Laboratory 4: How to compare 1D arrays

The aims of today's lab are:

- Use the STL vector class to store an array of `float` values;
- Use built-in functions to compute the min, max, and sum of the array;
- Compare two arrays using the Sum of Absolute Errors (SAE) and the normalised cross-correlation (NCC).
- Compare two arrays using `operator==` and `operator!=`.

Task 0: Using CMake

Same as usual, we will use CMake to make our lives easier.

Task 1: Min/Max/Sum/Average/Variance/Standard deviation

You have been given a ZIP file containing a small (incomplete class). For this task, you mostly have to modify `MyVector.cpp`. The methods you need to complete are:

1. `float getMinValue() const` (see previos labs)
2. `float getMaxValue() const` (see previos labs)
3. `float getSum() const` (see previos labs)
4. `float getAverage() const`
5. `float getVariance() const`
6. `float getStandardDeviation() const`

In the ZIP file, you also got 4 ASCII files:

- `y.mat`
- `y_quadriple.mat`
- `y_noise.mat`
- `y_negative.mat`

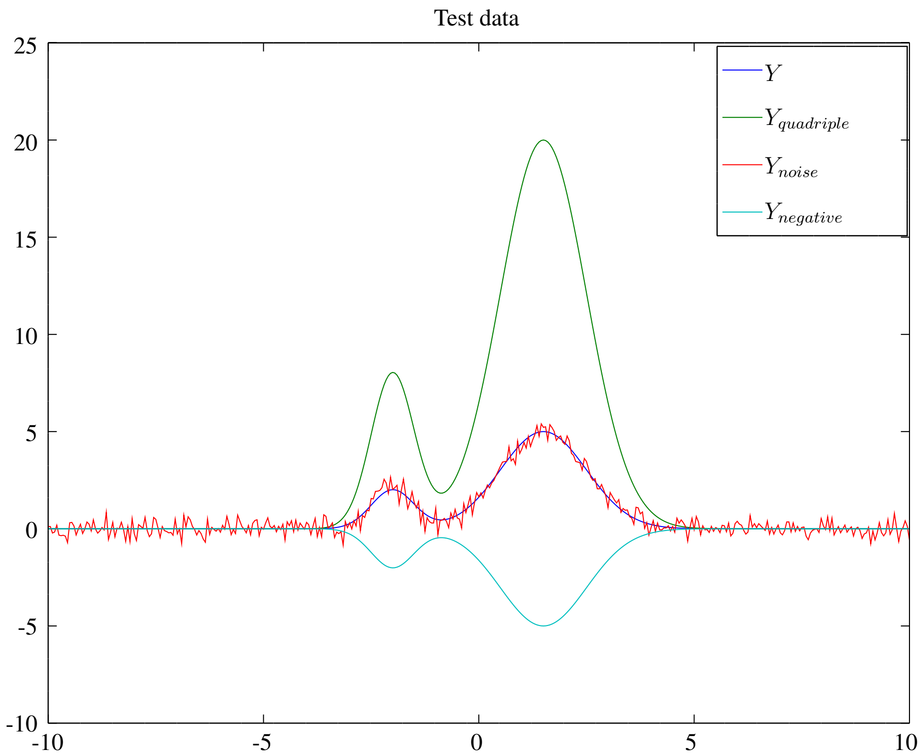

Figure 1: Test data from the ASCII files.

Table 1: Statistics about the test data from Figure 1

Test case	**Min**	**Max**	**Sum**	**Average**	**Variance** (σ^2)	**Standard deviation** (σ)
Y	9.5774e-29	5.0000	300.80	0.75011	1.8371	1.3554
$Y_{quadriple}$	3.8310e-28	20.000	1203.2	3.0005	29.393	5.4216
Y_{noise}	-0.83714	5.3959	303.32	0.75641	1.9532	1.3976
$Y_{negative}$	-5.0000	-9.5774e-29	-300.80	-0.75011	1.8371	1.3554

They contain test data that you can use to assess your code. Figure 1 shows the content of the files. You can load each file in independent instances of the class `MyVector`. Table 1 provides statistics about the test data from Figure 1. You can use them to compare the results of your computations.

Equations 1 to 3 show how to compute the average, variance and standard deviation of a vector X of N elements:

$$Average(\mathbf{X}) = \mu_X = \overline{X} = \frac{\sum X}{N} \tag{1}$$

$$Variance(\mathbf{X}) = \sigma_X^2 = \frac{\sum X - \mu_X}{N} \tag{2}$$

$$Standard\,deviation(\mathbf{X}) = \sigma_X = \sqrt{Variance(\mathbf{X})} \tag{3}$$

Task 2: Numerical inaccuracy

You may have seen some discrepancies between the values of Table 1 and the ones you computed. The differences should be extremely small. The reason is called *numerical inaccuracy*. Create a new test program `numerical_inaccuracy.cpp`. You need to add it to your `CMakeLists.txt`. You need the headers as follows:

- `iostream` for printing text in the standard output;
- `iomanip` to control how many digits are printed after the dot;
- `cmath` to use some mathematical functions.

Create 5 single-precision floating point numbers (32 bit) `i`, `j`, `k`, `l`, and `m` so that:

- `i` = 10.1111;
- `j` = 20.2222;
- `k` = i + j;
- `l` = j + i; and
- `m` = 30.3333.

One would expect `k`, `l` and `m` to be equal to 30.3333. Print the value in the console with:

```
std::cout << k << "\t" << ll << "\t" << m << std::endl;
```

It seems to be the case. Let us check this with:

```
std::cout << (k == l?"SAME":"DIFFERENT") << std::endl;
std::cout << (m == k?"SAME":"DIFFERENT") << std::endl;
std::cout << (m == l?"SAME":"DIFFERENT") << std::endl;
```

As expected `k` is equal to `l`. However, `m` is not equal to `k` and `m` is not equal to `l`. In other words, `30.3333` is not equal to `30.3333` and `30.3333` is not equal to `30.3333`.

What is going on???

Let us add more zeros after the dot with:

```
std::cout << std::setprecision(17) << k << "\t" <<
    std::setprecision(17) << l << "\t" <<
    std::setprecision(17) << m << std::endl;
```

`k` and `l` are equal to 30.333301544189453 but `m` is equal to 30.33329963684082. This is due to what is called numerical inaccuracy. At a rule of thumb, **DO NOT USE == and != with floating point numbers** (`float` or `double`).

What should we do then???

Check how close to numbers are. If the absolute difference is smaller than a threshold (ϵ) then consider that the two numbers are equal. If not, they are different.

Implement a new function:

```
bool isEqual(float i, float j);
```

and let us say that `EPSILON` is equal to 0.00001. If the absolute difference between `i` and `j` is smaller than `EPSILON` then `isEqual` will return `true`, else it will return `false`.

Now call:

```
cout << (isEqual(k, l)?"SAME":"DIFFERENT") << endl;
cout << (isEqual(m, k)?"SAME":"DIFFERENT") << endl;
cout << (isEqual(m, l)?"SAME":"DIFFERENT") << endl;
```

Task 3: operator== and operator!=

Add:

- `bool MyVector::operator==(const MyVector& aVector) const;`
- `bool MyVector::operator!=(const MyVector& aVector) const;`

Note that to limit the scope for errors we will reuse the code of `operator==` in the implementation of `operator!=`:

```
bool MyVector::operator!=(const MyVector& aVector) const
{
        return (!((*this) == aVector));
}
```

In the implementation of `operator==` use the same technique as what we saw previously in Task 2. To try your new operator, add the code as follows in your test program `test_my_vector.cpp`:

```
MyVector temp(y_quadruple / 4.0);
std::cout << (y == y_quadruple?"SAME":"DIFFERENT") << std::endl;
```

Task 4: How dissimilar two vectors are: the SAE

SAE stands for sum of absolute errors. It is also called sum of absolute distance (SAD), Manhattan distance, and L^1-norm. In statistics, it is used as a quantity to measure how far two vectors are from each other. The SAE between two vectors Y_1 and Y_2 of N element is:

$$SAE(Y_1, Y_2) = \sum_{i=0}^{N-1} |Y_1(i) - Y_2(i)| \tag{4}$$

Add the method as follows in your class:

```
float MyVector::SAE(const MyVector& aVector) const;
```

One of the main advantages of the SAE is that it is fast to compute. However, it has limitations. To test your computations, here are the results for:

- $SAE(y, y_{quadriple}) = 902.39$
- $SAE(y, y_{negative}) = 601.59$
- $SAE(y, y_{noise}) = 108.52$

$y_{quadriple}$ is equal to $4 \times Y$. However, the SAE between $y_{quadriple}$ and Y is the largest. $y_{negative}$ is equal to $-Y$. However, the SAE between $y_{negative}$ and Y is the second largest.

Task 5: How similar two vectors are: the NCC

NCC stands for normalised cross-correlation. The normalisation in NCC addresses the limitation highlighted in our tests. The formula is:

$$NCC(Y_1, Y_2) = \sum_{i=0}^{N-1} \frac{(Y_1(i) \times \overline{Y_1})(Y_2(i) \times \overline{Y_2})}{\sigma_{Y_1}\sigma_{Y_2}} \tag{5}$$

- $NCC(Y_1, Y_2)$ = 1, if Y_1 and Y_2 are fully correlated (e.g. $Y_1 = \alpha Y_2$);
- $NCC(Y_1, Y_2)$ = -1, if Y_1 and Y_2 are fully anti-correlated (e.g. $Y_1 = -\alpha Y_2$);
- $NCC(Y_1, Y_2)$ = 0, if Y_1 and Y_2 are fully uncorrelated (they are unrelated).

Often the NCC is expressed as a percentage. With our examples, we get:

- $NCC(y, y_{quadriple}) = 1 = 100\%$
- $NCC(y, y_{negative}) = -1 = -100\%$
- $NCC(y, y_{noise}) = 0.97 = 97\%$

Summary

Today, you saw how to compare 1D vectors in four different ways:

- `operator==`
- `operator!=`
- SAE
- NCC

All of them got advantages and disadvantages. You will adapt them to your next assignment.