12.010 Computational Methods of Scientific Programming

Lecturers

Thomas A Herring
Chris Hill

Review of Lecture 2

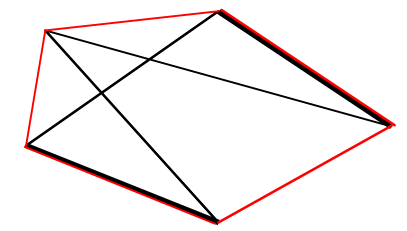
- Examined computer hardware
- Computer basics and the main features of programs
- Program design: Consider the case of calculating the area of an arbitrary n-sided polygon.
  - We had considered input of points that make up polynomial
- Finish this discussion and then start Fortran

Input options

- In some cases, for an arbitrary set of coordinates the figure is obvious, but in others it is not
- So how do we handle this?
- Force the user to input or read the values in the correct order?
- What if user makes a mistake and generates a figure with crossing lines?
- Warn user? Do nothing?

How do we define area of black figure?

Is red figure what we really meant?



Input options for Polygon

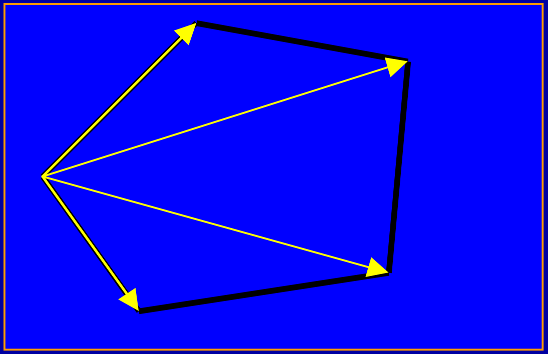
- By considering scenarios before hand we can make a robust program.
- Inside your program and in documentation you describe what you have decided to assume
- You may also leave “place-holders” for features that you may add later – not everything needs to be done in the first release.
- Final input option: Ask user number of points that will be entered? Or read list until End-of-file?
- Maximum number of points user can enter

Calculation of area

- Option 1: Area = base*height/2
  - Base compute by Pythagoras theorem; height some method
- Option 2: Cross product: form triangle by two vectors and magnitude of cross product is twice the area
- This sounds appealing since forming vectors is easy and forming magnitude of cross product is easy
- If we decide to go with option 2, then this sets how we will form the triangles.

Forming triangles

- If we use option 2 for the area calculation then we will form the triangles from vectors.



- To form the triangles, we form the yellow vectors which we do by sequencing around the nodes of the figure. Not the only method.

Output

- Mainly we need to report the area. What units will say the area is in? Need to ask user at being what the coordinate units are (e.g., m, km etc).
- Should we tell the user if lines cross?
- Anything we have forgotten?
- So we have now designed the basic algorithms we will use, now we need to implement.

Write program in English

Think of this operation as a Recipe

- Start: Get coordinates of nodes of polygon from user, also get units of coordinates — still debating how much checking we will do?
- Since we will sum areas of each triangle, set the initial value to 0.
- Loop over nodes starting from the third one (first node will be the apex of all triangles, the second node will form the first side of the first triangle) — need to check that 3 of more nodes have been entered!
  - Form the vectors between the two sides of the triangle (vector from apex to current node and previous node)
  - Cross the vectors to get the the area increment (only Z-component needed, so that we will not need to implement a full cross product)
  - Sum the area increment into the total sum.
- Output the results (and maybe a summary of the nodes)
- Done!

Design parts in more detail

- Usually at this stage you think of things you had not considered before.
- Useful to select variable and module names:
- Variables:
  - Max_nodes -- Maximum number of nodes needed
  - Num_nodes -- Number of nodes input by user (sometimes better to get program to compute this rather than have user specify it: they may make a mistake).

[Thought: for large numbers of nodes maybe it is better to compute area as nodes are entered? This will have impact on what we can output at the end of the program (or we could output as we go?]

  - Nodes_xy(2,max_nodes) -- coordinates of nodes saved as double indexed array (how these are specified is language dependent)

Variable/module names 2

- Triangle_vec(2,2) -- Two vectors that make up current triangle, first index is for X,Y; second index for sides 1 and 2.
- area, darea -- Total area and incremental area for current triangle.

- Modules needed
  - read_nodes -- reads the nodes of the polygon
  - Form_triangle -- Forms the triangle vectors from the node coordinate
  - triangle_darea -- computes increment in area (uses modified cross product formula)
  - Output_area -- Outputs the area of the triangle.

Implement

- With design in hand, variables and modules defined the code can be written.
- Usually, small additions and changes occur during the code writing, especially if it is well documented.
- Specifically: See poly_area.f code. While implementing code, it was realized that if nodes are not entered in a consistent direction, the sign of the triangle area changes and this can be detected and the user warned. (version 2 modification).
- Once code is running: Time to verify.

Verification

- Once code is implemented and running verification can be done a number of ways.
- Each module can be checked separately to ensure that it works. Especially check the error treatment routines to make sure that they work.
- One then tests with selected examples where the results are known.

Examine the program poly_area.f

- Implementation of algorithm described above.
- Take note of documentation
- Checks on human input
- This program will be looked in more detail when we cover Fortran.

Common problems

- Numerical problems. Specifically
  - Adding large and small numbers
  - Mixed type computations (i.e., integers, 4-byte and 8-byte floating point)
  - Division by zero. Generate not-a-number (Inf) on many machines
  - Square root of a negative number (Not-a-Number, NaN)
  - Values which should sum to zero but can be slightly negative.

Common problems 02

- Trigonometric functions computed low gradient points [e.g., $\cos^{-1}(\sim 1)$, $\tan(\sim\pi)$]
- Quadrants of trigonometric functions
  - For angle return, best to compute sin and cosine independently and use two-argument $\tan^{-1}$.
- Wrong units on functions (e.g., degrees instead of radians)
- Exceeding the bounds of memory and arrays.
- Infinite loops (waiting for an input that will never come or miscoding the exit)
- Unexpected input that is not checked.

Reading other’s code

- Often you will asked to existing software and add features to it
- You should apply the techniques described in normal program development to “reverse engineer” the program
- Specifically look for failure modes that you might fall into as you modify code (see check in form_triangles routine.

Start of Fortran

- Start examining the FORTRAN language
- Development of the language
- "Philosophy" of language: Why is FORTRAN still used (other than you can't teach an old dog new tricks)
- Basic structure of its commands
- Communications inside a program and with users
- Next lecture will go into commands in more detail
- There are many books on Fortran and an on-line reference manual at: http://www.fortran.com/fortran/F77_std/rjcnf0001.html

FORTRAN (Formula Translation).

- History
  - Developed between 1954-1957 at IBM
  - FORTRAN II released in 1958
  - FORTRAN IV released in 1962 (standard for next 15 years)
  - FORTRAN 66 ANSI standard (basically FORTRAN IV).
  - FORTRAN 77 standard in 1977
  - Fortran 90 Added new features, in 1997 Fortran 95 released (started phasing out some FORTRAN 77 features).

FORTRAN Philosophy

- FORTRAN developed a time when computers needed to do numerical calculations.
- Its design is based on the idea of having modules (subroutines and functions) that do calculations with variables (that contain numeric values) and to return the results of those calculations.
- FORTRAN programs are a series of modules that do calculations with typically the results of one module passed to the next.
- Usually programs need some type of IO to get values to computations with and to return the results.

FORTRAN 77:

- Commands are divided into executable and non-executable ones. All non-executable commands must appear before the executable ones.
- Syntax is quite rigid (discuss later)
- Basic Structure:
  - Module types:
    - Program (only one per program, optional)
    - Subroutine — Multi-argument return
    - Function —single return (although not forced)
    - Block data —discuss later

Basic Structure 02

- Variable types
  - Integer*n where n is number of bytes (2/4)
  - Real*n where n is 4 or 8
  - Complex*n where n is 4 or 8
  - Character*(m) where m is number of characters in string variable
  - Logical*n is n is 2 or 4
  - Byte (equivalent to integer*1)
- In this course, all variables must be explicitly declared. In Fortran all variable types must be declared implicitly or explicitly before executable statements.

Basic Structure 03

- Constants: Numerical, strings or defined in parameter statement
- I/O
  - Read (various forms of this command)
  - Write (again various forms)
  - Print (useful for debug output)
  - Command line arguments (discuss more later)
  - Format defines how results are read and written.
- Math symbols: * / + - ** (power) = (assignment). Operations in parentheses are executed first, then **. * and / have equal precedence (left to right), + - equal precedence left to right.

Basic Structure 04

- Control
    - If statement (various forms)
    - Do statement (looping control, various forms)
    - Goto (you will not use in this course)
- Termination
    - End (appears at the end of every module)
    - Return (usually at end of modules, except program)
- Communication between modules
    - Variables passed in module calls. Two forms:
        - Pass by address (memory address of variable is passed)
        - Pass by value (actual value passed, rarer and usually only applies when constants or expressions to be evaluated are passed)

Communication

- Communications between modules
  - Return from functions
  - Common blocks (specially assigned variables that are available to all modules)
  - Save (ensures modules remember values)
  - Data presets values before execution (during compilation)
  - Parameter (method for setting constants).

Other types of commands

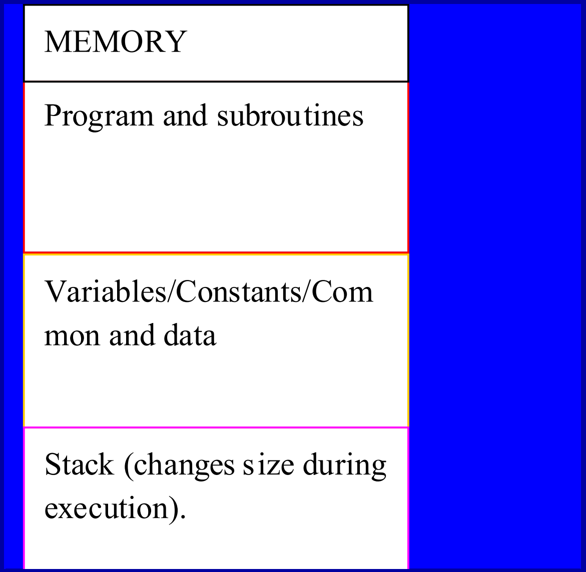
- Other types of commands
  - Open (opens a device for IO)
  - Close (closes a device for IO)
  - Inquire (checks the status of a device)
  - Backspace rewind change position in device, usually a file).
  - External (discuss later)
  - Include (includes a file in source code)
  - Implicit (we will use in only one form.

Syntax

- Relatively rigid (based on punched cards)
  - Commands are in columns 7-72 (option exists for 132 columns but not universal).
  - Labels (numerical only) columns 1-5
  - Column 6 is used for “continuation” symbol for lines longer than 72 characters.
  - Case is ignored in program compilation (but strings are case sensitive i.e., a does not equal A)
  - Spaces are ignored during compilation (can cause strange messages)

Program Layout

- Actual layout of program in memory depends on machine (reason why some “buggy” program will run on one machines but not others).
- Typically executable layout in memory.


MEMORY
Program and subroutines
Variables/Constants/Common and data
Stack (changes size during execution).

- Not all machines use a Stack which is a place memory is temporarily allocated for module variables.
- Good practice to assume stack will be used and that memory is “dirty”

Compiling and linking

- Source code is created in a text editor.
- To compile and link:

  f77 <options> prog.f subr.f libraries.a -o prog

Where prog.f is main program plus maybe functions and subroutines

  subr.f are more subroutines and functions

  libraries.a are indexed libraries of subroutines and functions (see ranlib)

  prog is name of executable program to run.

- <options> depend on specific machine (see man f77 or f77 -help)

Basic f77 options

- Options differ greatly between different machines although there are some common ones (these are not universal)
  - -c compile only do not link
  - -u assume implicit none in all routines
  - -ON where N is level of optimization. Optimization can lead to significant speed increases but on complex codes can generate strange errors.
  - -g compile for debugging
- Typically many more options often to provide for use of old codes (e.g., -onetrip). We will not explore these but useful to check if trying to get someone else's code running.

Basic layout and command details

- A basic Fortran program looks like (see poly_area.f for example).

```
      program name
*       Comments
        Non-executable declarations
           ……
        executable statements

        end
        subroutine sub1
*       Comments
        Non-executable declarations
           ……
        executable statements
        return
        end
```

Character of commands

- Modules are invoked by call for subroutines and assignment statements for functions.
- Certain system level modules are invoked just through their names. For example
  - OPEN   Opens a files (takes arguments)
  - CLOSE   Closes a file
  - READ and WRITE are of this type
- User modules or routines (these are the building blocks) are of types:
  - SUBROUTINE
  - FUNCTION

Summary

- Look at the basic characteristics of FORTRAN
- Next lecture we look in more details of the syntax of the commands (e.g., the arguments that can be used with an open statement)
- Trial FORTRAN programs will also be shown to examine the actual structure of a program.