Physics 750: Exercise 2

Tuesday, August 29, 2017

1. Use the `curl` command to download from the class website everything you'll need for this exercise.

```
$ WEBPATH=http://www.phy.olemiss.edu/~kbeach/
$ curl $WEBPATH/courses/fall2017/phys750/src/exercise2.tgz -O
$ tar xzf exercise2.tgz
$ cd exercise2
```

   The file ending in `.tgz` (or sometimes `.tar.gz`) is an archived (with `tar`) and compressed (with `gzip`) directory of files, sometimes called a *tarball*.

2. The `exercise2` directory contains a short C++ program named `curves.cpp` that instructs the computer to output x and $x^2 + 2x + 3$ in two columns (of width 20 characters) with x ranging from -10 to 10 in increments of 0.2. The `make` command invokes the compiler, which creates an executable named `curves`.

```
$ ls
curves.cpp     makefile
$ make
g++ -o curves curves.cpp -O -ansi -pedantic -Wall
$ ls
curves     curves.cpp     makefile
```

   The program outputs directly to the terminal. There are likely more lines of output from `curves` than there are lines in your terminal window. You can scroll through the output by *piping* it to the `more` command.

```
$ ./curves
                 -10                  83
                -9.8               79.44
                -9.6               75.96
                -9.4               72.56
                   .                   .
                   .                   .
                   .                   .
                -0.4                2.36
                -0.2                2.64
        -2.05391e-15                   3
                 0.2                3.44
                 0.4                3.96
                   .                   .
                   .                   .
                   .                   .
                 9.4              110.16
                 9.6              114.36
                 9.8              118.64
                  10                 123
$ ./curves | more
```

   Notice that we don't get exactly $x = 0$. Why might that be?

3. Use BASH's redirection operator (`>`) to save the `curves` output to a file named `poly.dat`. Use `tail` to inspect the last five lines of the file. The data can be viewed graphically in a program called `gnuplot`.

```
$ ./curves > poly.dat
$ tail -n5 poly.dat
                9.2             106.04
                9.4             110.16
                9.6             114.36
                9.8             118.64
                 10                123
$ gnuplot
> plot "poly.dat"
> plot "poly.dat" with lines
> plot "poly.dat", x**2 + 2*x + 3
> quit
```

The `plot` command will display both data from a file and arithmetic expressions in `x`. The `with` directive sets the style for the plot. You can see all the possible styles by typing `help with` at the `gnuplot` prompt. Note that `gnuplot` has an exponentiation operator (`**`), whereas C++ does not.

4. Open the `curves.cpp` file in your favourite text editor (`emacs`, `gedit`, `vim`, `pico`) and modify it so that the program now outputs three columns of data containing x, $p(x)$, and $q(x)$, where $p(x) = x^2 + 2x + 3$ and $q(x) = -\frac{1}{8}x^4 - \frac{1}{2}x^3 + x + 2$. (Warning: the expressions `1/8` and `1.0/8` do not evaluate to the same number!) You will have to write a new `polynom` function to handle the fourth-order polynomial. While you're at it, change the range to $-15 \leq x \leq 15$ and the increment to 0.1. Check that you can *exactly* reproduce the session below.

```
$ make
g++ -o curves curves.cpp -O -ansi -pedantic -Wall
$ ./curves > poly.dat
$ head -n3 poly.dat
                -15                198           -4653.62
              -14.9             195.21           -4519.98
              -14.8             192.44           -4389.22
$ tail -n3 poly.dat
               14.8             251.64           -7601.41
               14.9             254.81           -7798.13
                 15                258           -7998.62
$ gnuplot
> plot "poly.dat"
> plot "poly.dat", x**2 + 2*x + 3
> plot "poly.dat" using 1:2, x**2 + 2*x + 3
> plot "poly.dat" using 1:3, -0.125*x**4 - 0.5*x**3 + x + 2
> plot "poly.dat" using 1:2 with line, "poly.dat" using 1:3 with lines
> plot[-5:5] "poly.dat" using 1:2 with line, "poly.dat" using 1:3 with lines
> plot[-4:1][0:10] "poly.dat" using 1:2 with line, "poly.dat" using 1:3 with lines
> plot[-3:1] "poly.dat" using 1:(8*$3-2*$2), -x**4-4*x**3-2*x**2+4*x+10
> quit
```

The range is set explicitly by including `[`x_{left}`:`x_{right}`][`y_{bottom}`:`y_{top}`]`. In the final `plot` command shown above, the `using 1:(8*$3-2*$2)` directive says to plot column 1 on the x-axis and 8 times column 3 minus 2 times column 2 on the y-axis. This linear combination corresponds to the polynomial

$$\begin{aligned} 8q(x) - 2p(x) &= 8 \times (-\tfrac{1}{8}x^4 - \tfrac{1}{2}x^3 + x + 2) - 2 \times (x^2 + 2x + 3) \\ &= -x^4 - 4x^3 + 8x + 16 - 2x^2 - 4x - 6 \\ &= -x^4 - 4x^3 - 2x^2 + 4x + 10. \end{aligned}$$

5. By default `gnuplot` displays its output in an X11 window. You can also specify that the output be sent to a file instead (in any one of many possible graphics formats). For example, the following session creates an *Encapsulated PostScript* plot.

```
$ gnuplot
> plot[-3:1] "poly.dat" using 1:(8*$3-2*$2), -x**4-4*x**3-2*x**2+4*x+10
> set terminal postscript eps
> set output 'poly.eps'
> replot
> quit
$ evince poly.eps
```

6. `gnuplot` can fit numerical data to a given analytical form. In the following session, a general fourth-order polynomial `f(x)` is defined with five free parameters. Those free parameters are passed to the `fit` command using `via`.

```
$ gnuplot
> plot[-3:1] "poly.dat" using 1:(8*$3-2*$2)
> f(x) = a*x**4 + b*x**3 + c*x**2 + d*x + e
> fit[-3:1] f(x) "poly.dat" using 1:(8*$3-2*$2) via a,b,c,d,e
> plot[-3:1] "poly.dat" using 1:(8*$3-2*$2), f(x)
> quit
```

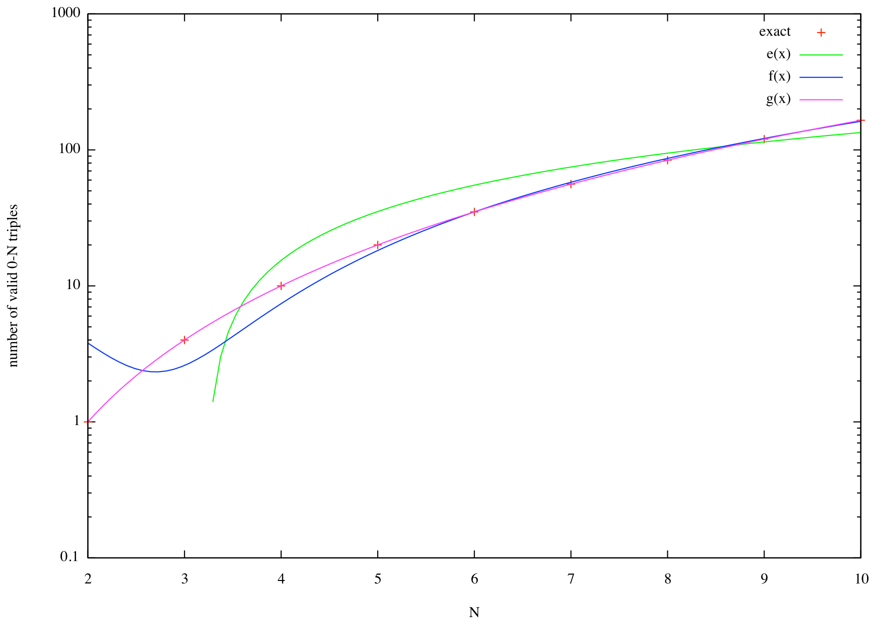
Find the best fit to a general third-order polynomial over the range `[-3:1]`. The fit should be quite bad. Now redo the fit for a fifth-order polynomial. What is the value of the x^5 coefficient? Why isn't it exactly zero?

7. The program `triples` outputs all strictly ordered triples of the integers 0 to 6 (inclusive).

```
$ make triples
g++ -o triples triples.cpp -O -ansi -pedantic -Wall
$ ./triples
(0,1,2)
(0,1,3)
(0,2,3)
(1,2,3)
.
.
.
(1,5,6)
(2,5,6)
(3,5,6)
(4,5,6)
```

Modify the program so that instead it *counts* the number of such triples. Generalize to integers $0, 1, \ldots, N$, and output the results in two-column format for all values of N from 2 to 10. You should be able to reproduce the following.

```
$ make triples
g++ -o triples triples.cpp -O -ansi -pedantic -Wall
 $ ./triples
     2           1
     3           4
     4          10
     5          20
     6          35
     7          56
     8          84
     9         120
    10         165
$ ./triples > trip.dat
$ gnuplot -persist view7.gp
```

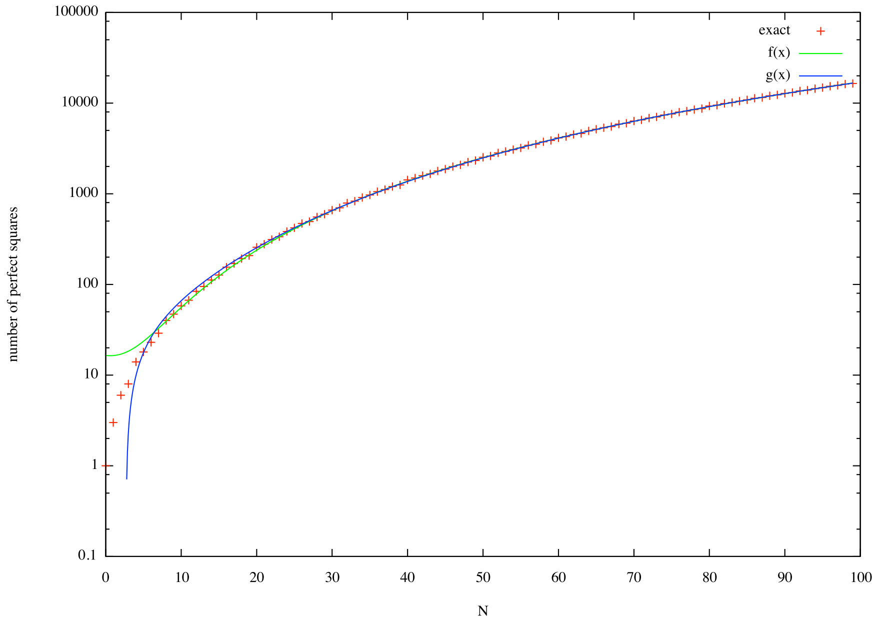


8. Using your solution to `triples.cpp` as a template, create a program file `quadruples.cpp` that writes three columns of data: N from 0 to 99, the number of *weakly* ordered quadruples (i, j, k, l) formed from the integers 0 through N, and how many of the sums $i^2 + j^2 + k^2 + l^2$ are perfect squares.

```
$ make quadruples
g++ -o quadruples quadruples.cpp -O -ansi -pedantic -Wall
$ ./quadruples
     0           1           1
     1           5           3
     2          15           6
     3          35           8
     4          70          14
     5         126          18
     .           .           .
     .           .           .
     .           .           .
    97     4082925       15689
    98     4249575       16213
    99     4421275       16462
```

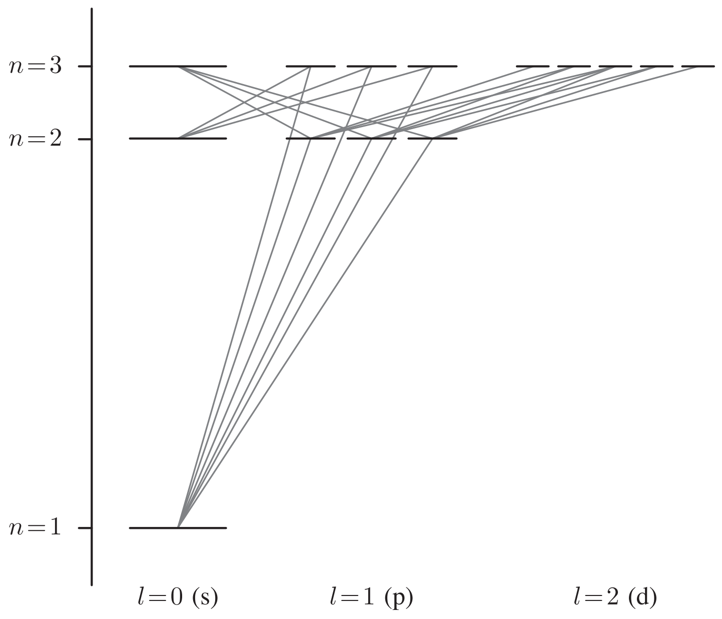
Determine whether an integer is a perfect square by whatever means. You may want to use `std::sqrt` from the `cmath` library in combination with integer casting.

Following `view7.gp`, compose a gnuplot script file `view8.gp` that plots the number of perfect squares alongside the 3rd- and 4th-order polynomial fits to the data.



9. The hydrogen atom consists of a single electron orbiting a positively charged nucleus. The electron can exist only in discrete orbitals characterized by the radial quantum number $n = 1, 2, 3, \ldots$ and the angular momentum quantum numbers $l = 0, 1, 2, \ldots, n-1$ and $m = -l, \ldots, -1, 0, 1, \ldots, l$. The energy of each orbital, $E_n = -(13.6\,\text{eV})/n^2$, is a function of the radial quantum number alone. Hence, each energy level is g_n-fold degenerate, where $g_n = \sum_{l=0}^{n-1}(2l+1) = n^2$. (In other words, there are $g_n = n^2$ states having the same energy E_n.)

   Transitions between orbitals can occur if the electron absorbs or emits a photon. But since a quantum of light has intrinsic angular momentum (1 in units of $\hbar$), conservation laws put a strict limit on which atomic transitions are possible. This leads to the famous electric dipole *selection rules*: $\Delta l = \pm 1$ and $\Delta m = 0, \pm 1$. Allowed transitions between the various states with $n = 1, 2, 3$ are shown in the following diagram.



   Complete the program `selection.cpp` so that it computes all possible transitions of the form $n_2 \to n_1$

with $n_1 < n_2 \le 20$. Have the program output the results in four columns indicating the initial and final radial quantum numbers, the number of allowed pathways, the energy of the emitted photon $\Delta E = E_2 - E_1$, and its wavelength $\lambda = hc/\Delta E$ in nanometers. (Recall that $hc = 1240\,\text{eV}\cdot\text{nm}$.)

```
$ make selection
g++ -o selection selection.cpp -O2 -ansi -pedantic -Wall -lm
$ ./selection | head -n5
 2->1         3          10.2         121.569
 3->1         3       12.0889         102.574
 3->2        15       1.88889         656.471
 4->1         3         12.75         97.2549
 4->2        15          2.55         486.275
```

10. Modify `selection.cpp` so that the spectral lines in the visible spectrum, $380\,\text{nm} < \lambda < 750\,\text{nm}$, are marked with a lowercase v.

```
$ make selection
g++ -o selection selection.cpp -O2 -ansi -pedantic -Wall -lm
$ ./selection | head -n5
 2->1          3          10.2         121.569
 3->1          3       12.0889         102.574
 3->2v        15       1.88889         656.471
 4->1          3         12.75         97.2549
 4->2v        15          2.55         486.275
$ ./selection | grep v
 3->2v        15       1.88889         656.471
 4->2v        15          2.55         486.275
 5->2v        15         2.856         434.174
 6->2v        15       3.02222         410.294
 7->2v        15       3.12245         397.124
 8->2v        15        3.1875          389.02
 9->2v        15        3.2321         383.652
```

11. Quantum particles have a property called *spin*, which is an intrinsic angular moment. The spin of a particle is restricted to be a multiple of $\hbar/2$. In units where $\hbar = 1$, the spin is either an integer or an odd multiple of one half. Electrons have spin $s = 1/2$.

The total spin S of a collection of electrons is determined by the angular momentum summation rule $\frac{1}{2} \otimes S = (S - \frac{1}{2}) \oplus (S + \frac{1}{2})$. The possible spin sectors for two, three, and four electrons are shown below.

$$\frac{1}{2} \otimes \frac{1}{2} = 0 \oplus 1$$
$$\frac{1}{2} \otimes \frac{1}{2} \otimes \frac{1}{2} = \frac{1}{2} \oplus \frac{1}{2} \oplus \frac{3}{2}$$
$$\frac{1}{2} \otimes \frac{1}{2} \otimes \frac{1}{2} \otimes \frac{1}{2} = 0 \oplus 0 \oplus 1 \oplus 1 \oplus 1 \oplus 2$$
$$\vdots$$
$$\underbrace{\frac{1}{2} \otimes \frac{1}{2} \otimes \frac{1}{2} \otimes \cdots \otimes \frac{1}{2}}_{2N \text{ times}} = \prod_{2S=0}^{N} \underbrace{S \oplus S \oplus \cdots \oplus S}_{C_S^{(N)} \text{ times}}$$

The final line shows the general result for N electrons. The coefficient $C_S^{(N)}$ denotes the number of states in a given spin sector. One can show that the total number of states in the singlet ($S = 0$) sector is

$$C_0^{(N)} = \frac{1}{N/2 + 1}\binom{N}{N/2} = \frac{N!}{(N/2)!(N/2+1)!}$$

and that the total number of states—counting the $(2S + 1)$-fold degeracy—is

$$\sum_{2S=0}^{N} (2S + 1) C_S^{(N)} = 2^N.$$

The 2^N result is just the number of ways to orient N electrons either spin-up or spin-down.

Read over the file `moments.cpp`. It provides the skeleton of a program that computes the coefficients $C_S^{(N)}$ and displays them in a table. Determine how elements of the array `current` (indexed by the integer $2S$) should be incremented in terms of the values in `last`. Accumulate the total number of states into the variable `num_states`. The output of your program should be identical to the following.

```
$ make moments
g++ -o moments moments.cpp -O2 -ansi -pedantic -Wall -lm
$ ./moments
          0    0.5      1    1.5      2    2.5      3    num
   +--------------------------------------------+-------
  1|             1                              |      2
  2|      1             1                       |      4
  3|             2             1                |      8
  4|      2             3             1         |     16
  5|             5             4             1  |     32
  6|      5             9             5         1|     64
  7|            14            14             6  |    128
  8|     14            28            20         7|    256
  9|            42            48            27  |    512
 10|     42            90            75        35|   1024
 11|           132           165           110  |   2048
 12|    132           297           275       154|   4096
 13|           429           572           429  |   8192
 14|    429          1001          1001       637|  16384
 15|          1430          2002          1638  |  32768
 16|   1430          3432          3640      2548|  65536
   +--------------------------------------------+-------
```

12. Write a function `verify_singlet` that computes $C_0^{(N)}$ *explicitly*. Check its result against `last[0]` for each even value of `n` from `2` to `Nmax`.