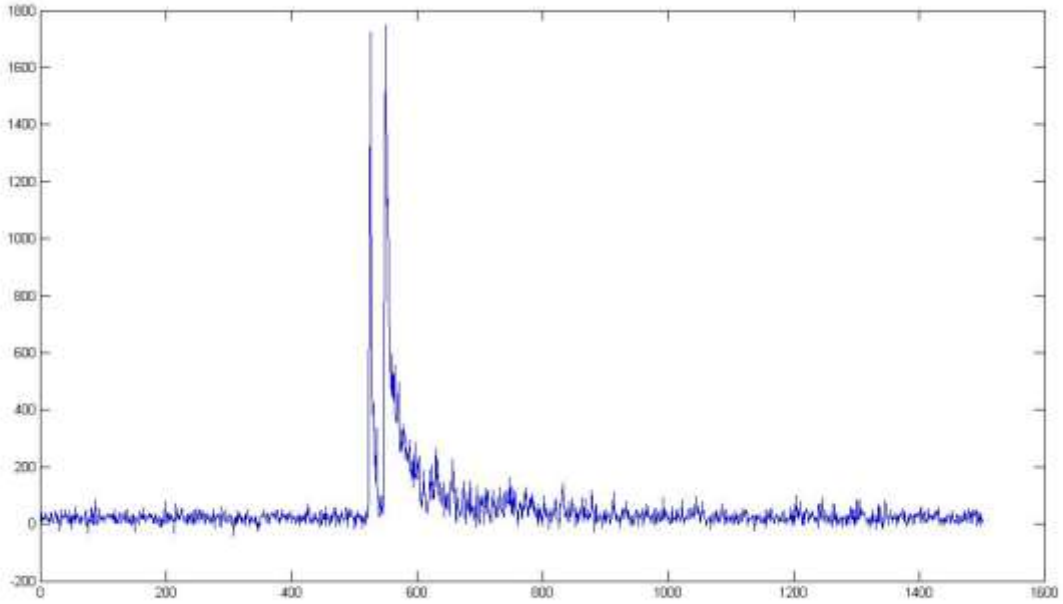
CS 3370

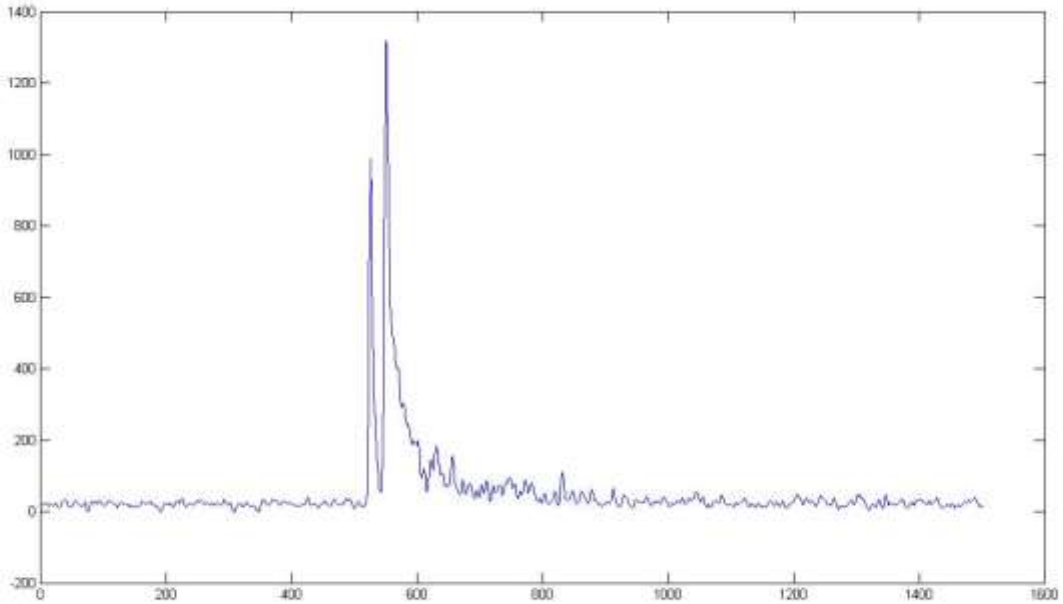
Program 4 – Data Reduction with C++ Algorithms

In this program you will analyze digitized data for pulses. The data comes from an actual digitizer that reports voltages (except all the data has been reversed by the hardware – you must negate all values before proceeding). Here is a graph of one of the sample data files (which consist of integers separated by whitespace) after negating the values:



The data is quite jagged, so for finding pulses we will use a smooth version instead. In a new vector, copy the first 3 numbers from the original, negated data. Then, starting with the 4th point of the file (position [3], of course), and ending with the 4th from the last, replace each of those points with the following weighted average of its neighbors (the current point in question is *pointed to* by "iter"):

```
(iter[-3] + 2*iter[-2] + 3*iter[-1] + 3*iter[0] + 3*iter[1] + 2*iter[2] + iter[3]) / 15
```



Then copy the last 3 numbers over to fill out the smoothed vector. Here's what the smoothed data looks like:

The smooth data will be better for detecting pulses (you can see two noticeable ones in the pictures above).

You detect a pulse by looking for a rise over three consecutive points. If the rise ($y_{i+2} - y_i$) exceeds `vt`, (for "voltage threshold" – supplied by an input parameter), then a pulse begins at position *i*. After finding a pulse, move forward through the data starting at y_{i+2} until the samples start to decrease before looking for the next pulse.

Write a program that process all files with a ".dat" extension in the current directory. There can be an arbitrary number of such files, and an arbitrary number of sample data values within each file. Print out where pulses are found and the "area" underneath the pulses. The area is merely the sum of the values starting at the pulse start and going for `width` samples (another input parameter), or until the start of the next pulse, whichever comes first. Use the original, unsmoothed data to compute the area, however. (The smooth data is just for detecting pulses.)

There is one "gotcha". Sometimes it is hard to distinguish "piggyback" pulses (where a pulse is *adjacent* to another) from a wide pulse with some variation to it. To distinguish these, we use the following parameters:

Parameter	Description
`drop_ratio`	A number less than 1
`below_drop_ratio`	The number of values less than `drop_ratio`
`pulse_delta`	The gap between pulses to look for piggybacks

After you have found where pulses begin, check to see if the start of a pulse is followed by another pulse that starts within (<=) `pulse_delta` positions of it. If this occurs, find how many points between the *peak* of the first pulse and the *start* of the second pulse (non-inclusive; only look at point strictly inside this interval) are strictly less than `drop_ratio` times the height of the peak of the first pulse. If the number exceeds `below_drop_ratio`, omit the first pulse from further consideration (it is not a pulse of interest).

Read all parameters from an .ini file like the following at program startup (specified in argv/argc; don't hard-code them):

```
# Pulse parameters
vt=100
width=100
pulse_delta=15
drop_ratio=0.75
below_drop_ratio=4
```

All parameters are necessary and no others are allowed. *Enforce this*. Lines beginning with '#' are comments so ignore them. Blank lines are allowed and should also be ignored.

If a .dat file has no pulses, ignore it. For those that do, report the pulse start positions and their areas in the following format (you should get these same answers):

```
as_ch01-0537xx_Record1042.dat:
Found piggyback at 1020
Found piggyback at 1035
1050 (8228)

Invalid file: test.dat

2_Record2308.dat:
1019 (3540)
1030 (22916)

2_Record3388.dat:
1020 (43272)
1035 (41264)
```

The second .dat file has pulses of interest at (zero-based) positions 1019 and 1030, with respective areas 3540 and 22916. The order of appearance of the .dat files is not important.

Implementation Notes

First get the program working, and then consider using lambda expressions and algorithms such as **copy**, **transform**, and **accumulate**, along with **istream_iterator**s to avoid some loops and make the code shorter and more readable. How are you going to discover all the .dat files in the current directory? Use the **std::filesystem**. Also, verify that the .ini file is present and valid. Here are some sample runs for the invalid .ini files:

```
Charless-MacBook-Pro:prog-analyze chuck$ ./a.out error1.ini
Invalid value for vt in error1.ini
Charless-MacBook-Pro:prog-analyze chuck$ ./a.out error2.ini
Invalid INI file: missing one or more keys
Charless-MacBook-Pro:prog-analyze chuck$ ./a.out error3.ini
Invalid or duplicate INI key: foo
```

Assessment Rubric

Competency ↓	**Emerging →**	**Proficient →**	**Exemplary**
File I/O			
Clean Code		No repeated code (refactor); No unnecessary code. Use **std::filesystem**.	Simplest possible logic to fulfill program requirements; use *algorithms* and *iterator adaptors* to avoid needless loops (I only have 4 loops) to make code more readable; use *lambdas* and/or standard function objects to avoid writing needless function bodies; use **std**::**accumulate** to compute the area.
Defensive Programming	Exceptions thrown as requested		
Other		Proper use of command-line arguments as assigned.	Reasonable error checking when processing .ini file.