PCSC Project Group 5—Data Approximation

Yuxuan Long (307335), Yiting Zhang (307065)
École Polytechnique Fédérale de Lausanne, Switzerland

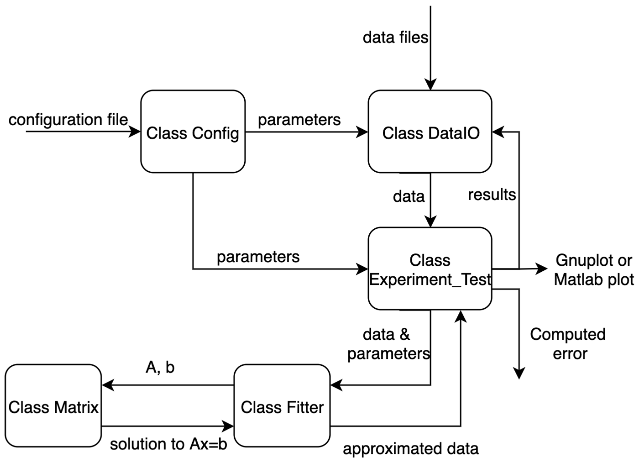

Figure 1: Function flow of the project

I. Introduction

In this project, we aim to implement numerical algorithms for data approximation in C++. In the context of interpolation, we are given a sequence of 2D points $\{(x_i, y_i)\}_{i=1}^{N}$, and a function $\hat{f}$ to be approximated must satisfy $\hat{f}(x_i) = y_i, \forall i$. In our project, we focus on real values and functions, namely $x, y \in \mathbb{R}$ and $\hat{f} : \mathbb{R} \to \mathbb{R}$.
For the parameters mentioned in this report, they refer to the ones in the configuration file (same as in ReadMe), not in the main function. For ease to distinguish, the parameters appear as italic letters.

II. Compiling

To compile our project, first change the current directory to the top directory of the project, then just type the commands step by step:

```
mkdir build
cd build
cmake ..
make
```

In this way, Cmake will compile the project in the new directory build. Under this folder, there are two separate executable files: core/core and tests/int_test. Cmake will also generate all Doxygen files into folder doc.

III. Function Flow

The figure 1 shows the function flow of our main codes in the project. Besides the fact that the configuration file is an input, another input can be the input data files. The configuration file passes all the parameters to the class DataIO and Experiment_Test, like the range of x_i. Besides DataIO can have methods like file reading and writing, it can also generate new input data and test data. Both input data and test data are generated by the true function f, namely $f(x_i) = y_i, \forall i$. We record them as two columns in the data file.
The data are read into vectors (using Stardard Template Library). We here use the denotation like the vector $\mathbf{x}$ which collects x_i. In the class Experiment_Test, it sends parameters and data to class Fitter in order to complete the data approximation. During this procedure, the class Fitter can receive the name of the approximation method, and input data sequence $\mathbf{x}, \mathbf{y} \in \mathbb{R}^N$ and test points $\mathbf{x}_{test} \in \mathbb{R}^{N_{test}}$. If a specific method needs to solve a linear system (not all methods need to solve linear system), then the Fitter sends a request to the class Matrix. Hence, the only things passed to Matrix can be $\mathbf{A}$ (a Matrix object) and $\mathbf{b}$, then Matrix gives the solution to $\mathbf{A}\mathbf{w} = \mathbf{b}$. This solution is usually the parameters of the approximated function, so Fitter can pass the $\mathbf{x}_{test}$ to the approximated function to yield the approximated data. Back to the class Experiment_Test, approximation results are sent to DataIO so that an output file is written. This output file can be used for visualization of the approximated function, e.g. in Matlab. The error of the approximation compared to the true values is also computed as an output of the flow.

IV. Data Generation

The data we use for approximation in our project can be generated from three types of function: trigonometric function: $\cos(3\pi x)$, Runge function: $1/(1 + x^2)$ and polynomial function: $x^3 - 2x + 3$. Users can choose between different types of function by the parameter, as referred to ReadMe.

Data points are generated in two kinds of distribution. One is uniform distribution with equal gap between two neighbor data points. The other one is Chebychev-Gauss-Lobatto (CGL) node distribution. The CGL nodes can be a remedy to Runge's phenomenon and improve the stability. In the internal $I[a, b]$ the formula for CGL nodes is:

$$x_i = \frac{a+b}{2} - \frac{a-b}{2}\cos(\frac{\pi}{N-1}i), i = 0, 1, ..., N-1$$

Note that CGL nodes are not used for Fourier interpolation.

V. Algorithm Implementation

A. Linear System

Besides the class Matrix includes basic matrix operations like multiplication, it also provides the method *gauss_solve* to solve linear system $\mathbf{A}\mathbf{w} = \mathbf{b}$, by Gaussian elimination and backward substitution. Note that the matrix stored in the object Matrix is just a 2D C++ array. For the spline and Lagrange polynomial interpolation (discussed in next subsection), $\mathbf{A}$ is square so that calling *gauss_solve* once can solve the parameters. For the polynomial interpolation, at the common case that the polynomial degree is not large, the interpolation becomes a data fitting problem (but we still call it polynomial interpolation for convenience). At such case, we use least squares (*least_squares* in Matrix class):

$$\mathbf{w}^* = (\mathbf{A}^T\mathbf{A} + \lambda\mathbf{I})^{-1}\mathbf{A}^T\mathbf{b}$$

λ is non-negative small number. A positive λ indicates ridge regression, a variant of least squares. In the configuration file, we can find λ as the parameter *polynomial_lambda*.

B. Polynomial Interpolation

Polynomial interpolation has two special variants, which are ridge regression and Lagrange polynomial interpolation. For Lagrange interpolation, we can set the polynomial degree (*polynomial_degree* in configuration) that is equal to the number of the data points minus one.

C. Piecewise Polynomial Interpolation

For both linear piecewise interpolation and cubic spline, extrapolation is also allowed if the test points are out of the input data range. Note that the cubic spline in our project uses natural condition at the two ending points, i.e. zero second derivative.

D. Fourier Interpolation

Discrete Cosine Transform (DCT) is implemented for Fourier interpolation since our data is in real domain.

VI. Experimental Test

In the experiment, we set *num_input_points*=41 and *num_test_points*=1001. The range in x coordinate of the test data is set to be same as the range of input data. The error for each test is measured as $e(f, \hat{f}) = \max|f(x_{test}) - \hat{f}(x_{test})|$. The range of x is $[-10, 10]$ (*x_input_min*=-10 and *x_input_max*=10). For the polynomial interpolation, by default, we set *polynomial_degree* = 20.

A. Test on Runge function

The input data are now generated from Runge function (*function_type*=2). We set that the nodes are uniformly distributed (*node_type* = 1). Gnuplot is used to show the result, i.e. *use_gnuplot*=1.
Then, we test several approximation methods. From figure 2

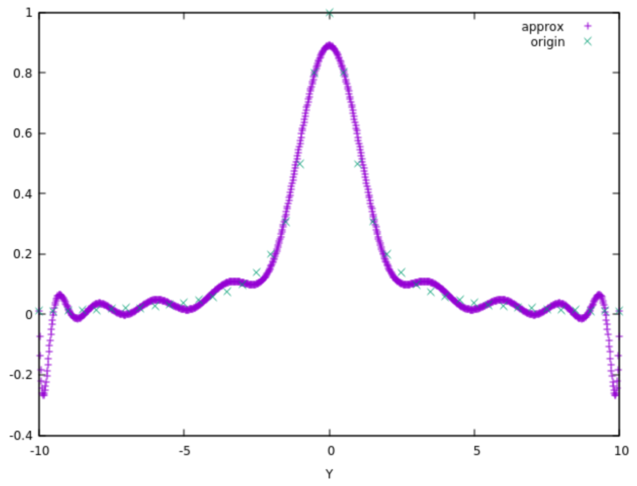

Figure 2: Polynomial interpolation

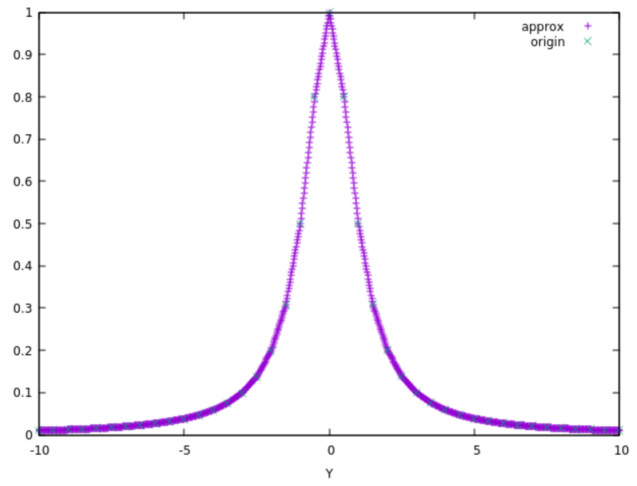

Figure 3: Runge function by linear piecewise interpolation

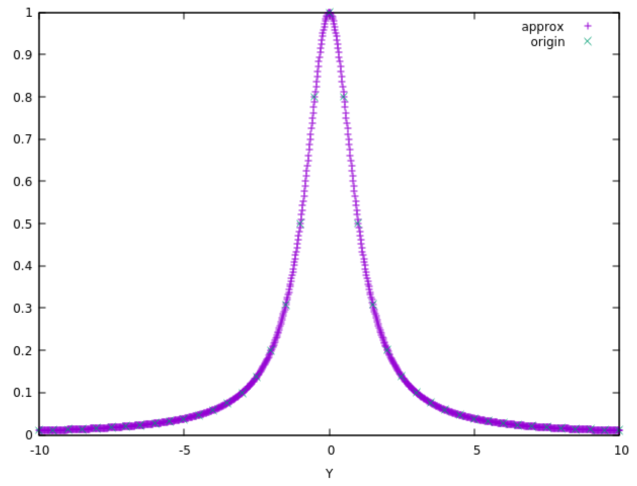

Figure 4: Runge function by spline interpolation

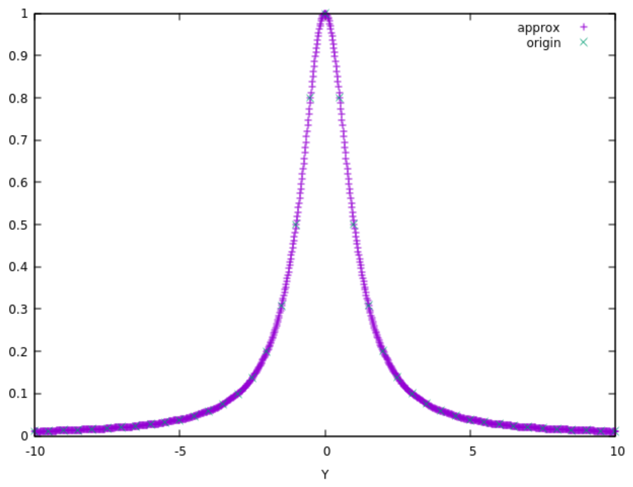

Figure 5: Runge function by Fourier interpolation

polynomial	linear piecewise	spline	Fourier
2.77×10^{-1}	4.18×10^{-2}	3.18×10^{-3}	1.84×10^{-3}

Table I: Error of approximation by different methods

to 5, we show the results by all methods except two variants of polynomial interpolation.

Table I shows the error of the approximation by different methods. It can be found that the Fourier interpolation have the least error. We also test the variant of polynomial interpolation, like Lagrange polynomial by setting *polynomial_degree* = 40, and we obtain an error of 3.08×10^5, which fits the Runge's phenomenon. Ridge regression with small λ performs almost same as the common polynomial fitting.

B. Test on trigonometric and polynomial function

We use Fourier interpolation to approximate the input data generated from three kinds of functions. The parameters are set same as last section (except the name of the method), so same input data and test data have been used. Results are shown from Figure 6 and 7. We find that the Fourier interpolation has a nice result when approximating the trigonometric function. But it does not well approximate the polynomial function. On the other hand, in figure 8 (plot by the Matlab script 'matlab_plot' in the folder core/plot), the polynomial interpolation gives a very good result.

C. Test on different nodes

The polynomial interpolation is very sensitive to the input nodes distribution. Here, we keep the same input and test data, and we use the polynomial interpolation to approximate the Runge function with two kinds of node distributions. From Figure 9 and 10, it can be found that, when CGL nodes are used (i.e. *node_type*=2), the Runge phenomenon is obviously alleviated.

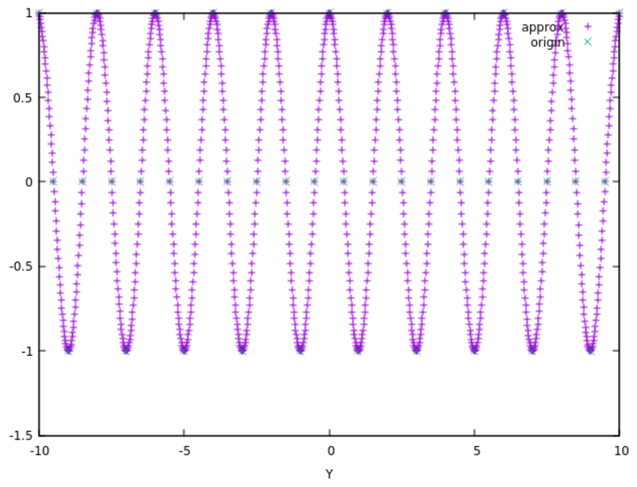

Figure 6: Trigonometric function by Fourier interpolation

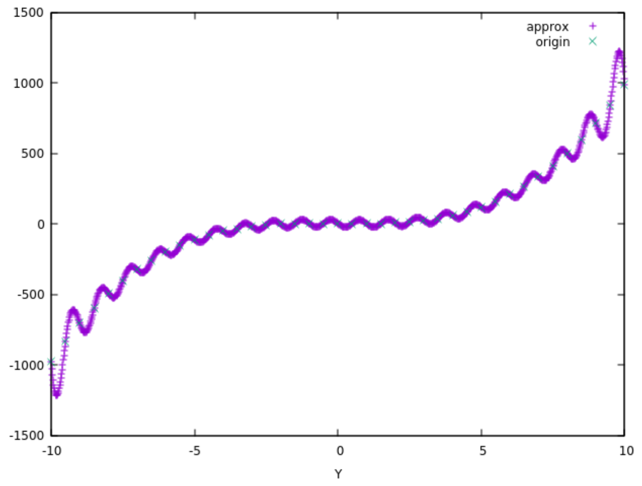

Figure 7: Polynomial function by Fourier interpolation

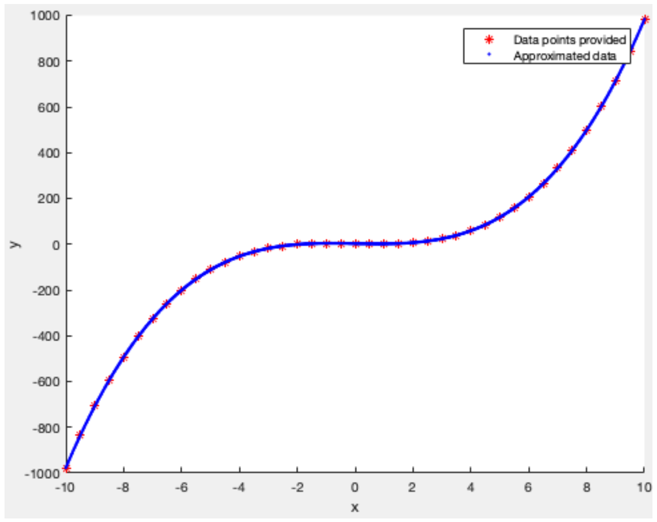

Figure 8: Polynomial function by polynomial interpolation

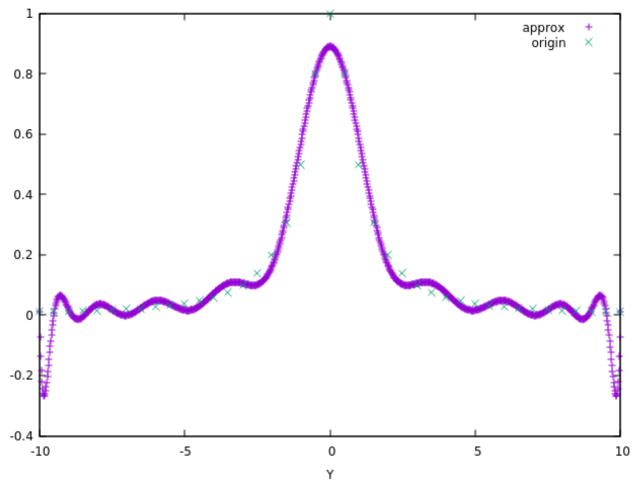

Figure 9: Polynomial interpolation using uniformly distributed nodes

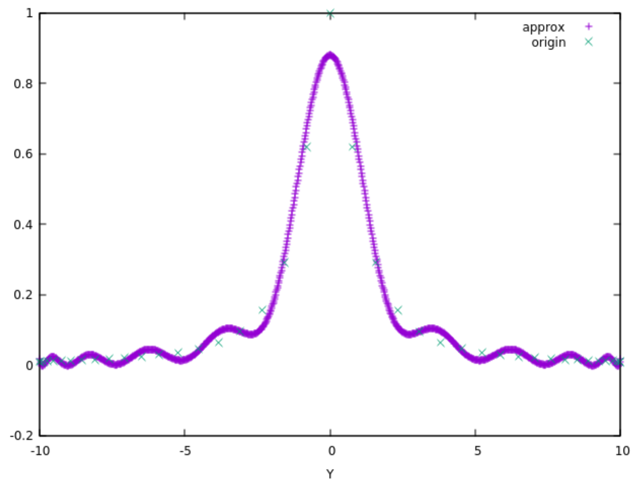

Figure 10: Polynomial interpolation using CGL nodes

VII. Google Test

The main purpose of the test is to make sure that our implementation of algorithms is correct. Since some built-in functions in MATLAB can both directly and reliably approximate data, we use them to judge the correctness our results. For all the tests, we use Runge function as the approximation target and the same setup for data (both input and output) used at the last subsection VI-A. By running 'generate_gt.m' in the folder tests/matlab_data, output data files (already provided) from all methods by Matlab built-in functions are generated. In the tests, we calculate the mean square error (MSE) between the our results and MATLAB results (the y-coordinates) for each approximation method.
Table II shows the MSE of the approximation by different methods. Note that we set the data file precision to be 8 digits for both C++ and Matlab outputs. Therefore, the only concern is that our polynomial interpolation may be implemented in a slightly different way with Matlab functions.

polynomial	linear piecewise	spline	Fourier
1.11×10^{-8}	1.73×10^{-18}	2.36×10^{-18}	1.22×10^{-17}

Table II: MSE between our C++ results and MATLAB results by different methods

In the google test, we set a tolerence value of 10^{-7}, which is used to judge if the test passes. The passing condition is only that the computed MSE is smaller than this tolerance.

VIII. Limitations and Future Work

A. User interface

From user perspective, the current interface is built on a configuration file and may still need enhancement, like advanced input checking. An example can be the function used to approximate. Currently, we only provide three kinds of functions for the user. So, a better user interface is to allow the user to input the function which they would like to test. This is somehow challenging since the input strings have to be converted into a function.
The error handling is expected to have improvement in the future. For instance, if the input data file is provided by the user (*use_file* = 1), we should check if the file has the valid data format (i.e. two columns).
For the problem that how to make more convenience for the user, it remains as a challenge to the program design in the future.

B. Algorithm Implementations

In the project, cubic spline is implemented by solving a large $4(N-1) \times 4(N-1)$ linear system, using the natural condition. In the future work, the matrix in the system could be formulated as a tridiagonal matrix, which allows more efficient solving. In the future, we could implement another version of spline using not-a-knot condition, which matches exactly with Matlab built-in function *spline*.
Fast Fourier Transform (FFT) on real data could be more efficient than current Discrete Cosine Transform (DCT). So, FFT can be worth to implement in the future.

C. Test

For the test of the approximation error, in fact, there exists some theoretical upper bound on the error for each specific approximation method. In the future work, one additional interest could be to inspect the difference between the practical error and theoretical error.
For the google test, we should test on various functions and inspect the MSE between our results and Matlab results. This may need more structured design, which may allow the user to easily perform tests with different parameters.