The One Billion Row Challenge is the initiative of the software developer and blogger Gunnar Morling (https://www.morling.dev/blog/one-billion-row-challenge/). Initially it was a challenge for the Java community to develop a fast parser for big files with one billion (or more) records. This challenge is a good opportunity to stretch one's brains, solving this problem in different languages.

Although 1BRC was the inspiration, this project deviates from the canonical task formulated by Gunar. I was more interested to see how fast Python and C++ code could perform solving quadratic and cubic equations. In addition, it was interesting to see how fast some OHLCV data processing can be when these data are coming from the file.

The formula for solving a quadratic equation, defined by three parameters: a, b, c is well known. Herea are examples of such equations:

Parameters			**Roots**	
a	**b**	**c**	**r1**	**r2**
1.00	-5.00	6.00	3.0	2.0
1	-4	4	2	
-36.26	-28.4	-198.01	None	None

The situation with a cubic equation is more complicated. There are 4 parameters: a, b, c, d, and much less simplicity about how to solve it. I used the approach based on the Rational Root Theorem:
https://en.wikipedia.org/wiki/Rational_root_theorem

Herea are examples:

Parameters				**Roots**		
a,	**b,**	**c,**	**d,**	**r1,**	**r2,**	**r3**
2.0,	-9.0,	13.0,	-6.0,	1.0,	2.0,	1.5
1.0,	-6.0,	11.0,	-6.0,	1.0	2.0	3.0
1.0,	-4.0	7.0,	0.0	-4.0	1.0	
1.0,	0.0,	-2.0	0.0	4.0,	-2.0	
1.0,	2.0	3.0	4.0	None, None, None		

The data for both types of equations can be generated with help of *make-many-records.py* scripts which produces the file data_100_000_000.csv, containing 100m parameters. All these numbers are random, generated by the call to *numpy.random uniform(-10000, 10000, (1000000, 3)).*

In the beginning, testing results would be another challenge, so there is a second script that makes life a bit easier creating a file populated with repeating records: *make-many-same-records.py.* The difference in their results is that same-records look like this:

```
a,        b,  c
1.0, -5.0, 6.0
1.0, -5.0, 6.0
2.0, -4.0, -6.00
2.0, -4.0, -6.00
3.00, 2.0, -8.00
3.00, 2.0, -8.00
1.0, -5.0, 6.0
...
```

These a, b, and c are making solvable equations, so we always can check whether the code is doing the job or just warming up the office:

```
a,b,c,r1,r2
1.0, -5.0, 6.0, 3.0, 2.0
2.0, -4.0, -6.0,3.0, -1.0
3.0, 2.0, -8.0, -2.0, 1.333
```

And there are convenience scripts for the cubic equation:
make-many-similar-cubic.py

It works in the same manner, creates a CSV file randomly populated with the following:

```
a,b,c,d
2,-9,13,-6
1,-6,11,-6
1,2,-1,-2
1,-4, 7, -4
```

For all experiments I decided I will limit the amount of data to 100,000,000 equations. It less than 1 billion but still is a big enough…

I did not bother to polish the code of both auxiliary scripts, so they are not taking command line arguments, not using logger etc. They just make data files.
The scripts that solve equations are different stories. I did my best to be pythonic and civic. There are two sections, one for Python, one for C++. C++ development is more time-consuming thus so far, I managed only one solution. Python results are more numerous.

The Python script *machine-info.py* tells what kind of computer was bestowed with the privilege to run these experiments. I was able to use the following machinery:

- HP Windows 11 (HPW)
- Ubuntu with x86_64, 8 cores (Ubuntu)

Detailed machine characteristics are put together into the Machines table below. At some point I am going to test these solutions using some more hardware.

Machines

HP Server System Information:	Ubuntu System Information
Windows	**Linux**
Release: 11	Release: 6.8.0-52-generic
Version: 10.0.26100	Version: #53~22.04.1-Ubuntu SMP
Machine: AMD64	Machine: x86_64
Processor: Intel64 Family 6 Model 151, Genuine_Intel	Processor: x86_64
Physical cores: 16	Physical cores: 8
Max Frequency: 3200.00Mhz	Max Frequency: 4825.00Mhz
Min Frequency: 0.00Mhz	Min Frequency: 800.00Mhz
Current Frequency: 3200.00Mhz	Current Frequency: 1384.17Mhz
Memory Information	**Memory Information**
Total: 127.69GB	Total: 31.12GB
Available: 95.83GB	Available: 25.61GB
Used: 31.86GB	Used: 4.41GB
Percentage: 25.0%	Percentage: 17.7%
Disk Information	**Disk Information**
Device: C:\	Device: /dev/nvme0n1p2
File system type: NTFS	File system type: ext4
Total Size: 3.64TB	Total Size: 937.33GB

Python Solutions of Quadratic Equations

I have 3 Python scripts:

- simple – a very straightforward approach: read the file and run the formula. I used a generator to read the data from the CSV file: an equation is read, passed to the solver, results are saved into the output.

```
QE> pypy mr-qe-simple.py

👍 :     100000007 records, 85714329 quadratic equations, 3 without roots

👍 :     Total time=221.923 sec

😀
```

  And there is the result of running the simple script with PyPy interpreter, which does the job a bit faster. It is PyPy 7.3.19 with MSC v.1941 64 bit (AMD64)

- pandas – employ pandas to read CSV. The solver formula is applied to each row as lambda. And all is saved back to result.csv. Very simple and quite fast.

```
QE> python mr-qe-pandas.py

👍 : 100000007 records, 3 without roots, calculation time=285.117 sec

👍 : Total running time=394.270 sec

😀
```

- polars – this is very fast. The solver implementation looks SQL than Pythonic but if speed matters then this is the way.

Quadratic Equations

Machine	Interpreter	Script	Time (sec)
HPW	python 3.12	simple	231.509
		pandas	394.270
		polars	13.175
	PyPy3 7.3.19	simple	221.923

Ubuntu	python 3.12	simple	306.722
		pandas	581.537
		polars	14.245

With dataframes, it is simple to separate two measurements: how much time it takes to read and solve all equations and how long writing into the file takes. The result table shows the total time.

Python Solutions of Cubic Equations

Solving cubical equations, Polars is still fastest of all. The code it runs is more involved and it took more time to implement but results pay back.

Cubic Equations

Machine	Interpreter	Script	Time (sec)
HPW	python 3.12	simple	403.991
		pandas	509.068
		polars	265.368
	PyPy	simple	309.957
Ubuntu	python 3.12	simple	472.752
		pandas	675.582
		polars	362.706

The C++ Implementation

I decided to use C++ 17.
On HPW it was Microsoft Visual Studio 2022 and on Ubuntu g++ 11.4.0.

The straightforward C++ code was very disappointing. On HPW machine it took: 1446 sec. Very slow, very sad… But this is C++: std::cout is slow, copy constructors are everywhere, like viruses, strings are big, container’s allocators are slow, pointers are dangerous. And not forget to use optimization. With all this in mind, I finished having the relatively simple but pleasantly fast implementation, which solves 100,000,000 quadratic equations in 70 seconds. This improvement is gained thanks to optimized input reading (read_csv.h), parallel processing of equations – std::par and output based on fprintf() instead of std::cout. And I did not use in-memory files and explicit multithreaded processing so there should be much room for improvement.

Quadratic Equations, C++ Solution

Machine	Compiler	Script	Time (sec)
HPW	MS Visual Studio 2022	simple	1446.28
		optimize CSV reading	430.6
		and std::par processing	371
Ubuntu	g++ 11.4	simple	142.83
		optimize CSV reading	79.57
		and std::par processing	69.71

Conclusion

Above are the few experiments with relatively big sets of data. Was it worth all the time and effort? I guess it was...

Mind, that I did not bother with in depths optimizations to win 5 seconds. I did try to run the quadratic solution, using the Dask, hoping to reproduce the lightning speed of mr-qe-polars, but to my surprise the solution based of the Dask dataframe took more than 800 seconds. Go figure… (I admit I am not an expert at Dask)

But there definitely will be more, as Python is evolving, and more interesting tools are coming. There are possibilities to use Modin or research multithreaded processing. Perhaps refactoring panda’s scripts with vectorization would win some more, but it is a question for the next vacation.

All scripts and C++ code can be cloned from here: github.coder-sasha/many-records

Happy coding!