Lec09 IAM550 J. Raeder 9/23/2019 Problem Solving, Program Design

Problem solving and programming requires some structured approach:

1. ***Problem analysis:*** Analyze and understand the problem.
2. ***Problem Statement:*** Build a mathematical model of the problem to be solved. That is sometimes straight-forward. At the end of this step you should have all the equations that need to be solved. Note: in lab/homework problems these two steps are often already worked out.
3. ***Processing Scheme:*** define the inputs and outputs. Inputs can be anything from parameters to huge data sets. The output is likely a processed data set, plots, or sometimes just a single number.
4. ***Design the Algorithm:*** Break the solution process down into steps. Follow a *top-down* process that leads to a *structure plan*. For simple problems that is just a sequential list, but for more complex problems it may look more tree-like: to solve A I need to solve B, but that requires to solve B1 and B2, which in turn require to solve B1a, B1b, B1c, and B2a, for example. The structure plan can be just plain English or *pseudo-code*. It should be easily translated into computer code. For more complex code you also need to find out which sub tasks you can use existing code (previously written yourself, from the web, libraries, toolboxes). You should not re-invent the wheel.
5. ***Program the Algorithm.*** Write it piece-by-piece. Outer loops first. Never write a program in one piece. As soon as you have a piece of code that accomplishes some subtask, or yet incomplete task*, test it!* If the code contains functions, write the functions one-by-one and test each before using them in the main code. Note that there are several types of error: Simple semantic errors (MATLAB does not understand and gives you a red error message), wrong code (either the algorithm was bad to begin with, or you mistyped something) that executes fine but provides wrong answers, runtime errors (i.e., division by zero), or numerical errors because of truncation or unstable numerical methods.
6. ***Evaluation and Testing.*** Once the whole code is written it should be verified. That can take several different approaches, like running the code for simple cases where the answers are known, reverse calculation (for example a matrix decomposition), comparing with experiments, comparing with results from other codes.
7. ***Application:*** Solve problems that the code is supposed to solve.

Example 1: loan payoff.

1. Given a loan amount, interest rate, and pay schedule visualize how the loan gets paid off, how long it takes, and how much interest is ultimately paid.
2. Mathematical problem: $Bn = Bo + Bo*I/100/N - payoff$; where Bo is the current balance, Bn the new balance, I is the yearly interest rate, N is the number of payments per year, and payoff is the loan payment.
3. Input: B0 (initial balance), I (interest rate), N (number of yearly payments). Output: time to pay off loan, a table of payments and balances, a graph showing balance as a function of time.
4. English: Make an infinite loop over time, calculate next balance, adjust payment so balance becomes zero and exit loop once balance is below zero. Print table while looping. Store time, balance, interest payment in array for later plotting. Make plots after loop.
   Pseudo-code:
   Infinite loop:
   Calculate next balance
   Print next line of table
   If balance < 0:
   Calculate last payment
   Store interest
   Print last line of table
   Exit loop
   Store time, interest, balance
   End loop
   Plot
5. Code it up.
6. Sanity tests: zero interest, more interest than payment.
7. Run code. Check out effect of doubling payment, shorter intervals (biweekly).