Laboratory 5: Comparing arrays of floating-point numbers

Most floating-point numbers end up being slightly inaccurate due to rounding errors. It means that numbers expected to be equal (e.g. when calculating the same result through different correct methods) often differ, even if very slightly. It means that simple equality tests are likely to fail:

```
float a = 0.15 + 0.15
float b = 0.1 + 0.2
if(a == b) // can be false!
if(a >= b) // can also be false!
```

Our 1-D arrays today, or 2-D images in the assignments, store values as floating-point numbers. The aims of today's lab are to compare two arrays of floating-point numbers using:

- `operator==` and `operator!=` (you'll have to overwrite one of them).
- Sum of Absolute Errors (SAE),
- Mean Absolute Errors (MAE),
- Sum of Squared Errors (SSE),
- Mean Squared Errors (MSE),
- Root Mean Squared Errors (RMSE), and
- Zero Mean Normalised Cross-Correlation (ZNCC).

Note, today you are working on 1-D arrays. In the 2nd assignment, you'll have to do the same but on 2-D images.

Task 1: Where is the code?

We are going to use the same source code as Lab 4. We'll rely on your implementation of

1. `float getMinValue() const` (see `std::min_element` previous labs)
2. `float getMaxValue() const` (see `std::max_element` previous labs)
3. `float getSum() const` (see `std::accumulate` previous labs)
4. `float getAverage() const`
5. `float getVariance() const`
6. `float getStandardDeviation() const`

We will use the same ASCII files:

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

that contain the test data you can use to assess your code. Figure 1 shows the content of the files.

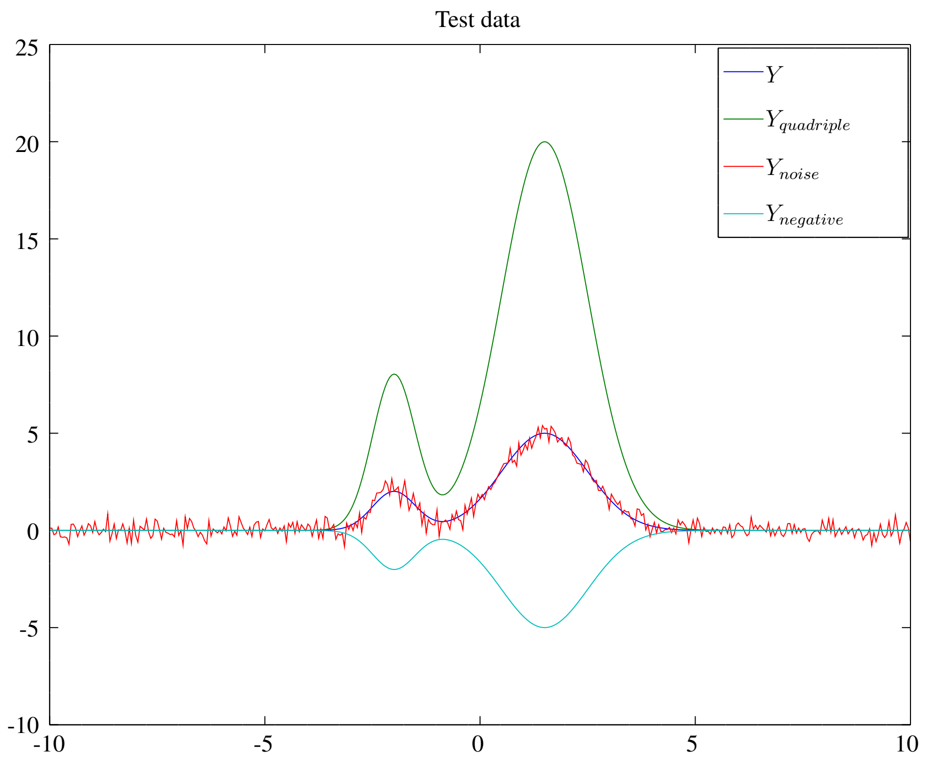

Figure 1: Test data from the ASCII files.

Task 2: Task 2: 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=" << k << "\tl=" << l << "\tm=" << m << std::endl;
```

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

```
std::cout << k << "␣is␣" << (k == l?"the␣SAME␣as␣":"DIFFERENT␣from␣") << l
    << std::endl;
std::cout << k << "␣is␣" << (k == m?"the␣SAME␣as␣":"DIFFERENT␣from␣") << m
    << std::endl;
std::cout << l << "␣is␣" << (l == m?"the␣SAME␣as␣":"DIFFERENT␣from␣") << m
    << std::endl;
```

The output I get is as follows:

```
k=30.3333   l=30.3333    m=30.3333
30.3333 is the SAME as 30.3333
30.3333 is DIFFERENT from 30.3333
30.3333 is DIFFERENT from 30.3333
```

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???

Absolute difference

The solution is to check not whether the numbers are exactly the same, but whether their difference is very small. The error margin that the difference is compared to is often called "epsilon". The most simple form is called absolute difference:

```
if (abs(a - b) < 0.00001) // wrong - don't do this
```

This is a bad way to do it because a fixed epsilon chosen because it "looks small" could actually be way too large when the numbers being compared are very small as well. In this case the comparison would return `true` for numbers that are quite different.

Create a new function:

```
bool isEqual(double a, double b, double epsilon = 0.00001)
{
    // Check equality using absolute difference
    return (abs(a - b) < epsilon);
}
```

and replace == in your code with the corresponding calls of `isEqual`. If the absolute difference is smaller than a threshold `epsilon` then consider that the two numbers are equal. If not, they are different. Test your new program. It seems better, but...

Relative difference

when the numbers are very large, the epsilon could end up being smaller than the smallest rounding error, so that the comparison always returns `false`. Therefore, it is necessary to consider if the relative error is smaller than epsilon:

```
if (abs(a - b) / b < epsilon) // still not right!
```

We should replace our function with:

```
bool isEqual(double a, double b, double epsilon = 0.00001)
{
    // Check equality using absolute difference
    //return (abs(a - b) < epsilon);

    // Check equality using relative difference
    return ((abs(a - b) / b) < epsilon);
}
```

But...

- when both `a` and `b` are zero. `0.0/0.0` is "not a number", which causes an exception on some platforms, or returns false for all comparisons.
- when only `b` is zero, the division yields `infinity`, which may also cause an exception, or is greater than epsilon even when a is smaller.
- it returns false when both `a` and `b` are very small but on opposite sides of zero, even when theyre the smallest possible non-zero numbers.

Final version of isEqual

Make sure you include these new headers:

```
#include <climits>
#include <algorithm>
```

And replace the function with:

```
    bool isEqual(double a, double b, double epsilon = 0.00001)
    {
        // Check equality using absolute difference
        //return (abs(a - b) < epsilon);

        // Check equality using relative difference
        //return ((abs(a - b) / b) < epsilon);

        // Handle infinities
        if (a == b) return true;

        double abs_A    = abs(a);
        double abs_B    = abs(b);
        double abs_diff = abs(a - b);

        // if a or b is zero or both are extremely close to it
        // then relative error is less meaningful
        if (a == 0 || b == 0 || (abs_A + abs_B < std::numeric_limits<float>::min
           ()))
            return abs_diff < std::numeric_limits<float>::min();
        // use relative error
        else
            return ((abs_diff / std::min(abs_A, abs_B)) < epsilon);
    }
```

Task 3: Task 3: operator== and operator!=

Go back to the `MyVector` class. We are going to edit:

- `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 our test program, we created:

```
        MyVector temp3(y + y + y + y);
```

It should be the same as $4 \times$ `y`. However, `temp3 == y_quadriple` always returns false despite the cout showing the same numerical values. It's due to rounding errors. In the implementation of `operator==` we MUST use the same technique as what we saw previously in Task 2. Add the `isEqual()` function to `MyVector.cpp`, e.g. at the top of the file. Make sure the header files are included. In `bool MyVector::operator==(const MyVector& aVector) const;`, comment `if (*ite0+` != *ite1++) return (false);+ and use `isEqual()`, e.g.

```
    if (!isEqual(*ite0++, *ite1++)) return (false);
```

Re-run your test program. Now `temp3 == y_quadriple` returns true.

With our new operators, we can check if our 1-D vectors, or 2-D images, are equal or different. We often want to quantify the amount of dissimilarity between two images in computer vision applications. In this case we rely on error (or distance) metrics. An error close to zero means that the two images are similar. The larger the error, the more dissimilar they are.

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 $\mathbf{Y_1}$ and $\mathbf{Y_2}$ of N element is:

$$SAE(\mathbf{Y_1}, \mathbf{Y_2}) = \sum_{i=0}^{i<N} |\mathbf{Y_1}(i) - \mathbf{Y_2}(i)| \quad (1)$$

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:

- $SAE(\mathbf{Y}, \mathbf{Y}) = 0$
- $SAE(\mathbf{Y}, \mathbf{Y_{quadriple}}) = 902.386$
- $SAE(\mathbf{Y}, \mathbf{Y_{negative}}) = 601.591$
- $SAE(\mathbf{Y}, \mathbf{Y_{noise}}) = 104.505$

$\mathbf{Y_{quadriple}}$ is equal to $4 \times \mathbf{Y}$. However, the SAE between $\mathbf{Y_{quadriple}}$ and $\mathbf{Y}$ is the largest. $\mathbf{Y_{negative}}$ is equal to $-\mathbf{Y}$. However, the SAE between $\mathbf{Y_{negative}}$ and $\mathbf{Y}$ is the second largest. We can conclude that even if SAE is commonly used in imaging it may not provide a good error metrics in some cases.

Task 5: How dissimilar two vectors are: the MAE

Sometimes, the Mean Absolute Error (MAE) is used. To get it, just divide the SAE by the number of samples:

$$MAE(\mathbf{Y_1}, \mathbf{Y_2}) = \frac{SAE(\mathbf{Y_1}, \mathbf{Y_2})}{N} \tag{2}$$

To test your computations, here are the results for:

- $MAE(\mathbf{Y}, \mathbf{Y}) = 0$
- $MAE(\mathbf{Y}, \mathbf{Y_{quadriple}}) = 2.2503$
- $MAE(\mathbf{Y}, \mathbf{Y_{negative}}) = 1.5002$
- $MAE(\mathbf{Y}, \mathbf{Y_{noise}}) = 0.26061$

Task 6: How dissimilar two vectors are: the SSE

SSE stands for sum of squared errors. It is also called sum of squared distance (SAD), Euclidean distance, and L^2-norm.

$$SSE(\mathbf{Y_1}, \mathbf{Y_2}) = \sum_{i=0}^{i<N} (\mathbf{Y_1}(i) - \mathbf{Y_2}(i))^2 \tag{3}$$

To test your computations, here are the results for:

- $SSE(\mathbf{Y}, \mathbf{Y}) = 0$
- $SSE(\mathbf{Y}, \mathbf{Y_{quadriple}}) = 8644.2$
- $SSE(\mathbf{Y}, \mathbf{Y_{negative}}) = 3841.9$
- $SSE(\mathbf{Y}, \mathbf{Y_{noise}}) = 43.341$

Task 7: How dissimilar two vectors are: the MSE

Sometimes, the Mean Squared Error (MSE) is used. To get it, just divide the SSE by the number of samples:

$$MSE(\mathbf{Y_1}, \mathbf{Y_2}) = \frac{SSE(\mathbf{Y_1}, \mathbf{Y_2})}{N} \tag{4}$$

To test your computations, here are the results for:

- $MSE(\mathbf{Y}, \mathbf{Y}) = 0$
- $MSE(\mathbf{Y}, \mathbf{Y_{quadriple}}) = 21.557$
- $MSE(\mathbf{Y}, \mathbf{Y_{negative}}) = 9.5807$
- $MSE(\mathbf{Y}, \mathbf{Y_{noise}}) = 0.10808$

Task 8: How dissimilar two vectors are: the RMSE

Sometimes, the Root Mean Squared Error (RMSE) is used. To get it, just return the square root of MSE:

$$RMSE(\mathbf{Y_1}, \mathbf{Y_2}) = \sqrt{MSE(\mathbf{Y_1}, \mathbf{Y_2})} \tag{5}$$

To test your computations, here are the results for:

- $RMSE(\mathbf{Y}, \mathbf{Y}) = 0$
- $RMSE(\mathbf{Y}, \mathbf{Y_{quadriple}}) = 4.6429$
- $RMSE(\mathbf{Y}, \mathbf{Y_{negative}}) = 3.0953$
- $RMSE(\mathbf{Y}, \mathbf{Y_{noise}}) = 0.32876$

Task 9: How similar two vectors are: the ZNCC

ZNCC stands for Zero Mean Normalised Cross-Correlation. The normalisation in ZNCC addresses the limitation highlighted in our tests. The formula is:

$$ZNCC(\mathbf{Y_1}, \mathbf{Y_2}) = \frac{1}{N} \sum_{i=0}^{i<N} \frac{(\mathbf{Y_1}(i) - \overline{\mathbf{Y_1}})(\mathbf{Y_2}(i) - \overline{\mathbf{Y_2}})}{\sigma_{Y_1} \sigma_{Y_2}} \tag{6}$$

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

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

- $ZNCC(\mathbf{Y}, \mathbf{Y}) = 0.999996 = 99.9996\%$
- $ZNCC(\mathbf{Y}, \mathbf{Y_{quadriple}}) = 0.999996 = 99.9996\%$
- $ZNCC(\mathbf{Y}, \mathbf{Y_{negative}}) = -0.999996 = -99.9996\%$
- $ZNCC(\mathbf{Y}, \mathbf{Y_{noise}}) = 0.97188 = 97.188\%$

Task 10: Error checks

Have you thought about what to do if the size of the two arrays that you are comparing is different? If you haven't revisit Tasks 3 to 8 to make sure that the program won't crash.