Tips and tricks for good (and fast) scientific programming, with and introduction to parallel computing

1 - Basics

Lecturer: Federico Perini

Madison, 2013/11/27

Lecture series outline

1. Basics of good programming practice

- Tools for good and comfortable code development and maintenance
- Good programming practice

2. Optimization, Debugging and Profiling

- Compiler-based vs. manual optimization
- Debugging tools, with examples
- Finding and solving bottlenecks in the code

3. Parallel programming

- Different tools for different applications
- Examples of code parallelization with OpenMP and MPI

This schedule is open to changes upon requests!

Lecture 1 – the basics

1. What does it mean to build a program
2. The KISS principle
3. Necessary programming practices
4. Q&A

What is a program

What is a program

```
! Example of Fortran 200X program
program hello
    implicit none

    call say_hi

    return

    ! Contained procedures
    contains

    subroutine say_hi()
      character(len=*), parameter :: fmt_hello = "('Hello World!')"
      write(*, fmt_hello)
    end subroutine say_hi

end program hello
```

Methods that operate on data structures

data structures

‘Building’ a program

= going from a set of source code files to an executable

1. Coding

- The process of writing source files containing the data structures and the methods that operate on them for our particular purpose, in a specific programming language syntax

2. Compiling

- The translation of the source code into machine language instructions
- Creates ‘object’ files or libraries (.o, .obj, .a, .dll, etc.) that are not language-bound anymore
- Look at one source file at the time and check for syntax errors

3. Linking

- The generation of an executable file from multiple object files
- Look at the global program structure
- Check that all the required functions/libraries are defined in the object files

Programming philosophies

Different approaches to how do data structures and methods interact

■ Procedural programming

- **Task-oriented** (subroutines, functions)
- Methods, tasks that operate on data structures of unknown origin

e.g.: We want to calculate an injection velocity

```
function injection_velocity(diam,p_amb,p_inj,fuel_dens,cD)

    real(8), intent(in) :: diam,p_amb,p_inj,fuel_dens,cD
    real(8)             :: injection_velocity

    injection_velocity = cD * sqrt(2.0*(p_inj-p_amb)/fuel_dens)

end function injection_velocity
```

Programming philosophies

- **Structured and object-oriented programming**
  - **Objects (instances of classes) have**
    - **Attributes (data)**
    - **Associated procedures (methods)**
  - **Methods and data structures are *encapsulated***
  - **Fortran:** `module/type/class`

e.g.: We want to calculate an injection velocity

- An injection velocity is not of general usage, it only makes sense within a certain representation of a fuel injector nozzle
- A fuel injector nozzle has some properties that are unique to that particular class of objects, e.g., a hole diameter, an injection pressure, a position in space, but whose values depend on the particular instance of that class, e.g. my PFI injector, your common rail injector, etc.

Programming philosophies

```
module injector_module
    implicit none
    private

    type, public :: injector_nozzle
        private

        ! Injector position
        real(8) :: x, y, z

        ! Nozzle geometry
        real(8) :: diameter

        ! Injection pressure
        real(8) :: injection_pressure

        ! Fuel properties
        real(8) :: rho_fuel


        contains

            set_geometry => injector_nozzle_set_geometry
            injection_velocity => injector_nozzle_velocity

    end type injector_nozzle


    contains
```

Data are encapsulated in an 'injector_nozzle' object

These are **not** modifiable unless made available through some interface function, e.g., set_geometry

Programming philosophies

```
subroutine injector_nozzle_constructor(this,d,p,rhol,x,y,z)
    class(injector_nozzle), intent(inout) :: this
    real(8)              , intent(in)    :: d,p,rhol,x,y,z

    ! Initialize nozzle position
    this%x = x
    this%y = y
    this%z = z

    ! Initialize geometry
    this%diameter = d

    ! Initialize operating conditions
    this%injection_pressure = p
    this%rho_fuel           = rhol

end subroutine injector_nozzle_constructor
```

We need to build our object, i.e., initialize its characteristic data properties

```
function injector_nozzle_velocity(this,p_ambient) result(vmag)
    class(injector_nozzle), intent(in) :: this
    real(8)               , intent(in) :: p_ambient
    real(8)                            :: discharge_coef
    real(8), parameter :: two = 2.d0

    discharge_coef = this%discharge_coefficient(p_ambient)

    vmag = discharge_coef &
         * sqrt(two*(this%injection_pressure-p_ambient)/this%rho_fuel)

end function injector_nozzle_velocity
```

Methods operate with these data without making them visible to the outside

Programming philosophies

```
program injector

    ! Header contains dependencies
    use injector_module, only: injector_nozzle
    implicit none

    ! Data
    type(injector_nozzle) :: my_nozzle
    real(8) :: p_amb

    ! Read nozzle geometry from input file
    read(myfile,*)diam,p_inj,rho_fuel,x,y,z

    ! Initialize nozzle properties
    call my_nozzle%create_nozzle(diam,p_inj,rho_fuel,x,y,z)

    ! Print injection velocity
    print *, my_nozzle%injection_velocity(p_amb)


end program injector
```

After object creation, there is no interaction anymore with its encapsulated properties

Suppose we want to implement a new nozzle flow model. We won't need to change the whole program! Changes will be confined to the injector object

Structured programming enhances maintainability and expandability, but both philosophies are equally valid, as long as code is good code

How to begin

- **Open-source Fortran compiler suite**
  http://gcc.gnu.org/wiki/GFortran
  http://www.equation.com (Windows)
  http://finkproject.org, http://www.macports.org (Mac)

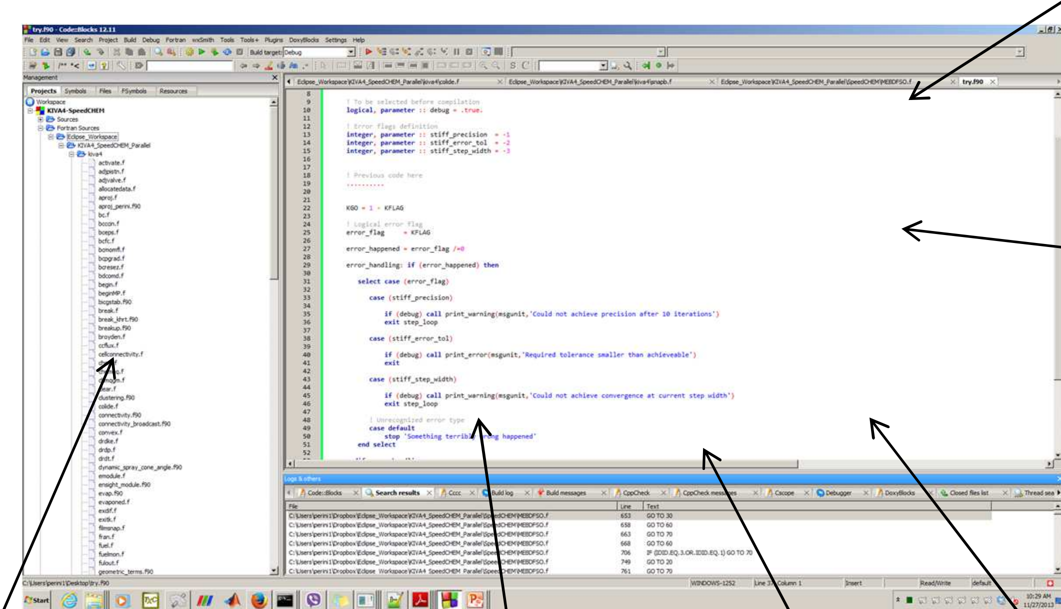
- **Open-source IDE with Fortran support**
  http://www.codeblocks.org/

- **Open-source MPI library**
  http://www.mpich.org/downloads/

How to begin

- An IDE is structured in terms of **Projects**

Customizable (plugins, scripts)



Handles debugging

Automatic makefiles

Constant access to all the source files

Syntax highlighting and code completion
➔ Including variables, methods, classes

Custom build methods

The KISS principle

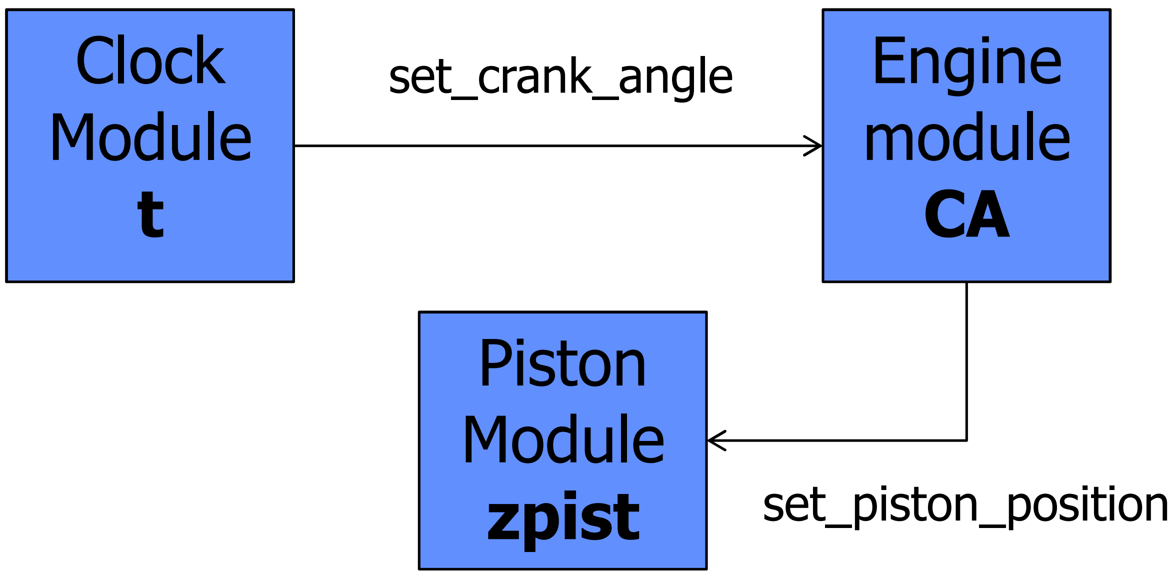
the KISS principle

- **Keep it simple, stupid**

- **Every construct should be**
  - **<u>self-consistent</u>: do not require unnecessary information from the outside**
    - **Just use variable input-output**
  - **<u>complete</u>: perform all the requested operations in the same subroutine/function**
  - **<u>single</u>: do not perform more than one task in every subroutine/function**

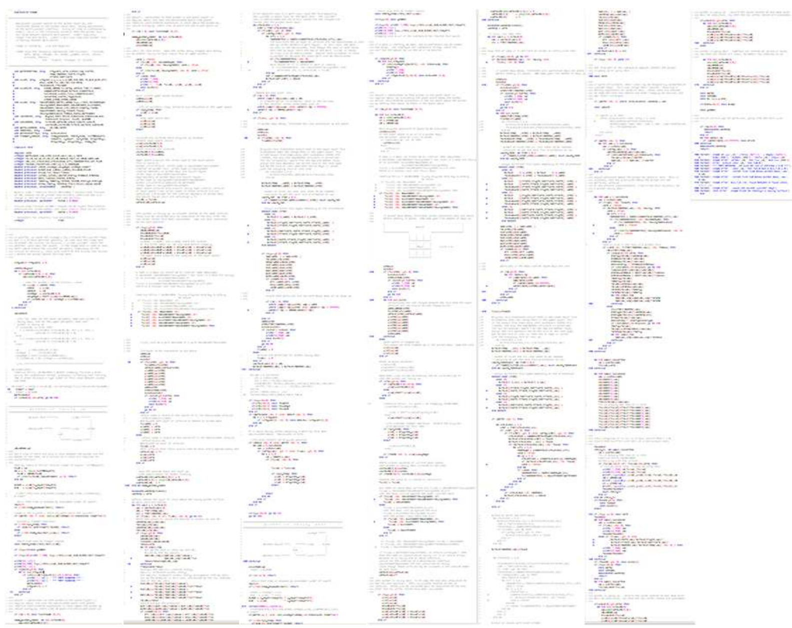
the KISS principle

- **See the code as a series of 'black' boxes that interact each other**
  (Modules or Classes can be very good boxes)
- **Ideally, the data relevant to every module/class should only be known within that module/class, and no data should be shared among them**, unless communicated through calls to public functions/subroutines



the KISS principle

- **Every function/subroutine should not be longer than one page**
  - We are not superheroes, handling long scripts that do not fit the page (screen) is extremely prone to errors

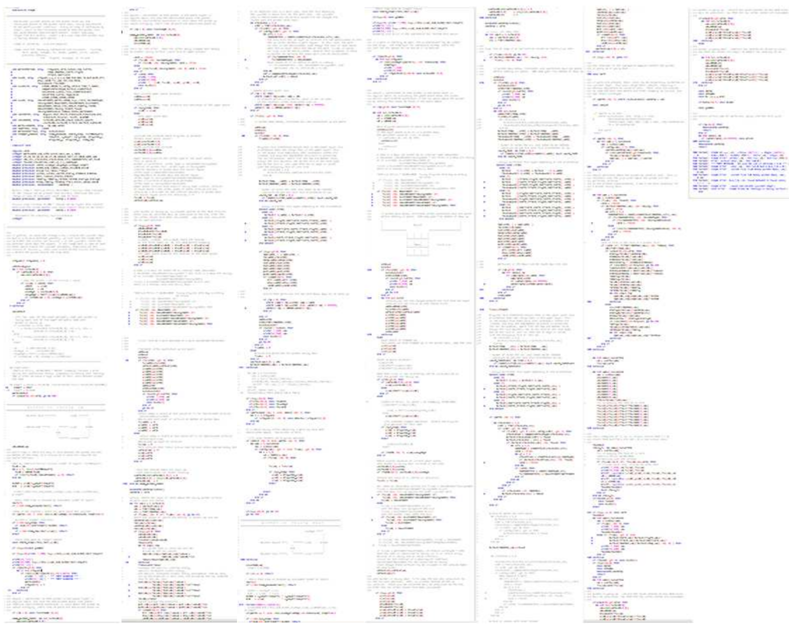


KIVA squish layer snapper

snapb.f
- 1362 lines
- 18 pages (print)

the KISS principle

- **If it is longer, probably there is some sub-task that can be contained in a separate construct**



KIVA snapb.f

- Find moving surface
- Understand moving direction
- Decide if a snap is needed
- Add / remove a layer
- Interpolate physical quantities

➔ This would apply to any moving surface in the domain!

Guidelines and tips for good programming practice

Use meaningful names

- **Stick to a suitable naming_convention, namingConvention**
- **Consistency between filename, contained modules and related type structures, e.g.**

  polygons_mod.f90

```
module polygons_mod
  implicit none

     type, public :: polygons
     …
     end type

contains
  …
end module polygons_mod
```

- **The more descriptive the better**

```
CALL CKHTYW(X,Y,KB22,RWRK,IWRK,IPAR)
```

Use meaningful names

Use Logical Names

- Subroutine names
  - newkiva.f
  - newnewkiva.f
  - sortanewkiva.f
  - newnewnewnewnewnewnewnewnewnewnewnewnew newnewnewnewnewnewnewnewnewnewnewnewnew newnewnewnewkiva.f
  - kiva_v7-09.f (note: dates are unique)

- Making many changes and not sure if they will be permanent? Copy a routine and give it a 'version' name:
  - newchem.f -> chem_cjr_v7-09.f

23

- **Credit: prof. C.J. Rutland, "Notes on Programming", 2009**

Use comments

- **At least 50% of the code should be occupied by comments!**

- **Header comments**
  - **Briefly describe what the routine / data structure is related to**
  - **Describe input/output/storage properties**
  - **List references to the algorithm**
  - **Keep track of subroutine changes and updates**

- **Comments within sections**
  - **if constructs / do loops / etc.**
  - **Comment what is being done**

- **Subroutine updates**
  - **When changing something, always write down name of the coder and date**

Use comments

- **Example**

```
! [FP] 1/1/1970
real(8), parameter :: pi = acos(-1.d0)

…
! [FP] 1/1/1970
! Hardcoded constant moved to parameters
! circ = 2 * 3.14 * radius
circ = 2 * pi * radius

…
```

Use comments

- **Always label do/if/case constructs**
- **Use logical variables with meaningful names**

➔ The code should mimic a language's semantic, and be almost human-readable

```
      KGO = 1 - KFLAG
      IF (KGO.EQ.1) THEN
C        NORMAL RETURN FROM STIFF
         GO TO 30

      ELSE IF (KGO.EQ.2) THEN
C        COULD NOT ACHIEVE REQUIRED PRECISION WITH HMIN
C        SO CHOP HMIN IF WE HAVEN'T DONE SO 10 TIMES
         GO TO 60

      ELSE IF (KGO.EQ.3) THEN
C        ERROR REQUIREMENT SMALLER THAN CAN BE HANDLED FOR THIS PROBLEM
         WRITE (LOUT,9010) T,H
         GO TO 70

      ELSE IF (KGO.EQ.4) THEN
C        COULD NOT ACHIEVE CONVERGENCE WITH HMIN
         WRITE (LOUT,9030) T
         GO TO 60
      ENDIF

   30 CONTINUE
```

- What is KGO?
- Where is the code jumping to at any of these conditions?
- Why do we have to specify a costly **if** clause if the code only has to continue?
- What is 9010 and 9030?
- What do I have to do to introduce a further error case?

Self-commenting code

- the code self-comments itself
- Error codes are stored and saved as **parameters** in a safe place, e.g. subroutine header or in a module
- The compiler will interpret them as numbers, but we can read their meaning
- Optional screen output handled with a 'debug' parameter → the compiler will just remove this lines during compilation if it is .FALSE.
- Easy to add handling for further error types
- We may put this into a subroutine if no special actions have to be taken

```
      KGO = 1 - KFLAG
      IF (KGO.EQ.1) THEN
C        NORMAL RETURN FROM STIFF
         GO TO 30

      ELSE IF (KGO.EQ.2) THEN
C        COULD NOT ACHIEVE REQUIRED PRECISION WITH HMIN
C        SO CHOP HMIN IF WE HAVEN'T DONE SO 10 TIMES
         GO TO 60

      ELSE IF (KGO.EQ.3) THEN
C        ERROR REQUIREMENT SMALLER THAN CAN BE HANDLED FOR THIS PROBLEM
         WRITE (LOUT,9010) T,H
         GO TO 70

      ELSE IF (KGO.EQ.4) THEN
C        COULD NOT ACHIEVE CONVERGENCE WITH HMIN
         WRITE (LOUT,9030) T
         GO TO 60
      ENDIF

 30   CONTINUE
```

```
subroutine integrator

      ! Previous code here
      ..........

      logical :: error_happened
      integer :: error_flag

      ! To be selected before compilation
      logical, parameter :: debug = .true.

      ! Error flags definition
      integer, parameter :: stiff_precision  = -1
      integer, parameter :: stiff_error_tol  = -2
      integer, parameter :: stiff_step_width = -3


      ! Previous code here
      ..........


      KGO = 1 - KFLAG

      ! Logical error flag
      error_flag     = KFLAG
      error_happened = error_flag /=0

      error_handling: if (error_happened) then

         select case (error_flag)

            case (stiff_precision)

                if (debug) call print_warning(msgunit,'Could not achieve precision after 10 iterations')
                exit step_loop

            case (stiff_error_tol)

                if (debug) call print_error(msgunit,'Required tolerance smaller than achieveable')
                exit

            case (stiff_step_width)

                if (debug) call print_warning(msgunit,'Could not achieve convergence at current step width')
                exit step_loop

            ! Unrecognized error type
            case default
                stop 'Something terribly wrong happened'
         end select

      endif error_handling

      ! Rest of code here
      ..........


end subroutine integrator
```

Do not duplicate code

- **If a series of lines has to be copied and pasted it means that it is representing a task that can be included in a subroutine/function**
- **This may be slower at runtime due to overhead for calling the procedure, but let in-lining to be decided by the compiler (next week)**
- **Cannot change / add more instructions without changing many parts of the code**
- **Code reusability is compromised**
- **Duplicating data multiplies the chances of errors**

Do not duplicate code

- Never do any expensive calculation twice! → storage/retrieval is always faster on modern computing architectures that do not have tight memory bounds

- Never change more than one thing at the same time!

Avoid scattered I/O

- Reading/writing data from/to disks is orders of magnitude slower than from/to memory
- Also screen output introduces interaction with the operating system → slow
- Always confine all I/O in very specific parts of the code
- Debugging I/O should be removed when not needed

Avoid hardcoding

- **Always think as <u>everything</u> in your code may have to be extended/modified at some point in the future**

→ Never include numbers in the code, even if they are constants

```
if ( crank >= -65.3 ) then

   do i = 1, 7
       inj_mass(i) = dt * eff_area * 820.1 * v_inj(i)
   end do

endif
```

```
injection_timing: if ( crank >= start_of_injection ) then

   loop_over_injectors: do i = 1, n_inj

       inj_mass(i) = dt * eff_area * fuel_dens * v_inj(i)

   end do loop_over_injectors

endif injection_timing
```

Have totems, but do not stick to taboos

- **Programming languages evolve pretty much as human languages do, to make communication simpler and more effective**

- **Complex programs always feature more than one programming language**

> Decide few simple guidelines,
> and then use the most appropriate
> language for your needs

Read before doing!

- **The Internet**

  http://fortranwiki.org

- **Metcalf, Reid, Cohen – "Modern Fortran explained", Oxford University Press**

- **Brainerd – "Guide to Fortran 2003 Programming", Springer**

Questions? Example requests?