Slide 1

OLD DOMINION UNIVERSITY

Workflow

A. Godunov

"If one approaches a problem with order and method, there should be no difficulty in solving it; none whatever."
Agatha Christie (from stories about Hercule Poirot)

1

Slide 2

The objectives in writing a code

"Never put your actions ahead of your objectives"

1. Correct
2. Efficient: you want to get your Thesis or/and Ph.D. in reasonable time (writing programs + run time should be minimized)
3. Maintainable: revise and resubmits, extensions of existing papers.
4. Reproducible: other researchers (and your future selves) must be able to replicate your results.
5. Documented: other researchers (and your future selves) must be able to understand how it works.
6. Scalable: code that can be used by you and by other researchers as a base for further development.
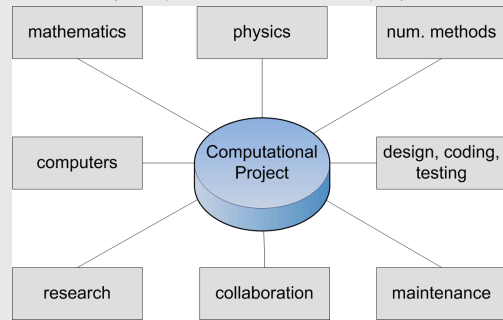7. Portable: code that can work across a reasonable range of machines.

2

Slide 3

Part 1:

Major parts

3

Slide 4

Computational projects

Computational Physics is an art (requires imagination and creativity) and science (uses specific methods and techniques)

mathematics	physics	num. methods
computers	Computational Project	design, coding, testing
research	collaboration	maintenance

4

Slide 5

Workflow

1. Problem definition
2. Problem analysis (check also for available codes)
3. Equations and data
4. Computational project (detailed design)
5. Numerical model(s) & libraries
6. Program coding (writing a computer code)
7. Get the code running (data flows, bugs)
8. Testing
9. Calculations and analysis of results
10. Program maintenance

5

Slide 6

Average distribution of efforts (after physics)

1. Computational project (design)   20%
2. Numerical model(s) & libraries   5%
3. Pseudo code   15%
4. Coding (using some language)   10%
5. Get the code running (data flows, bugs) 15%
6. Testing   30%
7. Documentation   5%

6

Slide 7

Steps 1-3 :

Physics

"I think you should be more explicit here in step two."

7

Slide 8
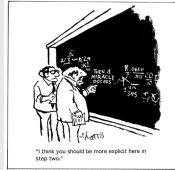
Most important

- You must have a clear understanding of your problem
- You must analyze what is already available and what needs to be done
- You must know very well physics involved

8

8

Slide 9

Step 4:

Design

9

Slide 10

Design: Two major approaches

- Top-down design
  (or hierarchical approach)
  Break a problem into a set of subtasks (called modules) until you are at the subroutine level
- Object-oriented design
  A problem-solving methodology that produces a solution to a problem in terms of self-contained entities called objects

What is better for Physics?

10

10

Slide 11

Design: practical advice

- o Consider available hardware, software and available time.
- o Arrange major tasks in order in which they need to be accomplished
- o Draw diagrams
- o For each module (subroutine)
  - o have well-defined input and output
  - o make it reasonably independent from other modules
- o Use clear logic and data structure
- o You program should be easy to use and modify
- o Protect your program from invalid inputs

11

11

Slide 12

Design: useful books

See also 100 Best Software Design Books of All Time
https://bookauthority.org/books/best-software-design-books

12

12

2

Step 5:

Numerical models and libraries

13
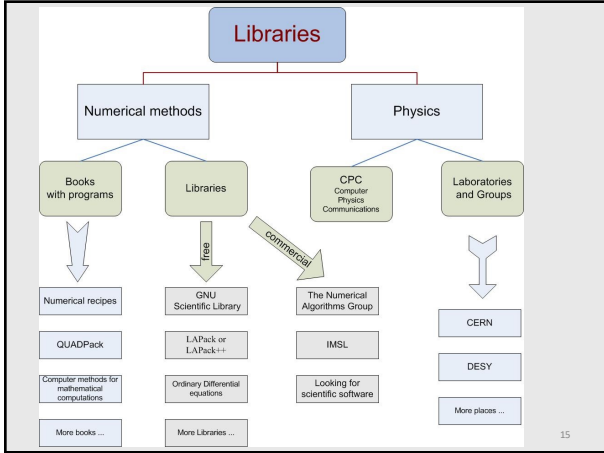

You should not reinvent the wheel

Attention: Use only trusted libraries!

- Computational Physics Libraries
- Numerical Libraries and Depositories

www.odu.edu/~agodunov/computing/lib_net.html

Attention: do not use routines as black boxes without understanding

14

14

15

15

Step 5:

Writing a code

16

Basic advice for writing a code

- Stick to the design
- Use pseudocode first
- Select widely a programming language
- Style: DOs and DON'Ts

Writing Scientific Software
A GUIDE TO GOOD STYLE
SUELY OLIVEIRA & DAVID STEWART

17

Style: Dos and DON'Ts

There are very many DOs and DON'Ts.

A couple examples

DO

- keep it clear and simple
- separating logic groups (structured programming)
- internal comments and external documentation
- linear programming

DON'T

- tricky technique
- complex logic
- platform (hardware, software) dependent
- steer clear of interactive or graphics-related routines.

18

18

Style: Each program unit should be well documented

- Opening documentation (what program does, when the program was written and modified, list of changes, …)
- Comments to explain key program sections
- Meaningful identifiers
- Labels for all output data

19

19

Style: A program should be readable

- Use spaces
- Use blank lines between sections
- Use alignment and indentation to stress relations between various sections of the program unit
- Labels for all output data
- Do not use "magic numbers" that appear without explanation

20

20

Programming hints

- Always keep an updated working version of your program; make modifications on a copy
- Use the standard version of the program language (easy to move to a different computer)
- Use descriptive names for variables and subroutines
- Declare ALL variables
- Do not optimize the program until you have right results

21

21

Style and habits

"Old habits die hard, I guess... if you don't kick them, they kick you"

22

22

Step 7:

Getting your code running

23

Bugs, bugs, bugs …

A computer bug is an error, flaw, mistake, failure, or fault in a computer program that prevents it from working correctly or produces an incorrect result.

24

24

4

Software horror stories

- NASA Mariner 1 went off-course during launch, due to a missing 'bar' in its FORTRAN software (July 22, 1962)
- NASA Mars Rover freezes due to too many open files in flash memory (January 21, 2004).
- The Mars Orbiter crashed in September 1999 because of a "silly mistake": wrong units in a program
- The year 2000 problem, popularly known as the "Y2K bug"
- The MIM-104 Patriot bug - rounding error, which resulted in the deaths of 28 solders (February 25, 1991).
- August 1991 – Sleipner A oil rig collapse (large elements in the Finite Element method) for solving PDE
- The 1988 shooting down of the Airbus 320 by the USS Vincennes was attributed to the cryptic and misleading output displayed by the tracking software.

25

Most expensive bug …

Ariane 5 Flight 501 the most expensive computer bugs in history (June 4, 1996 – cost was about 370 millions of dollars).

The Ariane 5 software reused the specifications from the Ariane 4, but the Ariane 5's flight path was considerably different and beyond the range for which the reused code had been designed.

Because of the different flight path, a data conversion from a 64-bit floating point to 16-bit signed integer value caused a hardware exception (an arithmetic overflow).

This led to a cascade of problems, culminating in destruction of the entire flight.

Result: self-destruction in 40 seconds after takeoff

26

Common types of computer bugs

- Inversion of logical tests
- Exceeding array bounds (can be difficult to catch it!)
- Off by one error - a loop iterates one too many or one too few times
- Using an uninitialized variable!
- Divide by zero
- Infinite loops
- Arithmetic overflow or underflow
- Accessing memory not owned (Access violation)
- Deadlock
- Loss of precision in type conversion

27

Searching for bugs … working like a detective

- Check global logic
- Check modules
    - Local logic
    - Data
    - Operators
    - Arrays
- Debugging

28

Step 8:
Testing

29

Verification and validation

Programs MUST NOT be used for research or applications until they have been validated!

One must first verify and validate each component, and then do the same for progressively larger ensembles of interacting components until the entire code has been verified and validated.

- Plan ahead
- Analytical solutions
- Other calculations
- Experiment
- Trends (does it make sense?)
- Special cases

Computational Science Demands a New Paradigm

The field has reached a threshold at which better organization becomes crucial. New methods of verifying and validating complex codes are mandatory if computational science is to fulfill its promise for science and society.

efficiently exploit the capacities of the increasingly complex computers. The prediction challenge is to use all that computing power to provide answers reliable enough to form the basis for important decisions.

January 2005   Physics Today   **35**

30

From Physics Today (January 2005)

- The few existing studies of error levels in scientific computer codes indicate that the defect rate is about seven faults per 1000 lines of a code. That's consistent with fault rates for other complex codes in areas as diverse as computer operating systems and real-time switching.
- Even if a code has few faults, its models and equations could be inadequate or wrong. As theorist Robert Laughlin puts it, "One generally can't get the right answer with the wrong equations."
- It's also possible that the physical data used in the code are inaccurate or have inadequate resolution.
  (*garbage in – garbage out*)
- Or perhaps someone who uses the code doesn't know how to set up and run the problem properly or how to interpret the results.

31

31

Five common verification techniques

1. Comparing code results to a related problem with an exact answer
2. Establishing that the convergence rate of the truncation error with changing grid spacing is consistent with expectations
3. Comparing calculated with expected results for a problem specially manufactured to test the code
4. Monitoring conserved quantities and parameters, preservation of symmetry properties, and other easily predictable outcomes
5. Benchmarking—that is, comparing results with those from existing codes that can calculate similar problems.

32

32

Step 9:

Research

33

Calculations and analysis

- Plan ahead: what to calculate, always think before running your code
- Keep records
  - Date
  - Version
  - Changes to the code
- Graphics
- Analyze results
- … and be ready to revise you project from any step

34

34

Step 10:

Program maintenance

35

Real programs may live for years!

- Program may need to add new features (including more physics)
- Program may require improving performance
- Program may need to be shared with other groups and labs
- Large projects may have obscure bugs that were not detected during testing

Example: Department of Energy - Accelerated Strategic Computing Initiative "it takes about eight years for a staff of 15–30 to develop a massively parallel three-dimensional weapons simulation".

36

36