Project 6: Data Approximation

Kseniia Shevchenko, Justine Stoll

Course of Programming Concepts in Scientific Computing (MATH-458), EPFL

I. Introduction

The aim of this project was to create a program that implements data approximation and interpolation using well known techniques. One of the key features of our program is that the input data can be read from files.

A model is fitted with one of the interpolation or approximation techniques, and can then be applied to new data and evaluated at new points.

An important distinction is made between the data that is used to fit the model, and the one on which we use the model. In this report we will refer to the first input file, containing the sampling points, as **file_1**. The second input file, containing the points at which to evaluate the interpolated or approximated function, as **file_2**.

In addition, we will refer to sampling points as couple of points of the form: (x_i, y_i), i=0...n, where n+1 is the number of data points. Indeed this initial version of our program only supports two dimensional data. Finally, we also implemented a tests for the individual parts of our program.

II. Overview of the project

The following shows the directory tree of our project.

```
PCSC_data_approximation
├── data
├── doc
├── googletest
├── libraries
│   └── eigen
├── src
└── test
```

In addition, the file *src* contains the totality of the code necessary for the execution of the project. Also, all files containing the data to be used by the program should be placed in the file *data*. An overview of the classes is shown in figure 1 and is explained more thoroughly subsequently.

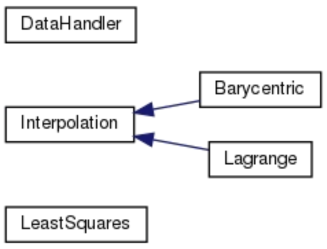


Fig. 1: Schematic representation of the class hierarchy for our program.

a. How to compile the program

First, clone the project with the command: git clone https://github.com/jstoll1233/PCSC_data_approximation.git

Before you can compile this program you have to make sure that the following libraries are properly installed:

– **eigen**. This repository can be cloned into a folder *libraries* as indicated in the project tree, like this:

  git clone https://gitlab.com/libeigen/eigen.git

– **googletest**. This repository can be cloned at the same level as the folder *test* like this:

  git clone https://github.com/google/googletest.git

Place yourself in the folder *PCSC_data_approximation*. The following commands compile and execute the program.

– mkdir build

– cd build

– cmake ..

– make

– ./data_approximation (or ./test_pcsc for execution of tests)

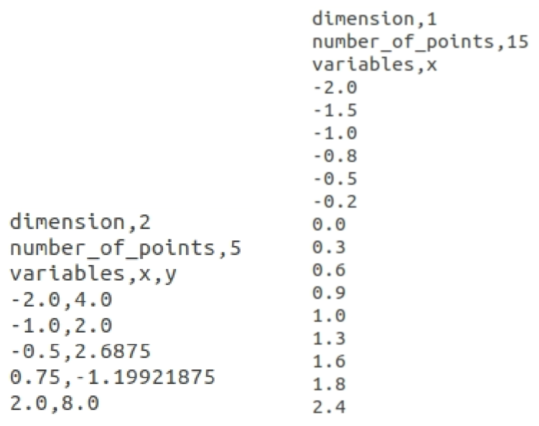
The program will then ask the user for two files, file_1 and file_2 respectively. *Note that we provide example files that can be used: execution_example.txt and evaluation_example.txt for file_1 and file_2 respectively. These files are also shown in figure 2.*

b. Typical program execution

First of all, the format of the input files has to follow a precise template. For the sake of clarity, we will refer to the two types of files that our program uses as:

file_1, contains the data on which the interpolation or approximation is performed. For this file, the number of dimensions should be equal to two and the data should have two columns. These two columns correspond to the couple of sampling points (x_i, y_i).

file_2, contains data on which the interpolation or approximation respectively is evaluated. For file_2, the dimension should be one and the data has one column - the points at which the values of the interpolating or approximating function will be evaluated.



```
dimension,2
number_of_points,5
variables,x,y
-2.0,4.0
-1.0,2.0
-0.5,2.6875
0.75,-1.19921875
2.0,8.0
```

```
dimension,1
number_of_points,15
variables,x
-2.0
-1.5
-1.0
-0.8
-0.5
-0.2
0.0
0.3
0.6
0.9
1.0
1.3
1.6
1.8
2.4
```

Fig. 2: Example of input files for this program. Left, an example of file_1. Right, an example of file_2.

These files (file_1 and file_2) thus contain the following information: *number of dimensions, number of points, names of variables,* and *data*.

Left and right part in figure 2 show an example for file_1 and file_2 respectively. It is important that this format is respected by the user.

A typical execution of our program is shown, using file_1 and file_2 as in figure 2. If the regular execution of the program is chosen (not the tests), it will ask for two files (corresponding to file_1 and file_2). We will also be asked which method to use (Lagrange, Barycentric or Least Squares). For this example we chose Lagrange interpolation.

Note that data in file_1 was obtained by sampling a known polynomial of degree four.

The program will then output the values of the Lagrange polynomial, evaluated at points in file_2. A visual assessment of the method can be made by looking at figure 3. We see that choosing five sampling points from the known polynomial of degree four, yields a perfect interpolation of the points. When the polynomial is evaluated in multiple other points, there is a perfect match with the original polynomial.

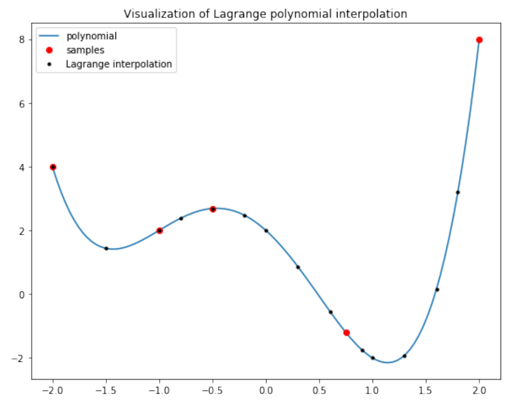


Fig. 3: Plot of the known degree four polynomial, the samples given to the method through file_1 and the points at which Lagrange polynomial was evaluated (contained in file_2).

III. Data Manipulation

Manipulation of the data is done through the class *DataHandler*. This class allows a good flow of data from the input files to the rest of the program. Public objects of *DataHandler* are the following. The constructor takes a file name as argument, several *getter* methods return values of the attributes, and a destructor takes care of the proper deallocation of the memory. Among the private objects of the class, we have a crucial method: *void read_data()*, which makes the conversion of the data written in the input file into a dynamically allocated matrix of the double type, that has number of rows as number of data points and number of columns equal to 2 for data from file_1 and 1 for data from file_2, since this method is used to handle data from file_1 as well as from file_2. This form is supported by the rest of the program.

IV. Interpolation and Approximation

After entering the name of file_1 and file_2, the user has to indicate what method to apply. Interpolation or respectively approximation of points in file_2 are then printed to the screen. Note that for the two interpolation techniques, a certain range is determined. That is, the data points in file_1 determine a range on which interpolation can be applied, and points of file_2 have to be in this range. Points outside of this range yield a Nan.

a. Lagrange Interpolation

In this class, the computation of the Lagrange interpolation is performed. This produces a polynomial of degree n (where n+1 is the number of data points in file_1) defined by [1]:

$$\Pi_n(x) = \sum_{k=0}^{n} y_k \phi_k(x) \tag{1}$$

Where,

$$\phi_k(x) = \prod_{\substack{j=0 \\ j\neq k}}^{n} \frac{x - x_j}{x_k - x_j}, \qquad k = 0, ..., n. \tag{2}$$

As can be seen in equation 1, Lagrange interpolation needs a series of points (those in file_1) to construct the polynomial, and can then be evaluated in any other point contained in the allowed interval. After initializing an object of type *Lagrange*, we make use of the public member *double Calculate_value_of_interpolant(const double point) override* to evaluate the polynomial at each point. This method is overridden from the class *Interpolation* (see figure 1) since it is one that all interpolation methods have, but with a different content, i.e. in the case of the Lagrange polynomial this method implements equation 1.

b. Barycentric Interpolation

This class implements another well known interpolation technique, Barycentric interpolation, which has the following definition [1]:

$$\Pi_n(x) = \frac{\sum_{k=0}^{n} \frac{w_k}{x - x_k} y_k}{\sum_{k=0}^{n} \frac{w_k}{x - x_k}} \tag{3}$$

$$w_k = \left(\prod_{\substack{j=0 \\ j\neq k}}^{n} (x_k - x_j) \right)^{-1}, \qquad k = 0, ..., n. \tag{4}$$

Note that equation 4 appears in equation 2, thus in equation 1. This was the principal motivation for the organisation of these two classes in a family.

As before, an object of type *Barycentric* is initialized, then we call a method *double Calculate_value_of_interpolant(const double point) override* which in the same way as Lagrange interpolation, overrides the method declared in the mother class *Interpolation* and implements equation 3.

c. Least Squares

In this class we perform data approximation rather than interpolation. That is, in addition to the data, the user has to input the degree of the polynomial that should approximate the data. The approximation is performed such that it satisfies the definition [1]:

$$\sum_{i=0}^{n}[y_i - \tilde{f}(x_i)]^2 \leq \sum_{i=0}^{n}[y_i - p_m(x_i)]^2 \tag{5}$$

where $p_m(x)$ is any polynomial of degree m, and $\tilde{f}(x)$ is the least squares approximation of the data with a polynomial of degree m.

This method works in a similar way as the two previous. After initializing an object of type *LeastSquares*, the method *double Calculate_value_at_point(const double point)* computes and returns the approximation at point *point*.

In this method, a system of linear equations is compiled to find the coefficients of the approximating polynomial. The matrix of this system is made up of nodes raised to a power from 0 to a user-defined one. The vector of the right part of the system of equations is made up of the corresponding values in the nodes. The matrix and vector are dynamic and have the corresponding types of *Matrix* [2] class of the *eigen* library. According to [1], the resulting system of linear equations can be efficiently solved by the QR factorization or by a Singular-Value Decomposition. To solve this system, the method *householderQr().solve(y)* [3] of the *eigen* library is used, where *y* is the right-hand-side of the equation to solve.

V. Program Testing

Note: Make sure that the googletest library is correctly included in the project.

Tests for this project can be found in *test.cpp* in the file *test*. There we test the different classes mentioned above.

First, we test the well behaviour of the class ***DataHandler***. As mentioned before, the form of the input file is extremely important. Besides testing the correct reading of the file, we also test the correct handling of cases where the file does not respect the format. All the input files that have a wrong format can be found in the file *test*. For example, we test that a correct exception is thrown if the number of variables does not match the dimension of the problem, or if a data value is missing.

We then test in a similar way ***Lagrange, Barycentric*** and ***LeastSquares***. For the first two classes, we assess that when the data point in which the user asks to perform the interpolation or approximation, is out of range, Nan is returned. Then, applying all three interpolation or respectively approximation techniques on the input file_1, we test a few values for which we computed the correct return by hand, following equations 1, 3 and 5. Note that we tested the same points for the three techniques. Lagrange and Barycentric interpolation yield the same results. When the degree of the polynomial is chosen to be n for Least Squares, thus the same as for the two interpolation techniques, it also returns the same results. This is an additional assessment.

Note also that we print a message saying that the memory allocated during the construction of the object of type *DataHandler* was correctly freed.

VI. Conclusion and Perspective

In conclusion, we were able to build a program that implements interpolation or approximation of data. A model is built on a first set of data, and is then applied and evaluated on an additional set of data. The three techniques that we implemented are: Langrange interpolation, Barycentric interpolation and Least Squares approximation. As can be seen in the presentation of the typical program execution, we obtained a visual assessment of our program.

Even though the obtained results are promising, we can imagine several improvements or extensions to the program. First we could implement a way to estimate the error that is done when applying the interpolation or approximation techniques to new data. This would give a concrete qualitative evaluation of the program. Then, we could improve the user-program interface. Not only could the program be more "user friendly" but it could also offer the possibility to write the output into a file instead of printing it in the terminal. In this way, the user could directly use this file for further work. Finally, we could extend our program to be able to support problems of higher dimensions.

References

[1] A. Quarteroni and F. Saleri, “Approximation of functions and data,” in *Scientific Computing with MATLAB and Octave: Second Edition*, ser. Texts in Computational Science and Engineering, A. Quarteroni and F. Saleri, Eds. Springer, pp. 71–99. [Online]. Available: https://doi.org/10.1007/3-540-32613-8_3

[2] Eigen: The matrix class. [Online]. Available: https://eigen.tuxfamily.org/dox-devel/group__TutorialMatrixClass.html

[3] Eigen: Eigen::HouseholderQR< _matrixtype > class template reference. [Online]. Available: https://eigen.tuxfamily.org/dox-devel/classEigen_1_1HouseholderQR.html