Taking up development of existing research codes

Dr. Spiridon Siouris
s.siouris@sheffield.ac.uk
Low Carbon Combustion Centre (LCCC), Mechanical Engineering
Contents

• My background
• Issues with developing software for a research project
• How to quickly get familiarised with a coding project
• Good programming practises
• Holistic view on optimisation
• Summary
My background

• Research fellow in fluids (Mech.Eng.)
• Worked on several projects in Low Carbon Combustion Centre and Aerodynamics groups such as:
  - Modelling of fuel deposition in gas turbine injectors (FINCAP)
  - Modelling of fuel and lubricant thermo-chemical degradation in gas turbine systems
  - Aerodynamics, active flow control and plasma modelling
My background

• No formal CS education, but I like:
  – Computers/Programming/HPC
  – Tidy and efficient things
  – Adhering to standards

• Lots of experience in software development for modelling fluids and/or chemical reacting flows over many years
Software in academia

• Great opportunities for developing very interesting software, but...
• Projects often start late
• Results driven – the quicker the better and no questions asked!
• Often written by researchers where programming is not part of their formal education or primary focus
• PI’s usually not involved at programming level to provide input
Software in academia

• Aim to generate results often leading to quick and dirty programming!

• Software generated without adhering to good programming practices

• Code is passed on from one researcher to another

• Elements are added on and on without any restructuring of code
Software in academia

- Lots of software generated with high impact, but:
  - Time is lost figuring out what various things do in the software
  - Can end up with several versions of the same software but very little information, if lucky!
  - Not written in the best way...

- Principle of least astonishment
Software in academia

• Lots of software generated with high impact, but:
  - Time is lost figuring out what various things do in the software
  - Can end up with several versions of the same software but very little information, if lucky!
  - Not written in the best way...

• Principle of least astonishment
Getting to know a project

• Look at a random source file. How is the formatting/coding? Did people take care? Is it easy to understand?
Getting to know a project

• Compile to check coding errors/warnings using:
   - gcc -Wall -Wextra -Wpedantic

• Run and check for memory leaks using:
   - valgrind --leak-check=full

• Generate documentation using:
   - doxygen
   - Suitable for C/C++, Fortran, bash, python, Java, TCL
   - Visualise relations between functions
   - Navigate around parts of code with hyperlinks
   - Can output HTML, pdf
Getting to know a project
Getting to know a project - summary

1. Have a look at a few source files
2. Compile with all warnings on
3. Run with memory checker
4. Generate documentation
Working on a project

1. Backup what you receive (ie `projectname_date.tar.gz`)

2. Set up source code version control (I like `git`)

3. Backup again

4. Ground rules. Discuss with PI or main developer the extent of your input (ie what is your freedom to change code beyond your scope?)

5. Be kind and avoid: ```Code is rubbish and needs major work```!
Working on a project

6. Carry on with your coding work

7. If you see anything that needs changing, then change it! (ie variables, functions, data structures)

8. Use GCC during development, and Intel for production
   • Intel is good for compilation and running speed, but supports non standard features – not portable!
   • GCC is good for errors and warnings, but slower than Intel (not by much though!) - portable!
Good programming practises

• Naming variables
  – Should be clear and meaningful without ambiguity (and don’t try to be funny)
  – No coded variables
  – Should not need any comments
  – Searchable, ie no single-letter variables
  – Length of name should roughly correspond to the size of its scope
Good programming practises

```c
double rate_of_progress[TOTAL REACTIONS];
int i=0;
int j=0;

/* get the pointer and values from user data structure */
double *reaction_rate_constants = (double *)user_data;

/* calculate the rate of forward progress, assuming there is no reverse */
* progress rate_of_progress[i] =
* reaction_rate_constants[i]*SUM_PROD(mass_fractions[j]*stoich_matrix[i][j]) */
for (i=0; i<TOTAL_REACTIONS; i++)
    rate_of_progress[i] = reaction_rate_constants[i];
for (j=0; j<TOTAL_SPECIES; j++)
    if (stoich_matrix[i][j] < 0)
        rate_of_progress[i] *= pow(mass_fractions[j],(-1)*stoich_matrix[i][j]);

/* account for any third bodies */
for (i=0; i<TOTAL_REACTIONS; i++)
    for (j=0; j<TOTAL_SPECIES; j++)
        if (collision_efficiency[i][j] > 0)
            third_body_coeff *= collision_efficiency[i][j]*mass_fractions[j];
    if (third_body_coeff > 0)
        rate_of_progress[i] *= third_body_coeff;

/* calculate the net rate of production or destruction net_rate */
for (j=0; j<TOTAL_SPECIES; j++)
    net_rate[j] = 0;
for (i=0; i<TOTAL_REACTIONS; i++)
    net_rate[j] += stoich_matrix[i][j]*rate_of_progress[i];

return 0;
```

```c
void calc_reaction_rate(const reactor_data *reactor, kinetic_parameters *arrhenius_coefficients) {
    int i=0;
    const double R_times_temp = R*(reactor->temperature);

    switch (ARRHENIUS_PARAMETERS) {
        case (2):
            for (i=0; i<TOTAL_REACTIONS; i++)
                reaction_rate_constants[i] =
                    arrhenius_coefficients->A[i] * 
                    exp((-arrhenius_coefficients->E[i])/((R_times_temp)));
Good programming practices

• Functions
  – The shorter the better
  – Two to three arguments but no more
  – Do one thing
  – No side effects
  – Error handling
  – Should always test in isolation
Good programming practices

```c
31 void calc_reaction_rate_const(reactor_data *reactor, kinetic_parameters *arrhenius_coefficients);
32 void calc_reactor_steps_and_error(reactor_data *reactor, timing_data *timings);
33 void prepare_data_file(reactor_data *reactor, double current_time);
34 void print_header(reactor_data *reactor);
35 void print_mass_fractions(double current_time, reactor_data *reactor);
36 void fprintf_reactor_conditions(FILE *operating_conditions, reactor_data *reactor);
37 static int degrade_reactor(reactor_data *reactor, timing_data *timings, cvode_data *cvode);
38 void copy_mass_fractions(double *source, double *destination);
39 void copy_reaction_rate Const(double *source, double *destination);
40 void linearly_interpolate_mass_fractions(reactor_data *reactor, double const time_increment);
41 static int check_flag(void *flagvalue, char *funcname, int opt);
42 void calculate_dep_thick(reactor_data *reactor, kinetic_parameters *arrhenius_coefficients);
43 double average_mass_fraction(int const species_index, reactor_data *const reactor);
44 double approx_1_minus_sqrt_1_minus_epsilon(double const epsilon);
```
Good programming practices

```c
double rate_of_progress[TOTAL_REACTIONS];
// Rate of progress
int i = 0;
/
/* get the pointer and values from user data structure */
double *reaction_rate_constants = (double *)user_data;
/
// Calculate the rate of forward progress, assuming there is no reverse
/* progress rate_of_progress[] = */
/* reaction_rate_constants[i] + sum_of_mass_fractions[i]*stoich_matrix[i][j] */
for (i = 0; i < TOTAL_REACTIONS; i++) {
    rate_of_progress[i] += reaction_rate_constants[i];
    for (j = 0; j < TOTAL_SPECIES; j++) {,
        if (stoich_matrix[i][j] < 0) {
            rate_of_progress[i] += pow(mass_fractions[j], -1)*stoich_matrix[i][j];
        }
    }
}
/
/* account for any third bodies */
for (i = 0; i < TOTAL_REACTIONS; i++) {
    for (j = 0; j < TOTAL_SPECIES; j++) {,
        if (collision_efficiency[i][j] > 0) {
            third_body_coeff *= collision_efficiency[i][j]*mass_fractions[j];
        }
    }
}
/!
if (third_body_coeff > 0) {
    rate_of_progress[i] *= third_body_coeff;
}
/
// Calculate the net rate of production or destruction net_rate */
for (i = 0; i < TOTAL_SPECIES; i++) {
    net_rate[i] = 0;
    for (j = 0; j < TOTAL_REACTIONS; j++) {,
        net_rate[i] += stoich_matrix[i][j]*rate_of_progress[j];
    }
}
/!
return 0;
```

```c
void calc_reactor_steps_and_error(reactor_data *reactor, timing_data *timings) {
    double computed_residence_time = 0;
    reactor->steps = (int)(reactor->residence_time/timings->time_increment);
    computed_residence_time = reactor->steps * timings->time_increment;
    reactor->unmodelled_percent_volume = (1 - computed_residence_time / reactor->residence_time) * 100;
}
```

```c
void allocate_mass_fractions(reactor_data *reactor) {
    reactor->internal_mass_fractions = (double **)malloc(reactor->steps * sizeof(int *));
    reactor->internal_mass_fractions[0] = (double *)malloc(reactor->steps * TOTAL_SPECIES * sizeof(double));
    for (int step = 0; step < reactor->steps; step++) {
        reactor->internal_mass_fractions[step] = *(reactor->internal_mass_fractions + step * TOTAL_SPECIES);
    }
}
```

```c
void initialise_internal_mass_fractions(reactor_data *reactor) {
    for (int step = 0; step < reactor->steps; step++) {
        initialise_mass_fractions(reactor->internal_mass_fractions[step]);
    }
}
```

```c
void initialise_mass_fractions(double *mass_fractions) {
    for (int i = 0; i < TOTAL_SPECIES; i++) {
        mass_fractions[i] = initial_mass_fractions[i];
    }
}
```

```c
void initialise_reaction_speeds(double *reaction_rate_constants) {
    for (int i = 0; i < TOTAL_REACTIONS; i++) {
        reaction_rate_constants[i] = 0;
    }
}
```

```c
void calcReactionRateConstant(reactor_data *reactor, kinetic_parameters *arrhenius_coeffs) {
    int i = 0;
    const double R_times_temp = R * (reactor->temperature);
    switch (ARRHENIUS_PARAMETERS) {
        case [1]:
            for (i = 0; i < TOTAL_REACTIONS; i++) {
                reaction_rate_constants[i] =
                exp(-arrhenius_coeffs->endl/R_times_temp);
            }
    }
}
Good programming practises

• Commenting
  − Use with caution
  − Better to improve naming of functions, data variables than to rely on comments
  − Useful for citing articles, algorithm names, etc. where appropriate
  − Watch out for line length. Do not assume everyone has a large wide screen monitor
Good programming practices

```c
double rate_of_progress[TOTAL_REACTIONS]; /*< Rate of progress */
int i=0;
int j=0;

/* get the pointer and values from user data structure */
double *reaction_rate_constants = (double *)user_data; /* Rate constant of each reaction */

/* calculate the rate of forward progress, assuming there is no reverse */
* progress rate_of_progress[i] =
  * reaction_rate_constants[i]*SUM_PROD(mass_fractions[j]*stoich_matrix[i][j]) */
for (i=0; i<TOTAL_REACTIONS; i++) {
    rate_of_progress[i] = reaction_rate_constants[i];
    for (j=0; j<TOTAL_SPECIES; j++) {
        if (stoich_matrix[i][j] < 0) {
            rate_of_progress[i] *= pow(mass_fractions[j],(1)*stoich_matrix[i][j]);
        }
    }
}

/* account for any third bodies */
for (i=0; i<TOTAL_REACTIONS; i++) {
    for (j=0; j<TOTAL_SPECIES; j++) {
        if (collision_efficiency[i][j] > 0) {
            third_body_coeff = collision_efficiency[i][j]*mass_fractions[j];
        }
    }
    if (third_body_coeff > 0) {
        rate_of_progress[i] *= third_body_coeff;
    }
}

/* calculate the net rate of production or destruction net_rate */
for (j=0; j<TOTAL_SPECIES; j++) {
    net_rate[j] = 0;
    for (i=0; i<TOTAL_REACTIONS; i++) {
        net_rate[j] += stoich_matrix[i][j]*rate_of_progress[i]
    }
}
return 0;

void calc_reaction_rate_const(reactor_data *reactor, kinetic_parameters *arrhenius_coefficients) {
    int i=0;
    const double R_times_temp = R*reactor->temperature;
    switch (ARRHENIUS_PARAMETERS) {
    case 2:
        for (i=0; i<TOTAL_REACTIONS; i++) {
            reaction_rate_constants[i] = 
                arrhenius_coefficients->Arrhenius_expo(reactor->temperature,
                exp((arrhenius_coefficients->E[i])/(R_times_temp));
        }
    ```
Good programming practices

• Further reading:
  – Martin R. C., 2013, Clean code, A handbook of agile software craftsmanship, Prentice Hall
  – Ortiz P. F., 2018, First steps in scientific programming, Amazon
Good programming practises

- Further reading
  - ASTG coding standard, Recommended coding styles for software development in Fortran, C++, Java, and Python, Ver. 1.7, 2015
Good programming practices

- Further reading for C/C++
  - NASA C style guide, August 1994, SEL-94-003
  - SEI CERT C coding standard, Rules for developing safe, reliable and secure systems in C++, 2016
  - SEI CERT C++ coding standard, Rules for developing safe, reliable and secure systems in C++, 2016
  - High integrity C++ Coding Rules, Programming Research Ltd., www.codingstandard.com
  - Joint strike fighter air vehicle C++ coding standards for the system development and demonstration program, December 2005
Good programming practises

- Further reading for Fortran
  - Fortran coding standards for new JULES code, Joint UK land environment simulator, June 2010
  - European standards for writing and documentign exchangeable Fortran 90 code, Ver. 1.1, Andrews P., 1995
Optimising code

- Very subjective and debatable topic
- Done to save time and computational resources
- Which one are you most interested in saving?
- Scrutinise every advice you receive, including mine
- Work out what is best for you
- Let’s start with a quote...
Optimising code

The real problem is that programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times; premature optimization is the root of all evil (or at least most of it) in programming.

Optimising code

• Rule No. 1: Code optimisation should be the last thing you should do.
  – More important to adhere to good programming practises
  – Make sure code is as scalable as possible
  – Find something else to optimise in your workflow

• Rule No. 2: If you need to do it, use proper evidence for it
  – No data – no optimising
  – Use profilers for performance analysis, eg gprof for single-core, and/or tau for multi-core
Optimising code

- Example of a typical week of mine:

- What is taking up most of my time?
- Does it make sense for me to put effort to speed up HPC running?
- How about your workflow?
• Better for me to spend time on:
  – Making the code more understandable - Good programming practises!
  – Automating post processing

• Organising order of tasks is very important too as simulations can be running in the background
  – Is it possible to submit a job before a 2 hour meeting? If so, 2 hours saved!
Before optimising code

• Make sure the code is correct and bug free
• Make it portable
• Are the algorithms numerically stable/fast? What is the latest literature?
• Use libraries as much as possible as they are (most likely) already efficient - no point reinventing the wheel
Before optimising code

- Do you have a table with a design of experiments?
- Are the simulations run with the correct settings, boundary conditions, etc?
- Is the time/space discretisation suitable enough?, and do you really need a 1M cell mesh?
- After having checked all the above, then....
Optimising code

- Learn (briefly) how CPU’s work, eg registers, cache
- Too many techniques to list here
- Very good read (old, but still useful)
  - Severance C. and Dowd K., High performance computing (RISC architectures, optimization and benchmarks), 2nd ed., O’Reilly, 1998
  - Or even look at anything from the early days of game programming, LAPACK, BLAS, etc where every kB of memory and CPU cycles matters!
- Plenty other references exist that are more recent, but the core knowledge remains the same
Optimising code

- Profile first to generate data and then optimise the most time consuming parts of the code

```c
double rate_of_progress[TOTAL_REACTIONS];
/** Rate of progress of
  3 4
  5 int i=0;
  6 int j=0;
  7
  8 // get the pointer and values from user data structure */
  9 double *reaction_rate_constants = (double *)user_data;
  10 /** Rate constant of a rea
  11 for (i=0; i<TOTAL_REACTIONS; i++) {
  12   reaction_rate_constants[i] = reaction_rate_constants[i] * stoich_matrix[i][i];
  13   for (j=0; j<TOTAL_SPECIES; j++) {
  14     if (stoich_matrix[i][j] < 0) {
  15       rate_of_progress[i] *= pow(mass_fractions[i], j) * stoich_matrix[i][j];
  16     }
  17   }
  18 }
  19
  20 // calculate the rate of forward progress, assuming there is no reverse
  21 // progress rate of progress[i] =
  22 // reaction_rate_constants[i] * Tcıp * (mass_fraction[i] * stoich_matrix[i][i])
  23 for (i=0; i<TOTAL_REACTIONS; i++) {
  24   rate_of_progress[i] = reaction_rate_constants[i] * stoich_matrix[i][i];
  25   for (j=0; j<TOTAL_SPECIES; j++) {
  26     if (stoich_matrix[i][j] < 0) {
  27       rate_of_progress[i] *= pow(mass_fractions[i], j) * stoich_matrix[i][j];
  28     }
  29   }
  30 }
  31
  32 // account for any third bodies */
  33 for (i=0; i<TOTAL_REACTIONS; i++) {
  34   for (j=0; j<TOTAL_SPECIES; j++) {
  35     if (stoich_matrix[i][j] > 0) {
  36       third_body_coef = collision_efficiency[i][j] * mass_fractions[j];
  37     }
  38   }
  39   if (third_body_coef > 0) {
  40     rate_of_progress[i] *= third_body_coef;
  41   }
  42 }
  43
  44 // calculate the net rate of production or destruction net_rate */
  45 for (i=0; i<TOTAL_SPECIES; i++) {
  46   net_rate[i] = 0;
  47   for (j=0; j<TOTAL_REACTIONS; j++) {
  48     net_rate[i] += stoich_matrix[i][j] * rate_of_progress[j];
  49   }
  50   return net_rate;
  51 }
  52
  53 void calc_reaction_rate(const reactor_data *reactor, kinetic_parameters *Arrhenius_coefficients) {
  54   int i=0;
  55   const double R_times_temp = R * (reactor->temperature);
  56   switch (Arrhenius_parameters) {
  57     case [3]
  58   for (i=0; i<TOTAL_REACTIONS; i++) {
  59     // calculate reaction rate
  60     reaction_rate_constants[i] = 
  61     arrhenius_coefficients->all[i] * 
  62     exp(-arrhenius_coefficients->all[i] * R_times_temp));
  63   ```
Optimising code
Optimising code

- Optimise with focus on memory as well as CPU cycles and communication between cores
- Memory is not cheap and can get consumed very easily if not careful

Fig. 5. Memory hierarchy. Typical latencies for data transfers from the CPU to each of the levels are shown. The numbers shown here are only an indication, and the actual numbers will depend on the exact architecture under consideration.

Summary

• Try and write quality code with focus on being:
  - Readable, simple, and therefore maintainable
  - Correct, error/warning free
  - Portable
  - Stable
  - Can handle errors where most likely to occur

• Optimise your whole workflow, and lastly work on speeding up your code

• If you optimise be careful not to speed-up at the expense of code quality

• Good luck!