Computational Physics

is the use of computer algorithms to model physical systems.

Typical CP tasks:

- Solving equations: e.g. simulation, root-finding
  - Symbolically
  - Numerical Approximation
- Optimization
- Matrix manipulation

Technically, CP falls under “theoretical” physics or “applied math” because it involves interacting with a model rather than the physical world directly. However, there are many aspects of CP which resemble experimental science, especially in the simulation of nonlinear systems – one can conduct “virtual experiments.”

CP is typically used for solving ‘complicated’ problems, e.g. nonlinear equations, large matrices, partial differential equations.

Unix Basics

The majority of CP is done on UNIX systems, so a working knowledge of UNIX operation is necessary. While many GUI (graphical user interface) and windows-like interfaces exist for UNIX systems (e.g., on Mac OS X), the use of the *command line* or *terminal* is essential. On UNIX this is known as the “shell,” and various flavors of shell are available, e.g. sh, csh, bash, tcsh. For this class, we’ll use tcsh.

For remote login, we’ll use the “secure shell” ssh, via

```
ssh <username>@<computer hostname or IP address>
```

which will put you on the remote machine and drop you into a shell.

Filesystem

Like other systems you may have worked on, programs and data are stored as “files”, organized within folders known as “directories.” The filesystem is *case-sensitive*, except on many Macs. The current directory is known as “.”, and the parent directory is “..”. Thus “ls .” shows all the files & directories in a current directory. “ls ..” shows all files & directories in the parent directory.

Basic Shell Commands

- **ls** “list”. Show files in a directory
  - **ls –l** “long list”, shows detailed info
  - **ls –a** “list all”, shows hidden files too
- **man** Shows the “Manual Page” (help page) for a given program.
  - **“man ls”** tells about the ls command & additional options
  - **“man man”** tells about man

    - **"man woman"** reveals nothing!
- **cp** copy. "cp <file> <newfile>"
- **mv** move. Like cp, but removes original
- **rm** remove. Deletes a file.
- **mkdir** make directory. Creates a "folder" in the filesystem
- **cd** change directory. "Go into" a directory
- **rmdir** remove (empty) directory
- **cat** shows contents of a file
- **more** like cat, but with page breaks

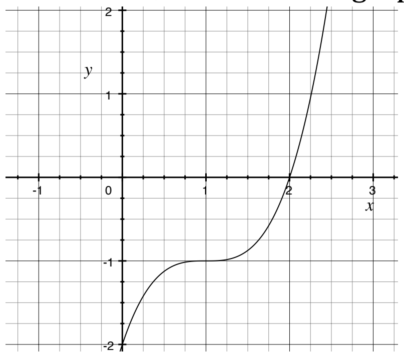
Running Programs

Just type in the name of the program. It must be regarded as "executable" by the filesystem in order to run. Often a "./" before the program name is necessary if the program is in the current directory, as in "./myprogram". This is for security reasons: it is inadvisable to have "." as part of the PATH. (More on PATH later…)

Root-Finding

Intro

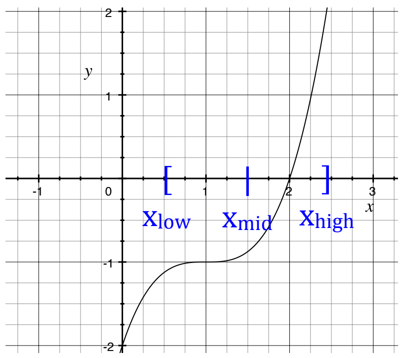
Consider the function $f(x) = x^3 - 3x^2 + 3x - 2$, shown in the graph below:



We call the value of *x* for which *f(x)* =0 the "root" or "solution" to the equation, and denote this value by *x**. One can see by inspection of the equation that the root in this case is at *x* = 2, so we say *x** = 2. Let's pretend we don't know this, as we will later want to find roots of non-linear equations in which the root is not easily obtained by inspection. In fact this will often require *numerical* root-finding algorithm to *approximate* the solution to some desired accuracy or *tolerance* δ.

Binary Search

One method of root finding is to put a "bracket" around the solution, using an upper bound x_{high} which you "know" is greater than the solution *x**, and a lower bound x_{low} which is known to be less than *x**.



The next step is to compute the mid-point $x_{\rm mid} = (x_{\rm high} + x_{\rm low})/2$, and evaluate $f(x_{\rm mid})$. If $f(x_{\rm mid}) < 0$, then we "move the upper bound in", by setting $x_{\rm high} = x_{\rm mid}$; however if $f(x_{\rm mid}) > 0$, then we set $x_{\rm low} = x_{\rm mid}$.

After this we compute a new $x_{\rm mid}$, and move the bounds of the bracket as above. This process repeats, with $f(x_{\rm mid})$ getting progressively closer to zero, until it deviates from zero by less than the desired tolerance, *i.e.,* until $|f(x_{\rm mid})| < \delta$.

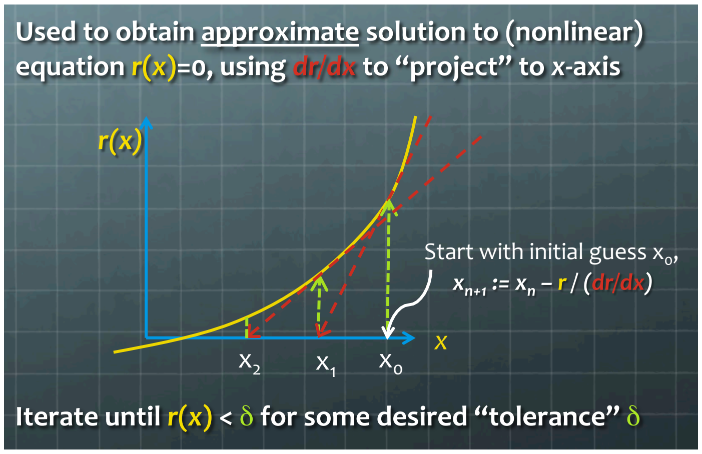
This process is certain to find a value which closely approximates x^*, if a root exists within the initial bracket. If *multiple* roots exist within the bracket, then it will produce one of them, but it may not be obvious at the beginning which root that will be. So proper bracketing of the solution is a requirement for this method to be used effectively.

The algorithm itself can be regarded as "slow" compared to other approaches which converge more quickly. Let's see how it performs on our original equation for an initial bracket extending from x = -10 to +10. Below is the output of a script which gives columns for the iteration number n and other relevant values, for $\delta = 0.01$:

```
n     x_low        x_mid        x_high       f(x_mid)                   |x* - x|
---------------------------------------------------------------------------------------------------------
1     -10          0            10           -2                         2
2     0            5            10           63                         3
3     0            2.5          5            2.375                      0.5
4     0            1.25         2.5          -0.984375                  0.75
5     1.25         1.875        2.5          -0.330078125               0.125
6     1.875        2.1875       2.5          0.674560546875             0.1875
7     1.875        2.03125      2.1875       0.096710205078125          0.03125
8     1.875        1.953125     2.03125      -0.134136199951172         0.046875
9     1.953125     1.9921875    2.03125      -0.0232548713684082        0.0078125
```

One can observe that the solution error $|x^* - x|$ is going down by a factor of 4 every other iteration, for an average convergence of 2^{-n}, that is an average decrease by a factor of 2 per iteration. If we continue for two more iterations, we find a solution error of `0.001953125`.

Newton's Method



Used to obtain approximate solution to (nonlinear) equation $r(x)=0$, using dr/dx to "project" to x-axis

Start with initial guess x_0, $x_{n+1} := x_n - r / (dr/dx)$

Iterate until $r(x) < \delta$ for some desired "tolerance" δ

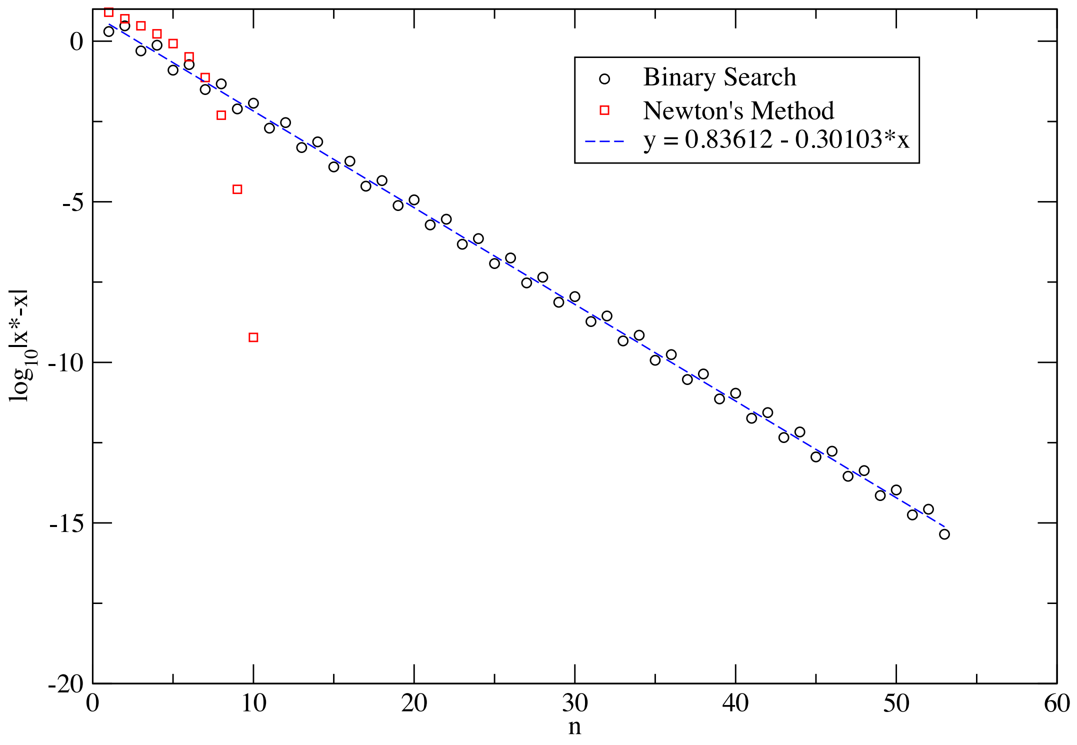
...converges much faster. Here's the output for the same function as above, starting with x_0 of 10:

```
n      x                    f(x)                     |x* - x|
------------------------------------------------------------------------------------------------
1      10                   728 8
2      7.00411522633745     215.444749345717         5.00411522633745
3      5.01199005522363     63.5772495115326         3.01199005522363
4      3.69536903307254     18.5818943654043         1.69536903307254
5      2.84279468192733     5.25793215979049         0.842794681927328
6      2.32668759444275     1.33510279948754         0.326687594442753
7      2.07384157815964     0.238285097487909        0.0738415781596395
8      2.00496125174223     0.014957719399586        0.00496125174223083
9      2.00002445220083     7.33583962357898e-05     2.44522008308046e-05
10     2.00000000059789     1.79367276587072e-09     5.97891069986645e-10
11     2                    0                        0
```

where "0" means "machine precision", typically around 1e-16 for many computers.
Notice the "precision doubling" which begins around $n = 7$, once enough iterations place us in the "convergence regime". This is where the error (a small number) is approximately *squared* with each iteration!

For comparison, recall that for the binary search the error only decreased by some constant factor (on average, 2), and after 11 iterations it had only reached a precision of about 0.002. To get to a tolerance of 3e-15, we would need to continue the binary search to over *fifty iterations.*

Here's a graph comparing the two methods, each with $\delta = 3e\text{-}15$:



(Note that the final iteration of Newton's method cannot be displayed on this graph.)

Newton's Method can fail "spectacularly" if it hits a region where the df/dx goes to zero, in which case the "linear projection" will send the "next point" out to infinity. Thus it is wise to start the initial guess such that df(x)/dx in the region between x_0 and x^* is always positive or always negative.

<u>Plotting in Python</u>

Many programming languages are used in computational physics. One of the currently most popular languages, especially for rapid prototyping, is Python. Python enjoys a large user base with many scientific packages available such as `numpy` and `scipy`.

Often we'll want to look at graphs of our work, and the `matplotlib` module does just want we want. Consider the following example in http://hedges.belmont.edu/~shawley/PHY4410/code/...

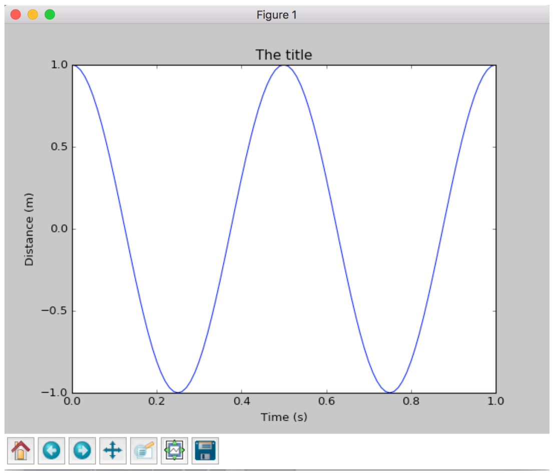
```
% cat simple_plot.py
#!/usr/bin/env python

import numpy
import matplotlib.pyplot as plt

x = numpy.arange(0.0, 1.0+0.01, 0.01)
y = numpy.cos(2*2*numpy.pi*x)

fig = plt.figure()
axes = fig.add_subplot(111)
#axes.set_xlim(0,1)     # if you don't set limits, it will autoscale
#axes.set_ylim(-1,1)
axes.set_title("The title")
axes.set_xlabel("Time (s)")
axes.set_ylabel("Distance (m)")

axes.plot(x,y)
plt.show()
```

We can run this either via

```
% python simple_plot.py
```

or, if the script is executable (obtained via running "`chmod u+x simple.plot.py`"), by simply typing

```
% ./simple_plot.py
```



Next: Animations…

Next: Numerical Integration…