12.010 Computational Methods of Scientific Programming 2021

Tom Herring, tah@mit.edu

Chris Hill, cnh@mit.edu

Lecture 5: Input/Output, Program design

Summary

- Input/Output: Simply first, more later
- Programming approach: Algorithm development
  - Example of how to think about the process: Speed of coding versus speed of execution
- Example with Lec05_polyarea.ipynb
- Verification of code. Expecting the unexpected.
- Common problems that can occur.

Simple terminal input output*

- Notebook for this section is Lec05-keyboardIO.ipynb
- Keyboard access in Python uses the **input** function
- This function returns a string that can be cast if a single value (use int or float function)
- For list or array input, other processing is needed.
- The **map** method can be used as an iterator to cast each value in the string (see Notebook for example)
- These methods will be applied in an example program development: The area of a polygon

Problem solving

- The clearer you can state the problem you are solving, the algorithms to use, and the possible problem areas, the easier your programming will be.
- Rough rule: 90% of time should be spent on design, only 10% on actually writing code and getting it to work.
- A good program is like good literature (it should logically flow)
- All your programs should be written in English before you start.
- It is tempting to develop the code as you type with notebooks, but it's best to plan it beforehand and think about algorithms.
- With Python, web searches to see what modules are available and possible approaches are useful too.

Program structure

- Basically, all programs can be broken into three major parts:
  - Input: The program collects the information it needs
  - Computation: it does the necessary evaluations to solve the problem
  - Output: Output its results for the user
- In an interactive program, these parts may be looped over.

Language features

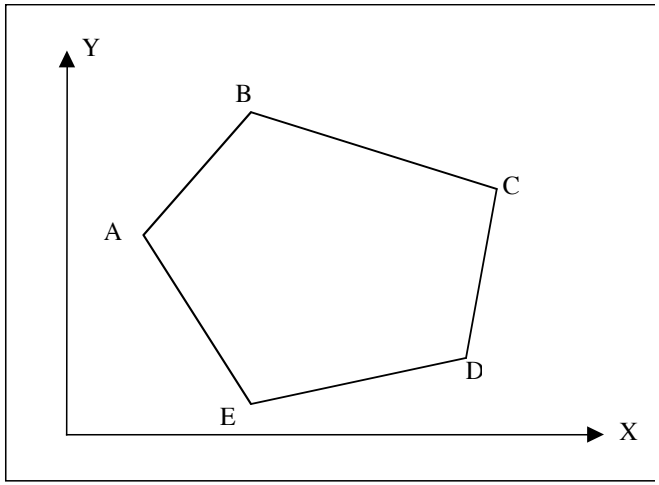
- All languages have the following basic features:
  - Start and end features
  - Input/output commands from and to a variety of sources
  - Decision structure (i.e., conditional branching and looping structures, error checking)
  - Assignment (setting variables to values, computing results. Be cautious here; binding of objects is common in Python. For numpy arrays use np.copy to make a copy, = assignment with bind the variable – the **x is z** case)
  - Module structure that allows separation of functions.

Features

- A program is made up of the appropriate combinations of these features.
- The specifics of the features vary between languages, and some have more features than others.
- The syntax is different between all the languages, although there are enough similarities to make it confusing
- So, while learning the syntax, you should keep careful note of how commands are structured in each language.

Specific problem example

- **Problem:** Find the area of an arbitrarily shaped plane figure.
- Figure defined by X, Y coordinates of vertices



How do we start solving problem?

- First: Ask questions, and lots of them
  - What basic algorithm should be used?
  - Numerical integration by discretizing the shape? (i.e.. Make a fine grid over the shape and sum the area of grid elements inside the shape)
    - This approach has a problem in that the accuracy will be limited by element size in the integration. Runtime will depend on the size of the grid.
  - Break figure into triangles?
    - Sounds OK. Can be made arbitrarily accurate
  - Look for an analytic solution in a numerical algorithms book. Hint: Look at Green's Theorem, which relates an area integral of the curl of a function to the line integral of the function (see: http://mathworld.wolfram.com/GreensTheorem.html

Sample problem

- Let's say we decide that breaking the figure into triangles is the algorithm to be used.
- So what will we need to do this:
  (a) How do we get the information about the shape into the program?
  (b) A way of computing the area of a triangle
  (c) A way of forming triangles from the coordinates.
  (d) how do we report the result?
- Algorithms (and routines) to compute areas of polygons can be found on the web (search "area of polygon"). In programming, we look at how to build these modules into a complete system that includes the algorithm, IO, and logic needed.

Input options

(a.1) How do we get the information into the program?

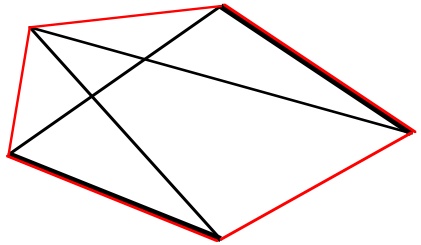
(a.2) Consider possible cases:

(a.3) Input can not be completely arbitrary (although in some cases it can be)

Input options

- In some cases, for an arbitrary set of coordinates, the figure is obvious, but in other cases it is not.
- So how do we handle this?
- Force the user to input or read the values in the correct order.
- What if the 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

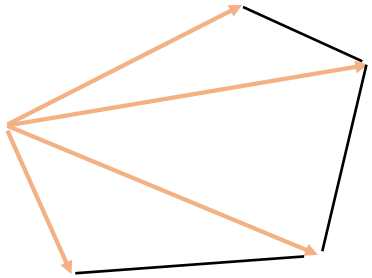
- By considering scenarios beforehand, we can make a robust program.
- Inside your program and in documentation, you describe what you have decided to assume
- You may also leave “place-holders” for features that you may add later – not everything needs to be done in the first release.
- Final input option: Ask the user the number of points that will be entered? Or read a list until the end of the list is specified by some means?
- The maximum number of points a user can enter could be an issue for some languages (not Python, except for memory size).
- If we plan to plot the polygon, do we make the user close the polygon, or do we do it?

Calculation of area

- Option 1: Area = base*height/2
  - Base compute by Pythagoras theorem; height some method
- Option 2: Cross product: form a triangle by two vectors, and the magnitude of the cross product is twice the area
- This sounds appealing since forming vectors is easy and forming magnitude of a 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 brown vectors, which we do by sequencing around the nodes of the figure. Not the only method.

Output

- Mainly, we need to report the area. Consider units and possibly scale factors if we get coordinates from an image.
- Should we tell the user if lines cross? How do you detect this?
- Make a plot of the polygon (may require saving all the node information)
- Is there anything we have forgotten?
- So we have now designed the basic algorithms we will use, and now we need to implement them.

Write program in English

- Think of this operation as a Recipe
- Start: Get coordinates of nodes of the polygon from the user. Possibly get units of coordinates — still debating how much checking we will do?
- Since we will sum the areas of each triangle, set the initial value to 0.
- Loop over nodes starting from the third one (the first node will be the apex of all triangles, and the second node will form the first side of the first triangle) — need to check that three or 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 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.
- Plot and 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:
  - Numnode – Number of nodes input by the user (sometimes it is better to get the program to compute this rather than have the 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 an impact on what we can output at the end of the program (or we could output as we go?]
  - nodes_xy[2,:] -- coordinates of nodes saved as double indexed array (how these are specified is language dependent)

Variable/module names

- trivec – Array with the vectors that form each triangle
- area, darea -- Total area and incremental area for the current triangle.

- Functions/Modules needed
  - Readnodes – read the nodes of the polygon
  - Form_vectors – Forms the triangle vectors from the node coordinate
  - Triarea – computes the area (uses modified cross-product formula)
  - Plotpoly – Plots the polygon
- Notebook or Python script calls these functions

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 polyarea.ipynb code.
- Once the code is running, **time to verify**.

Verification

- Once code is implemented and running, verification can be done in 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 polyarea.ipynb*

- Implementation of the algorithm described above.
- Take note of the documentation
- Checks on human input

- How might we expand this notebook? Allow an image to be read (e.g., a satellite image of growing crops and select the points in the polygon graphically.

Common program problem with algorithm development

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

Common problems

- Trigonometric functions computed low gradient points [e.g., $\cos^{-1}(\sim 1)$, $\tan(\sim \pi)$]
- Quadrants of trigonometric functions
  - For angle return, it is best to compute sin and cosine independently and use two-argument $\tan^{-1}$ function (arctan2).
- Wrong units on functions (e.g., degrees instead of radians)
- Exceeding the bounds of memory and arrays.
- Being confused by separate copies of objects versus bound versions.
- Infinite loops (waiting for an input that will never come or miscoding the exit)
- Unexpected input that is not checked.

Summary

- Input/Output: Simply first, more later
- Programming approach: Algorithm development
  - Example of how to think about the process: Speed of coding versus speed of execution
- Example with Lec05_polyarea.ipynb
- Verification of code. Expecting the unexpected.
- Common problems that can occur.

MIT OpenCourseWare

https://ocw.mit.edu

12.010 Computational Methods of Scientific Programming, Fall 2024