10. 780.20 Session 10

a. Follow-ups to Session 9

- **Profiling in Mac OS X.** It's unfortunately the case that gprof, which is the program we use to do profiling in Linux and Cygwin, is broken in some versions of Mac OS X. If someone knows of an easy guide to a good replacement, please let me know.
- **Comparing Two Files.** The `diff` command is a very convenient way to quickly determine the difference between two files. For example, you have saved the original version of a program and someone gives you a modified version. To find the changes,

  `diff` *from-file to-file*

  will list just the lines that are different, with `<` in front of the line as it appears in *from-file* and `>` in front of the line as it appears in *to-file*, with the relevant line numbers given. To find out more, use `man diff` (there are many options).

b. Adaptive Differential Equation Solvers

In general, it's best to adjust the step size h in solving a differential equation because the optimal size will vary for different parts of the function. A routine that varies the step size automatically to keep the local error under control is called "adaptive". We'll try out the adaptive routines from GSL in this session with the `ode_test.cpp` code. This code is based on the example included with the GSL reference manual.

The test program will solve the Van der Pol oscillator, which is defined by the equation

$$\frac{d^2x}{dt^2} + \mu\frac{dx}{dt}(x^2 - 1) + x = 0 \ , \tag{10.1}$$

where μ is the only parameter. To specify a solution, we give initial values for x and $v = dx/dt$ (which we call x_0 and v_0, respectively). We'll take $\mu = 2$ with a variety of initial conditions.

Choosing different initial conditions means starting at different points in a phase space plot (v vs. x). In Session 10, you'll start at three different initial conditions. You should find that the phase-space trajectories all end up on the same curve. (Does this work for *any* initial conditions?) This is called an "isolated attractor" [1] (the phase-space trajectories are *attracted* to the universal curve).

This is just one example. You are invited to explore further!

c. Interpolation vs. Data Fitting

In this session, we will look at *interpolation*, which is sometimes confused with *data fitting*. Our basic interpolation problem will be to take a table of function values $\{x_i,\ y_i = f(x_i)\}$ for which an analytic or at least easily evaluated form is not available, and estimate $f(x)$ for $x \neq x_i$ (for interpolation, x should be *between* two of the x_i's, otherwise it is extrapolation, which is much harder to do with controlled errors). An important generalization is to interpolation in more than one dimension, so that we start with a table $\{\mathbf{x}_i,\, y_i = f(\mathbf{x}_i)\}$ and seek an estimate for $f(\mathbf{x})$, where $\mathbf{x}$ is a vector. Here are some possible applications for interpolation:

- we need to evaluate $f(x)$ for many x points but each evaluation is very expensive computationally;
- we want to calculate $\int_a^b [f(x)]^2\, dx$ using Gaussian quadrature;
- we want the derivative (or 2nd derivative) of the tabulated function f;
- we want to solve an ode involving f using a GSL routine.

There is an important assumption in all of these applications: the values of f should not be *noisy* (although they invariably have round-off errors). An example of a noisy function f would be experimental data. If you want to interpolate a noisy function, it's usually best to first fit a curve to the data and then to interpolate on the fit function. (We'll be looking at fitting next!)

If we assume the values y_i are not noisy, then between x_i and x_{i+1}, $f(x)$ should look like a polynomial, if the points are spaced closely enough. (How close is close enough? Try to answer this after going through this section.) But what polynomial should we use? The GSL provides several interchangable interpolation methods, which you'll compare for a simple application. The handout "Using GSL Interpolation Functions" takes you through the steps needed to use the GSL interpolation functions. Once we have an interpolated function, we can treat it as a continuous function in any of our codes to evaluate integrals, solve differential equations, and so on.

Lagrange interpolation fits an $(N-1)^{\rm th}$ degree polynomial to $f(x_i)$ given N values of x_i. That is, the polynomial is constructed to exactly pass through those N points; if we call this polynomial $P_N(x_i)$, then

$$P_N(x_i) = f(x_i) \equiv y_i \ , \qquad i = 0, 1, \cdots, N-1 \ . \tag{10.2}$$

That is, we have N pairs $\{x_i, y_i\}$ that we use to determine the polynomial function $P_N(x)$ (which is then applied for intermediate values of x). The formula for $N = 1$ (linear interpolation) is:

$$P_1(x) = \frac{x - x_0}{x_1 - x_0}\, y_1 + \frac{x - x_1}{x_0 - x_1}\, y_0 \ , \tag{10.3}$$

while for $N = 3$ (a parabolic approximation) we have:

$$P_3(x) = \frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)}\, y_2 + \frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)}\, y_1 + \frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)}\, y_0 \ . \tag{10.4}$$

The general formula is

$$P_N(x) = \sum_{i=0}^{N} \prod_{k\neq i} \frac{(x - x_k)}{(x_i - x_k)}\, y_i \ . \tag{10.5}$$

(Note that there is no requirement that the x_i points be equally spaced.) You might think that applying this to n points with $N = n$ would be optimal. It is not! Lagrange interpolation works best when the higher derivatives in $f(x)$ are small (or zero). Therefore, you should only apply polynomial interpolation to a relatively small region (you'll see what can happen over a larger region in the Session 10 example!). Added note: "barycentric Lagrange interpolation" is a fast and stable variant of classical Lagrange interpolation. See J.-P. Berrut and L. N. Trefethen, SIAM Review **46**, 501 (2004) for details.

A spline function is built from polynomial pieces defined on subintervals of the entire interval for the function. The most commonly used spline is the *cubic spline.* The idea in this case is to fit $f(x)$ in the interval $[x_i, x_{i+1}]$ with a cubic polynomial

$$f_i(x) = f_i + f_i^{(1)}(x - x_i) + \frac{1}{2} f_i^{(2)}(x - x_i)^2 + \frac{1}{6} f_i^{(3)}(x - x_i)^3 \ , \tag{10.6}$$

with the requirement that the function $f(x_i)$ is reproduced at all of the x_i and the first and second derivatives be continuous with the next interval. Thus the function and the first and second derivatives are continuous through the entire interval. We still need boundary conditions for $f^{(2)}$ at the endpoints. A "natural" spline chooses $f^{(2)}(a) = f^{(2)}(b) = 0$. The spline coefficients are determined by a GSL library routine. In the process, we get approximations to the first and second derivatives for free.

The example problem we will consider in Session 10 is a set of scattering cross section data at a range of energies, given in the following table:

E(MeV)	0	25	50	75	100	125	150	175	200
σ_{exp}(mb)	10.6	16.0	45.0	85.5	52.8	19.9	10.8	8.25	4.7
σ_{th}(mb)	9.34	17.9	41.5	83.5	51.5	21.5	10.8	6.29	4.09

In general you would expect experimental noise (e.g., statistical fluctuations) in the experimental cross section. We would be more likely to *fit* the data to a theoretical model (e.g., a least-squares fit). The theoretical cross sections in the table were generated with the Breit-Wigner function

$$\sigma_{\text{th}} = \frac{\sigma_0}{(E - E_r)^2 + \gamma^2/4} \tag{10.7}$$

with $\sigma_0 = 63900\,\text{mb}\cdot\text{MeV}^2$, $E_r = 78.0\,\text{MeV}$, and $\gamma = 55.0\,\text{MeV}$ determined by a fit. We'll use the theoretical data at the energies in the table to test interpolation routines at intermediate values of the energy against the exact fit function.

d. Classes for the GSL ODE Solver

In Session 10 we use the code `ode_test.cpp`, which applies an adaptive differential equation solver from the GSL numerical library to integrate the Van der Pol oscillator equation (which exhibits an isolated attractor). The code was taken more-or-less directly from an example in the GSL manual and did not use any C++ object-oriented features. Here we describe its transformation to `ode_test_class.cpp` and the creation of the `Ode` class and related classes. If you have suggestions for improvements (or a better way to implement it), please let me know!

- **Motivation and Strategy.** The purpose of the original `ode_test.cpp` code is simple: set up the Van der Pol differential equation as coupled first-order equations, pick a parameter and initial conditions, then integrate over a given time range in given steps and output the result at each step to a file.
  - As we've said in other cases, there is no good reason to have all of the GSL apparatus visible in the main program. It should be hidden, not only for clarity but so that it is easy to switch to another numerical library (or use our own code like the Runge-Kutta function).
  - Ideally, from the main program we would only need to set the initial conditions and the range we want to integrate, along with the accuracy required. This goal guided the final form of `ode_test.cpp`.
  - We also want to be able to use the code with minor modifications for other differential equations. So we don't want anything in the GSL part to be specific for this equation. This implies a separation of data and functions that leads us to introduce a derived class.
  - Our general strategy is to group the Gsl calls and structures of a given type into a separate class. So the differential equation parts (which begin with `gsl_odeiv_`) will become a class while the matrix parts (which begin with `gsl_matrix`) are in a different class.
- **Class Overview.** Again, there are no fixed rules for choosing classes but we can follow some guiding principles. Since we plan to use the GSL library, the classes should be compatible with the GSL function calls but also generic enough that we could use another library. Here we made the following choices:
  - A class for the differential equation solver, which we called `Ode`. This class should know what algorithm is used to integrate the equation (and have a choice of more than one). And it should have all of the apparatus to allocate and deallocate working objects and, of course, to do the actual stepping from one time to another.
  - A class for the differential equation itself. This means both the right-hand side functions and (if needed) the Jacobian functions. We called this class `Rhs`.
  - It is apparent that this will not work directly, because we want to be able to define a `Rhs` for each differential equation of interest, with different numbers of equations and parameters. We'd also like to avoid the GSL solution of using void pointers (at least in the part of the code visible to the user). Our solution is to define a "base class" `Rhs` that has common functionality and is particular to the differential equation solver implementation (in this case GSL) but to have the actual equation defined in a "derived class" `Rhs_VdP`. Every different equation would have a different derived class (with a unique name). The "VdP" here is to remind us it is the Van der Pol equation.
  - We could imagine having classes for other elements of the code, such as the output file. But we decided just to move those features into a "evolve-and-print" function and defer to future revisions a more modular approach to that part.
- **What are the details?** There are various implementation details.

– The constructor for the `Ode` class (`Ode::Ode`) needs as arguments the tolerances for the solver as well as the type of the solver. We decided to pass the latter as a string. Some types might not be available for a different implementation than GSL, but it would just be treated as an "Illegal ode type". A tower of `if` statements is used to convert the string to the corresponding GSL type, which is then implemented in the call to `gsl_odeiv_step_alloc`.

– The other argument to the `Ode` class is an `Rhs` class. In particular, it is an instance of the class derived from `Rhs`. In practice a `Rhs_VdP` class object would be passed, or a corresponding object for another differential equation. An important detail is that in the prototype for the `Ode` constructor, the `Rhs` class is passed as a *reference*. This is indicated by the `&` in `Rhs &this_rhs`. It means that the actual object is used (via an alias) rather than a copy of it. The class would fail if we didn't do this!

– The rest of the `Ode` constructor are the allocation and set-up calls to GSL that used to be in the main function of `ode_test.cpp`. The `Ode` destructor gets the deallocation GSL calls. Finally, there is one method, called `evolve`, which simply combines the internal `gsl_odeiv` structures with the start/finish times, step size, and current vector inputs and passes them on to the GSL evolve function.

– The public functions of `Rhs` include two versions of the rhs and jacobian functions. The first set, called `rhs` and `jacobian`, are declared here but never intended to be called from a direct instance of the `Rhs` class. Instead, we expect that an object from a derived class like `Rhs_VdP` is being used and will have its own version of these routines. To make sure that those versions are invoked, we declare the ones in `Rhs` to be "virtual". The second set, called `gsl_rhs` and `gsl_jacobian` just pass along to the GSL functions the results of calling `rhs` and `jacobian`. They are declared as "static" so that they can be passed as ordinary function pointers (as we've used from the beginning with GSL). The implementation of these functions takes advantage of their having the GSL form with an argument that is a void pointer. Recall that a void pointer can point at *anything*, so here we use it as a pointer to a `Rhs` class. What actually gets referenced is a `Rhs_VdP` object, which then gives us access to its data and methods. Note how it is deferenced to extract the number of equations and the `rhs` and `jacobian` functions.

– Notice that the `Rhs_VdP` constructor takes the argument `mu_passed` while the `Rhs` constructor has no arguments (and, in fact, does nothing!). There is no problem with this. It means we could pass any number of arguments to another `Rhs` derived class representing a different differential equation. Besides setting the private value of `mu` to `mu_passed`, the number of equations is also set. Two points to note here: the number of equations is a fixed property of this particular differential equation (two, since it is second order), so it should be set here and not passed (and not set in the base class). But the variable `num_eqs` needs to be a member of the base `Rhs` class (since we need to refer to it in general) and therefore not declared in `Rhs_VdP`. It must be "protected" rather than "private" so that `Rhs_VdP` can inherit it. (If we want to use private methods or data in a derived class, we declare them as protected, not private.)

– We could (and probably would in general) move the `Rhs_VdP` parts of `ode_test_class` into a header file (the class declaration at the top) and a class file (the function definitions at the bottom). But we chose to leave it in the main file as an example that classes *can* be in the same file as the main program.

– We also simply moved the bulk of the code that evolved from `tmin` to `tmax` and then printed out the results to the function `evolve_and_print`. Note that the `Ode` and `Rhs_Vdp` objects are passed as references, so that they are not copied. By putting all of this in a separate function, we can easily add additional integrations (e.g., with different initial conditions) without making multiple copies of these statements. It might be useful to go further and define a class for the output file, but at some point there are diminishing returns on the modularization.

- **Application to other GSL functions.** This was our first example of GSL-related classes that used user-defined functions (i.e., the `rhs` functions). It can serve as a prototype for similar implementations. We have done so with the minimization and least-squares fitting implementations from Session 11.

e. Python Scripts to Run C++ Programs

A Python "script" to run C++ programs is simply a Python program that interacts with the input or output of the C++ code. It is often just imitating a series of commands you might type instead to the command line. We will imagine that we want to run the C++ code multiple times with different inputs and possibly process the output (e.g., to create a file suitable for plotting).

We will use our simple `area.cpp` code from Session 1 as an example C++ code that we would like to control from a script. We will want to replace the interactive input with either command-line arguments or reading from an input file and in the latter case we'll also output to a file. We'll call these versions `area_cmdline` and `area_files`, respectively. Let's start with `area_cmdline`:

```
//*********************************************************************//
//  file: area_cmdline.cpp
//
//  This program calculates the area of a circle, given the radius,
//   with input from the command line.
//
//  Programmer:  Dick Furnstahl  furnstahl.1@osu.edu
//
//  Revision history:
//      28-Dec-2010  original version, from area.cpp
//
//  Notes:
//   * compile with:  "g++ -o area_cmdline area_cmdline.cpp"
//   * we need to convert argv[1] to a double using atof
//
//*********************************************************************//

// include files
```

```
#include <iostream> // this has the cout, cin definitions
#include <stdlib.h>      // this has atoi and atof
#include <cmath>         // this has atan
using namespace std;     // if omitted, then need std::cout, std::cin

//*********************************************************************//

const double pi = 4.*atan(1.);

int
main (int argc, char *argv[])   // use the standard, if obscure, names
{
  if (argc != 2)   // check if there is exactly one argument
  {
    // argv[0] is the program name
    cout << "usage: " << argv[0] << " <radius>" << endl;
    cout << "Try again!" << endl;
    exit (0);   // quit the program
  }

  double radius = atof(argv[1]);    // convert 1st argument to a double

  double area = pi * radius * radius; // area formula

  cout << "radius = " << radius << ",  area = " << area << endl;

  return 0; // "0" for successful completion
}

//*********************************************************************//
```

The command-line arguments are specified as arguments to the `main` function, namely an integer and then a pointer to an array of character strings. These variables are conventionally called `argc` and `argv`, but you could use different names. The value of `argc` is the total number of arguments plus one; this will be the number of entries in `argv` (the extra one is for the program name, which is always `argv[0]`). We can access the first argument from `argv[1]`, the next one (if there is one) from `argv[2]`, and so on. Even if we use numbers as command-line arguments, they are stored as character strings, so we need to convert them to appropriate data types in the program. The functions `atoi` and `atof` from the `stdlib.h` header file do the conversions to integers and doubles, respectively. This is how we get the radius in the example code.

Ok, now that we have a program that can take arguments, what kind of Python script can we use? The first example calls the program for a specified list of input values (which can be changed by the user in the `run_area_cmdline1.py` script, and simply prints all the results to the screen:

```
#!/usr/bin/env python
#
#  file: run_area_cmdline1.py
```

```
#
#  This python script runs the C++ code area_cmdline for a sequence
#   of values that are hardcoded in the script.  The output is to
#   the screen.
#
#  Programmer:  Dick Furnstahl  furnstahl.1@osu.edu
#
#  Revision history:
#      28-Dec-2010  original version
#
#  Notes:
#   * run with: "python run_area_cmdline1.py" or just "./run_area_cmdline1.py"
#   * should check the return code to make sure it worked!
#   * could use Popen instead of call
#   * pass a full string to the shell with the first method below
#      or pass a list of arguments without "shell=True".
#
#*************************************************************************
from subprocess import call    # make the "call" function available

# Define the inputs at the top, so they are easy to find and change
value_list1 = [1, 10, 100]
value_list2 = [1.0, 0.5, 0.25]

# Ok, let's do it . . .
print "\nFirst try a list of integers:"

for radius in value_list1:    # don't forget the colon!
  my_command = "./area_cmdline " + str(radius)  # convert radius to a string
  retcode = call(my_command, shell=True)    # pass "my_command" to be executed

print "\nNow try a list of decimal numbers:"

# note the alternative passing of the command
for radius in value_list2:    # don't forget the colon!
  my_command = ["./area_cmdline", str(radius)]  # convert radius to a string
  retcode = call(my_command)

print "The last return code was", retcode

#*************************************************************************
```

The Python script above and the ones below are designed to be rather generic, so you can adapt them without knowing many details of the Python language. Here are some relevant comments on the above script.

- A key feature you do need to remember is that (consistent) indentation matters. Blocks of statements in for loops or if blocks are not indicated by {}'s as in C++, but simply by being indented.

- No semicolons to end the lines. If you want a continuation line you need an explicit `\`.
- The lists of input radii are defined between `[]`'s, with commas between the entries.
- The `for` loop steps through values in the list. Note the colon at the end of the `for` statement, which indicates the start of the loop. All the indented lines are part of the loop and it ends when the indentation ends. (There is no required number of spaces for the indentation, it just has to be the same indendation for each line in a block.)
- The `call` function is used to execute the command, which is specified as a string that includes the program name and any command-line arguments, each converted to a string (that is what the `str` function is doing).
- The printed output here is dictated by the C++ code. (Try running it in Session 10.)
- The "return code" has a value that is 0 if the program executed successfully. We should really check for this!

Next we'll try defining the set of values to pass to the C++ program (in this case radii) with some functions defined in a Python library called `numpy`. This may not be included in the Python installation you are using. (If you are allowed to do so, you can simply install it.) If it is not there, the script will naturally fail with an error message.

```
#!/usr/bin/env python
#
#  file: run_area_cmdline2.py
#
#  This python script runs the C++ code area_cmdline for a sequence
#   of values that are hardcoded in the script.  The output is to
#   the screen.  Here the ranges are specified from numpy functions.
#
#  Programmer:  Dick Furnstahl  furnstahl.1@osu.edu
#
#  Revision history:
#      28-Dec-2010  original version from run_area_cmdline1.py
#
#  Notes:
#   * run with: "python run_area_cmdline2.py" or just "./run_area_cmdline2.py"
#   * should check the return code to make sure it worked!
#   * could use Popen instead of call
#   * pass a full string to the shell with the first method below
#      or pass a list of arguments without "shell=True".
#
#*********************************************************************
from subprocess import call
import numpy   # could use "from numpy import *" to avoid typing numpy

# Define the inputs at the top, so they are easy to find and change
value_list1 = numpy.linspace(1,2,num=11)   # num chosen so nicely spaced
value_list2 = numpy.arange(1.,0.,-0.25)
```

```
# Ok, let's do it . . .
print "\nUsing functions from numpy to specify radii."

print "\nFirst try a list of evenly spaced numbers from linspace:"
for radius in value_list1:    # don't forget the colon!
  my_command = "./area_cmdline " + str(radius)  # convert radius to a string
  call(my_command, shell=True)    # pass  "my_command" to be executed

print "\nNow try a list using arange:"
for radius in value_list2:    # don't forget the colon!
  my_command = ["./area_cmdline", str(radius)]  # convert radius to a string
  call(my_command)

#**********************************************************************
```

You can look up the `linspace` and `arange` functions online. They are patterned after the MATLAB `linspace` and `range` functions.

Finally, we include a sample script that processes the output from the C++ code. We won't describe it in detail (although the comments may be helpful) but you can find more information simply by Googling "python" and any of the commands like "Popen" and "findall". You should be able to reproduce most of the functionality without understanding all of the details.

```
#!/usr/bin/env python
#
#  file: run_area_cmdline3.py
#
#  This python script runs the C++ code area_cmdline for a sequence
#   of values that are hardcoded in the script.  The output is processed
#   to convert it to a nice table.
#
#  Programmer:  Dick Furnstahl  furnstahl.1@osu.edu
#
#  Revision history:
#      28-Dec-2010  original version based on other run_area scripts
#
#  Notes:
#   * run with: "python run_area_cmdline3.py" or just "./run_area_cmdline3.py"
#
#**********************************************************************
from subprocess import Popen,PIPE
from re import findall
from operator import itemgetter

# Define the inputs at the top, so they are easy to find and change
value_list1 = [1, 10, 100]
value_list2 = [1.0, 0.5, 0.25]

my_filename = "test.file"
```

```
# Ok, let's do it . . .
print "\nOutput is to", my_filename, ". . ."

tmp_file = open (my_filename,'w')
for radius in value_list1:    # don't forget the colon!
  my_command = "./area_cmdline " + str(radius)  # convert radius to a string
  my_pipe = Popen(my_command, shell=True, stdout=tmp_file)

for radius in value_list2:    # don't forget the colon!
  my_command = "./area_cmdline " + str(radius)  # convert radius to a string
  my_pipe = Popen(my_command, shell=True, stdout=tmp_file)

print "All done!"

tmp_file.close()   # all done, close the file

print "\nNow try it without files!"
my_output = []   # start with an empty list
for radius in (value_list1 + value_list2):
  my_command = "./area_cmdline " + str(radius)  # same command as before
  # We're sending no input to communicate and keeping the first element
  #  of the tuple that is returned.  Then we append it to the my_output string.
  my_output.append(Popen(my_command, shell=True, stdout=PIPE).communicate()[0])

# At this stage we have a list of all of the output in the string my_output
print " Here is what the output list looks like:"
print my_output, "\n"

# Step through the output and pull out the radius and area values
print " radius          area  "; print "-------------------------"
for entry in my_output:
  # pull out the two numbers in each entry (if no words, this would be easier!!)
  my_matches = findall("\d+\.*\d*",entry)  # 1+ digits, 0+ periods, 0+ digits
  radius = float(my_matches[0]); area = float(my_matches[1])

  # Here is the old way of formatting
  print ' %6.2f  %12.3f ' % (radius, area)
  # Here is the new way (for newer versions of Python)
  #  print ' {0:6.2f}  {1:12.3f}'.format(radius, area)

# Step through the output and gather all the radius and area values
radii = [];  areas = [];
for entry in my_output:
  # pull out the two numbers in each entry (if no words, this would be easier!!)
  my_matches = findall("\d+\.*\d*",entry)
  radii.append(float(my_matches[0])); areas.append(float(my_matches[1]))

print "\n radius          area  "; print "-------------------------"
```

```
for radius,area in zip(radii,areas):
  print ' {0:6.2f}  {1:12.3f}'.format(radius, area)

# Now try sorting with itemgetter
print "\n Now we sort by radius:"
print "\n radius          area  "; print "------------------------"
for radius,area in sorted(zip(radii,areas),key=itemgetter(0)):
  print ' {0:6.2f}  {1:12.3f}'.format(radius, area)

#*************************************************************************
```

Ok, let's move on to our other modification of `area.cpp`, which is `area_files.cpp`. The new feature here is input from a file that uses `getline` and `sscanf` rather than just `cin`. This is not necessary for a simple input file in which every line is the same. But here we also allow an arbitrary number of comment lines to illustrate how to handle more general input files.

```
//*********************************************************************//
//  file: area_files.cpp
//
//  This program calculates the area of a circle, given the radius.
//   This version reads in input from a file and outputs to another file.
//
//  Programmer:  Dick Furnstahl  furnstahl.1@osu.edu
//
//  Revision history:
//      28-Dec-2010  original version, from area.cpp
//
//  Notes:
//   * compile with:  "g++ -o area_files area_files.cpp"
//   * this version has hardwired filenames; we could also allow
//      optionally to pass them as command-line arguments
//
//*********************************************************************//

// include files
#include <iostream> // this has the cout, cin definitions
#include <iomanip>       // manipulators like setprecision
#include <fstream> // file input and output
#include <cmath>         // M_PI lives here
#include <string>        // strings and things
using namespace std;     // if omitted, then need std::cout, std::cin

//*********************************************************************//

const double pi = M_PI;   // define pi using the cmath pre-defined version
double area (double radius);   // separate out the area calculation

int
```

```
main (void)       // no command-line arguments here
{
  ifstream input_file ("area_files.inp", ios::in);  // open for input only
  if (input_file.bad())   // check if the file is there and can be read
  {
    cerr << "Unable to open area_files.inp. " << endl;
    exit(1);   // bail out
  }

  // open the output file so that old versions are overwritten ("trunc")
  ofstream output_file ("area_files.out", ios::out|ios::trunc);
  output_file << "#  radius      area   " << endl;

  // print radii to the output file
  string line;   // entire line from the input file
  while (getline(input_file,line))
  {
    if (line[0] == '#')   // skip line if the first character is "#"
    {
      cout << "comment line: " << line << endl;
    }
    else   // we assume this is a real line
    {
      double radius;
      sscanf(line.c_str(),"%lf",&radius);  // find a double (%lf) in line

      output_file << setw(8) << fixed << setprecision(2) << radius << "  "
                  << setw(12) << setprecision(3) << area(radius) << endl;
    }
  }

  input_file.close();
  output_file.close();

  return 0; // "0" for successful completion
}

//*********************************************************************//

double
area (double radius)
{
  return pi * radius * radius;  // area formula
}

//*********************************************************************//
```

Some comments:

- The `while` loop with the `getline` function reads in an entire line at a time from the input file until the end of the file. Each line is written to a string variable (which we cleverly named `line`).
- We look at the first element of the string, which is `line[0]`, to decide if it is a comment (using the same convention used by gnuplot and many other programs that a line starting with `#` is a comment).
- If we decide the line is a real line, we scan it for a radius in the form of a double using the string scanning function `sscanf`:
  ```
    sscanf(line.c_str(),"%lf",&radius);
  ```
  The `c_str()` method converts the C++ string to a C-style character string as required by `sscanf`. The `%lf` in quotes indicates a double ("long float"). The last argument has to be the memory address of the variable, so we put `&` in front of `radius`.

It is not hard to generalize to more inputs on each line.

Our final Python script is an example of how to run the `area_files` C++ program by creating the input file and then executing the code. Most of it is the same as in the Python scripts above or is internally documented, so we will not explain it further. Have fun!

```
#!/usr/bin/env python
#
#  file: run_area_files2.py
#
#  This python script runs the C++ code area_files for a sequence
#   of values that are hardcoded in the script.  The output is to
#   the screen.  Here the ranges are specified from numpy functions.
#
#  Programmer:  Dick Furnstahl  furnstahl.1@osu.edu
#
#  Revision history:
#      28-Dec-2010  original version from run_area_cmdline2.py
#
#  Notes:
#   * run with: "python run_area_files2.py" or just "./run_area_files2.py"
#   * should check the return code to make sure it worked!
#   * could use Popen instead of call
#   * pass a full string to the shell with the first method below
#      or pass a list of arguments without "shell=True".
#
#*************************************************************************
from subprocess import call
import numpy   # could use "from numpy import *" to avoid typing numpy
from os import rename

# Define the inputs at the top, so they are easy to find and change
```

```
value_list1 = numpy.linspace(1,2,num=11)   # num chosen so nicely spaced
value_list2 = numpy.arange(1.,0.,-0.25)

# Ok, let's do it . . .
print "\nUsing functions from numpy to specify radii in an input file."

# First we specify the file name and then open it in the next line
input_filename = "area_files.inp"
input_file = open (input_filename,'w')  # we'll refer to the file as input_file

# Define strings in quotes for comment lines. We use \n to get new lines.
cmt_line1 = "# This is an input file for the area_files.cpp program\n"
cmt_line2 = "#  After five lines with #'s, any number of radii are given\n"
cmt_line3 = "#  one to a line, with any alignment and formatting.\n"
cmt_line4 = "#\n"

# Write the comment lines to the file (referred to by input_file)
input_file.write(cmt_line1)
input_file.write(cmt_line2)
input_file.write(cmt_line3)
input_file.write(cmt_line4)

hdr_line1 = "#   radius\n"    # a header line
input_file.write(hdr_line1)

for radius in value_list1:    # step through radii; don't forget the colon!
   input_file.write(str(radius)+'\n')  # "+" concatenates the strings

for radius in value_list2:    # step through radii; don't forget the colon!
   input_file.write(str(radius)+'\n')

input_file.close()    # all done, close the input file

my_command = "./area_files "   # the C++ executable
print "\n Now running area_files ..."
call(my_command, shell=True)    # pass  "my_command" to be executed

rename("area_files.out","area_files.out2")

#**************************************************************************
```

f. References

[1] R.H. Landau and M.J. Paez, *Computational Physics: Problem Solving with Computers* (Wiley-Interscience, 1997). [See the 6810 info webpage for details on a new version.]

[2] M. Hjorth-Jensen, *Computational Physics* (2013). These are notes from a course offered at the University of Oslo. See the 6810 webpage for links to excerpts.

[3] W. Press *et al.*, *Numerical Recipes in C++*, 3rd ed. (Cambridge, 2007). Chapters from the 2nd edition are available online from `http://www.nrbook.com/a/`. There are also Fortran versions.