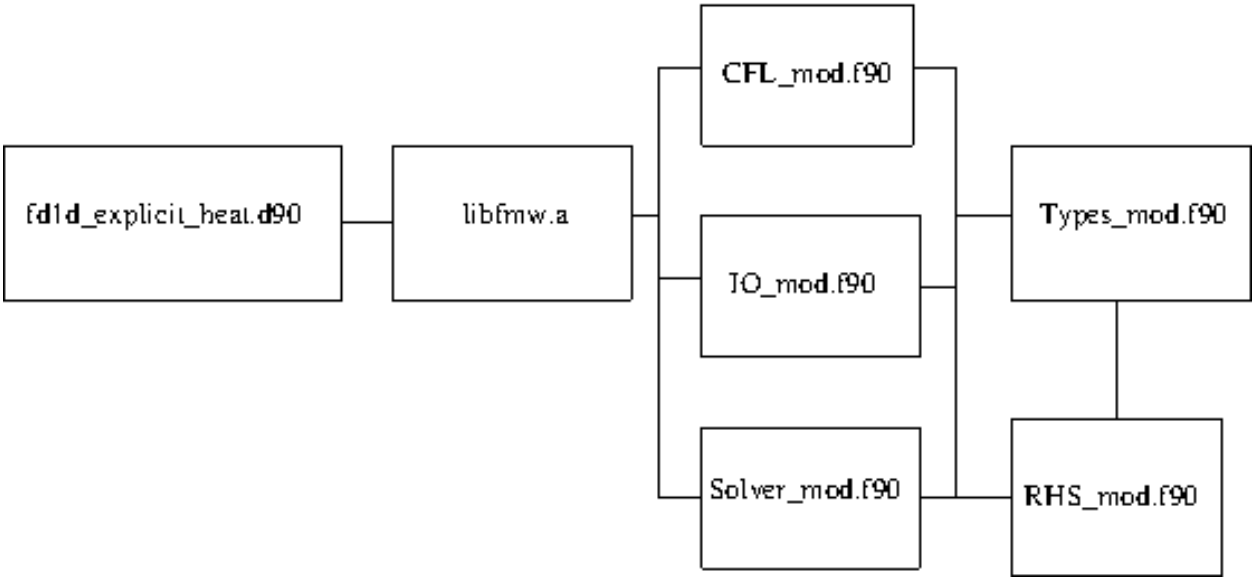

Figure 1: Dependency graph for Makefile

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

```
.PHONY: clean

clean:
        rm -f *.mod *.o *.png fd1d_heat_explicit.exe
```

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

(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 git 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 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 source code directory. This configuration file will specify which files to not 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
doxygen
```

(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. To create a Doxygen template configuration, type `doxygen -g fortran.dxg` in the same directory where the code resides. Then open the file `fortran.dxg` in any editor and set the following variables:

Description	Variable and value
Free text for project name	`PROJECT_NAME = "Fortran Workshop"`
Free text for project description	`PROJECT_BRIEF = "Fortran Workshop"`
Output directory for doxygen files	`OUTPUT_DIRECTORY = doxygen`
Configuring Doxygen for Fortran	`OPTIMIZE_FOR_FORTRAN = YES`
Input directory where code resides	`INPUT = .`
Fortran code file extension	`FILE_PATTERNS = *.f90`
Generate HTML reports	`GENERATE_HTML = YES`
Using Graphvis for generating call graphs	`HAVE_DOT = YES`
Generate call graph	`CALL_GRAPH = YES`
Generate caller graph	`CALLER_GRAPH = NO`
Extract all documentation	`EXTRACT_ALL = YES`
Extract private members of class	`EXTRACT_PRIVATE = YES`
Extract static members of file	`EXTRACT_STATIC = YES`
List source code of file	`SOURCE_BROWSER = YES`
Use free form Fortran	`EXTENSION_MAPPING = f90=FortranFree`

Run the Doxygen command `doxygen fortran.dxg` and then load the file `doxygen/html/index.html` in any Web browser to browse the source code documentation.

(a) Click on `Files` → `fd1d_heat_explicit.f90` which should display a call graph;

(b) Click on `Got to the source code of this file` to see the source code of the main program code;

(c) Click on `Files` → `Types_mod.f90` → `types_mod` to see the public constants in the module

(d) Have a browse around the other links to familiarise yourself with Doxygen;

(e) Then add the Doxygen configuration file to Git by typing `git add fortran.dxg`

(f) On line 1 of the file `fd1d_heat_explicit.f90` add the following Doxygen tags:

```
!> @author
!> <Your name>, <your affiliation>
!> @brief
!> Solves the one dimensional heat diffusion equation
!> \f$ \frac{\partial{\bf H}}{\partial t}
!>       - K\frac{\partial^{2}{\bf H}}{\partial x^{2}} = f(x) \f$
```

(g) On line 1 of the module file `CFL_mod.f90` add the following Doxygen tags:

```
!> @author
!> <Your name>, <your affiliation>
!> @brief
!> calculates the CFL number
```

In the same module file and before the subroutine `fd1d_heat_explicit_cfl( )` is defined add the following Doxygen content:

```
!> @author
!> <Your name>, <your affiliation>
!> @brief
!> calculates the CFL number
!> @param[in] k heat constant
!> @param[in] t_num number of intervals in t-axis
!> @param[in] t_min lower bound of t-axis
!> @param[in] t_max upper bound of t-axis
!> @param[in] x_num number of intervals in x-axis
!> @param[in] x_min lower bound of x-axis
!> @param[in] x_max upper bound of x-axis
!> @param[inout] cfl calculated CFL number
```

(h) Rerun the Doxygen command `doxygen fortran.dxg`

(i) Refresh your browser and click on `Files` → `fd1d_heat_explicit.f90` and you should now see the LaTeX heat diffusion equation with the description and author

(j) Click on `Files` → `CFL_mod.f90` → `More...` and you should now see the module author, subroutine author and description. In addition, you should see the subroutine dummy arguments with their description

(k) In your own time and after the workshop has ended do the same for the remaining module files (`IO_mod.f90`, `RHS_mod.f90`, `Solver_mod.f90`, `Types_mod.f90`)

(l) Type `git add fd1d_heat_explicit.f90 CFL_mod.f90` to stage the changes and then `git commit -m "added Doxygen tokens in source code"`

6. The following exercises will involve using the NetCDF API by writing the `x(:)`, `t(:)` and `hmat(:, :)` varaibles in one file including meta-data. Please refer to the NetCDF documentation at `http://www.nag.co.uk/market/training/fortran-workshop/netcdf-f90.pdf` 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
- `NF90_DEF_DIM( )` to create the x and t dimensions
- `NF90_DEF_VAR( )` to create the `x(:)`, `t(:)` and `table(:, :)` 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

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

(g) To compile the remember to add the line `-I/usr/lib64/gfortran/modules` in your Makefile which is the NetCDF Fortran module file directory on Fedora Linux. This might be different for your system

(h) To do the final link, remember to add the link line `-L/usr/lib64 -lnetcdff -lnetcdf` which is where the Fortran NetCDF wrapper resides on Fedora Linux. This might be different for your system

(i) Once your code completes, you can view the contents of the NetCDF file using `ncdump h_test01.nc | less`

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

```
ncdiff -O h_test01.nc h_test01.nc.valid diff.nc
ncdump diff.nc | less
```

The second command should show all zero values for the `x-range`, `t-range` and `solution` NetCDF variables, namely that (4) holds:

$$||\mathbf{u}_{\texttt{numerical}} - \mathbf{u}_{\texttt{exact}}||_\infty < \epsilon \tag{4}$$

7. The following exercises will allow you to visualise the solution at every 10 time steps using the PLplot visualisation library. Please refer to Section 2 of the PLplot documentation at `http://www.nag.co.uk/market/training/fortran-workshop/plplot-5.11.1.pdf` for further information.

The visualistion will be done in the main program `fd1d_heat_explicit.f90` using the following sequence of subroutine calls:

- `PLPARSEOPTS( )` to parse command line options to control PLplot. This subroutine call should be done outside the main time loop
- `PLSFNAM( )` to set the output file name - all subroutines from now on should be called within the main time loop
- `PLSDEV( )` to set the output device to use. Set this to `"pngcairo"` which will save the images in the portable network graphics 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, add the line `-I/usr/lib64/gfortran/modules` - this path might be different on your system

(f) To link the code to the PLplot libraries including the Fortran wrappers, use `-L/usr/lib64 -lplplotf95cd -lplplotf95d`

(g) 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. The `%` wildcard is similar to `*` wildcard used in the Linux shell.

8. This exercise will involve creating pFUnit test codes. This exercise will only test the CFL subroutine.

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

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

  integer(KIND=SI), parameter :: t_num = 201
  integer(KIND=SI), 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
```

Place it in the same directory as the Fortran source code.

(b) Create the test configuration file `testSuites.inc` which will tell pFUnit which tests to execute:

```
ADD_TEST_SUITE(testCFL_suite)
```

(c) To preprocess the pseudo Fortran test driver code to produce Fortran code:

```
pFUnitParser.py testCFL.pf testCFL.F90 -I.
```

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:

```
gfortran -I$PFUNIT/mod -c testCFL.F90
```

where `$PFUNIT` is the environment variable which points to the installation directory of pFUnit

(e) Then create the final test driver executable:

```
gfortran -o tests.x -I$PFUNIT/mod $PFUNIT/include/driver.F90 \
         CFL_mod.o testCFL.o  -L$PFUNIT/lib -lpfunit -I.
```

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

(f) This command will create the `tests.x` binary executable which needs to be executed and will print the result of the test which is a pass

(g) Change the value `cfl_exact` to `0.34_DP` in the pseudo Fortran code and repeat steps (b), (c) and (d). Execute the `tests.x` which should fail the test

9. 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 your code accordingly. 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.