2D Interpolation in C++

Karan Hiranandani*

18-August-2022

1. Installation

(a) Installing `vcpkg`: `vcpkg` is a package downloader that installs CGAL alongwith its additional dependencies with a single command.
Clone `vcpkg` from here

```
C:\dev> git clone https://github.com/microsoft/vcpkg
C:\dev> cd vcpkg
C:\dev\vcpkg> .\bootstrap-vcpkg.bat
```

(b) Installing CGAL
Note: `vcpkg` installs 32-bit binaries by default. To develop 64-bit software, the Windows environment variable `VCPKG_DEFAULT_TRIPLET` must be set to `x64-windows` or append the suffix `:x64-windows` to the package name that must be installed.
Note: `yasm-tool` must be installed in 32-bit to correctly build `gmp` 64-bit, which is needed for CGAL.

```
C:\dev\vcpkg> ./vcpkg.exe install yasm-tool:x86-windows
C:\dev\vcpkg> ./vcpkg.exe install cgal
```

The abovementioned `bash` script installs and compiles `GMP`, `MPFR`, and several boost libraries.
Note: CGAL is a header-only library and does not contain any `lib` or `dll` files.

(c) IDE and Compiler
Visual Studio 19.0 is used as the IDE. The Intel C++ 19.1 compiler is used to compile the code.

(d) CMake

i. `CMake` is used to build the solution files. The application of `CMake` requires a `CMakeLists.txt` file, which is included in the CGAL examples. Custom programs require the user to create their own `CMakeLists.txt` file. The basic format and structure of a `CMakeLists.txt` file for CGAL applications can be found here here.

ii. Creating a base `CMakeLists.txt` file: A base can be created using the script `cgal_create_CMakeLists`. This script resides in the scripts directory of CGAL (CGAL-5.5/scripts). Executing

*Under the guidance of Jianguo Liu and leadership of Kent Harms.

this script in the directory containing the source (`.cpp`) files creates the `CMakeLists.txt` file, which contains the rules required to build the contained application(s). However, this requires the user to first set certain environment and PATH variables.

A. User variable `QTDIR`: `C:\dev\vcpkg\packages\qt5-base_x64-windows\share\cmake\Qt5`

B. User variable `PATH`: `C:\dev\vcpkg\buildtrees\cgal\src\v5.4-bee2811ad2.clean\Scripts\scripts`

C. System variable `PATH`: `C:\Program Files\CMake\bin; C:\dev\vcpkg\packages\qt5-base_x64-windows\share\cmake\Qt5`

The abovementioned environment variables also set up the package `Qt5`, which is described later.

iii. Customizing the `CMakeLists.txt` file: The script is executed by using the following command

```
sh cgal_create_cmakelists
```

The following command line options can be used to modify the configuration of the file.

A. -s source
Creates a single executable for the source file, linked with compilations of other source files. If this parameter is not specified, an executable is created for every source file.

B. -c com1:com2.....
Lists the components to which the executables must be linked. Example: -c Qt5

iv. Linking with CGAL libraries: The following commands are used to link the project with the CGAL library, thereby ensuring that the CGAL libraries can be used.

```
find_package(CGAL)
add_executable(my_executable
my_source_file.cpp)
target_link_libraries(my_executable CGAL::CGAL)
```

(e) Additional Dependencies: Some programs require additional libraries and dependencies. The two libraries used in this project include `Qt5` and Boost.

i. Boost: CGAL only requires the headers of the Boost files. It is recommended to define the environment variable `BOOST_ROOT` and set its value equal to the location of the Boost root folder (`C:Boost\boost_1_69_0`)

ii. `Qt5`: This library is essential to run the CGAL demos and basic viewers. Since `Qt5` is modular, the user can only install those modules of `Qt5` that are needed by executing the following command.

```
C:\textbackslash dev\textbackslash vcpkg>./vcpkg.exe
install cgal[qt]
```

2. **Theory**

(a) Geometric traits (Kernel)
The algorithms and data structures in the basic CGAL library are parametrized by a traits class. For most data structures and algorithms, you can use a kernel as a traits class. These kernels provide definition for basic points, segments, and different geometric operations that are applicable to most algorithms. The choice of the kernel depends on the kind of computation the user wishes to perform.
The current project uses the Exact_predicates_inexact_constructions_kernel. This provides a satisfactory balance between the accuracy and speed of the computations.

(b) Concepts and Models
A *concept* is a general set of requirements for the application of a data type. It contains certain nested types, member functions, and/or free functions that can be applied to the data type. A *model* of a concept is a specific class that fulfills the requirements listed by the concept.

(c) Triangulation
Before interpolating the desired features of a dataset, the dataset must be triangulated to compute the nearest neighbors of the query point. The basic elements of a CGAL triangulation are vertices and faces. The triangulations classes of CGAL provide high-level geometric functionalities such as location of a point in the triangulation, insertion, removal, or displacement of a point. They are build as a layer on top of a data structure called the triangulation data structure. The triangulation data structure can be thought of as a container for the faces and vertices of the triangulation. This data structure also takes care of all the combinatorial aspects of the triangulation.
The triangulation class is based on two parameters, namely

- **geometric traits**: provides the geometric primitives (points, segments, and triangles) of the triangles and the basic elementary operations.
- **triangulation data structure**: The `triangulation_data_structure_2` <Vb, Fb> is a default model of the triangulation data structure concept.The vertex and base classes are templated by the geometric traits class which enables them to obtain some knowledge of the geometric primitives of the triangulation. Those default vertex and face base classes can be replaced by user customized base classes in order, for example, to deal with additional properties attached to the vertices or faces of a triangulation.

The vertices and handles are accessed through `handles, iterators, and circulators`. They provide the derefence operators * and ->. While circulators are designed to visit circular sequences, handles are used whenever the element to be accessed is not part of a sequence. Unlike handles, the iterators and circulators are bidirectional.
There are different kinds of triangulations provided in the CGAL library–Delaunay, constrained, regular, and constrained Delaunay. Each triangulation corresponds to a different class. Every 2D triangulation is parametrized by a geometric class and triangulation data structure. There exists a unique concept `TriangulationDataStructure_2`

that is applicable to every triangulation class, certain classes have different requirements that are fulfilled by other data structures.

(d) Delaunay Triangulations
Delaunay Triangulations are represented by the class `Delaunay_Triangulation_2<Traits, Tds>`. Delaunay triangulations are preferred to regular triangulations because of the former's empty circumcircle property–no facet of the triangulation contains a data point in its interior. This ensures that the triangles are not extremely acute–thereby improving the linear interpolation. The `Delaunay_Triangulation_2<Traits, Tds>` class inherits from the basic class `Triangulation_2<Traits, Tds>`. The geometric traits class used to implement this program must be a model of the concept
`DelaunayTriangulationTraits_2`. The insertion of a vertex into the triangulation requires $\mathcal{O}(d)$, where d is the degree of the new vertex in the updated Delaunay Triangulation.

(e) Natural Neighbors
Natural neighbors of the inserted vertex are computed to gauge the influences of the known features of the neighboring coordinates. If the known points of the triangulation form a set $\mathcal{P}$, the insertion of a query point $\mathbf{x}$ into the triangulation steals components from the Voronoi cells of the neighboring points.
If $\pi(\mathbf{x})$ denotes the volume of the potential Voronoi cell of $\mathbf{x}$ and $\pi_i(\mathbf{x})$ denotes the volume of the sub-cell that would be stolen from $\mathbf{p}$, the natural neighbor coordinate of $\mathbf{x}$ is given as

$$\lambda_i(\mathbf{x}) = \frac{\pi_i(\mathbf{x})}{\pi(\mathbf{x})}$$

The implementation of the natural neighbor coordinates is performed by `CGAL::natural_neighbor_coordinates_2()`

(f) Linear Interpolation
Sibson used the linear combination of the natural neighbor weights and features of the nearest neighbors to compute the feature of the query point $\mathbf{x}$. The interpolation of $\Phi(x)$ is given as

$$Z^0(\mathbf{x}) = \sum_i \lambda_i(\mathbf{x}) z_i$$

The implementation of the linear interpolation function is performed by `CGAL::linear_interpolation()`

3. **Implementation: 2D linear interpolation of multiple query points**

(a) Dataset
The current implementation involves $60,000$ data points. For the purpose of verification, the features of the data points are defined by the function f

$$f : z(\mathbf{x}) = ax^2 + by^2 + cxy$$

where x and y represent the x-coordinate and y-coordinate of $\mathbf{x}$.
The dataset points are generated randomly using the `randint` function in MATLAB. $5,000$ query points were also generated randomly

using an identical procedure. The datasets contain the x and y coordinates. Input iterators of C++ are used to read the datasets into a vector of type `Kernel::Point_2`

```
// importing points from file
std::ifstream in("data_2/60000/triangulation_prog1_60000.cin");
std::vector<Point> pts(std::istream_iterator
<Point>(in), {});
```

(b) Delaunay Triangulation and Value Function

```
typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef K::Point_2 Point;
typedef CGAL::Delaunay_triangulation_2<K> Delaunay;
```

The points are inserted into a variable of type `Delaunay` using `Delaunay::insert(Point, Point)`.

```
Delaunay T_function;
T_function.insert(pts.begin(),pts.end());
for (int i = 0; i < pts.size(); ++i) {
        value_function_function.insert(std::make_pair(pts[i],
        (a * pts[i].x() * pts[i].x()) + (b * pts[i].y() *
        pts[i].y()) + (c * pts[i].x() * pts[i].y())));
        }
```

The above lines of code use the object `value_function_function` to make a pair between the data points and their corresponding features. This object is used to compute the natural neighbors of the query points.

(c) Natural Neighbors

`CGAL::natural_neighbor_coordinates_2()` is the CGAL function that computes the natural neighbor weights of the query point. The function has three input parameters, namely the Delaunay triangulation `Dt`, query point `p`, and a `std::back_inserter(coords)` iterator of the vector `coords`, which is of the type `std::pair<Point, Coord_type>`. The object `Coord_type` is of the type `Kernel::FT`, where `FT` is a `typedef` for `FieldNumberType`, which is model of `Kernel`. It represents the value of a geometric field object.
The function output is of type `CGAL::Triple<std::back_insert_iterator <Point_coordinate_vector>, K::FT, bool>`. The first parameter is an iterator to the vector that contains the natural neighbor coordinates of the query point, the second parameter represents the area of the Voronoi cell $\pi(\mathbf{x})$ of the query point $\mathbf{x}$, and the third parameter is a `bool` that confirms if the computation was successful.

(d) Linear Interpolation

`CGAL::linear_interpolation()` conducts the linear interpolation of the desired feature of the query point. It takes four input parameters, namely `coords_begin`, `coords_end`, `norm`, and `Value_access (value_function)`. The first two parameters represent the first and last element of the natural neighbor coordinates vector, the third parameter represents the value of the Voronoi cell of the query point, and the fourth parameter represents a functor that is a pair of a function value type and a Boolean. Additional information is provided

here. It returns the field object value, i.e., the interpolated value, of the desired feature of the query point. Thus, it's return type is `Kernel::FT`.

```
Coord_type linear_interpolation_vector(Point& p, Delaunay& Dt,
Coord_map& value_function) {
        Point_coordinate_vector coords;
        Coord_type norm = CGAL::natural_neighbor_coordinates_2(Dt, p,
                std::back_inserter(coords)).second;

        // linear interpolation function with exception block
        try {
                if (norm == 1) {
                        throw(norm);
                }
                else {
                        Coord_type result = CGAL::linear_interpolation
                        (coords.begin(), coords.end(),
                        norm, Value_access(value_function));
                        return result;
                }
        }
        catch (Coord_type norm) {
                Coord_type norm_0 = 0;
                return norm_0;
        }
}
```

The in-built function is aborted if the query point is outside the convex hull of the triangulation. To avoid that, I have placed it in an exception loop that instead returns a feature value of 0 for any point outside the convex hull.

(e) `std::transform`
The in-built linear interpolation function only takes a single query point as an input. In order to conduct linear interpolation on multiple query points, the `std::transform` function is used.

```
std::vector<Coord_type> result;
std::transform(begin(query_points), end(query_points),
back_inserter(result), boost::bind<Coord_type>(&linear_
interpolation_vector, _1, T_function, value_function_function));
```

The first and second parameters represent the first and last elements of the query points vector. The third parameter represents an iterator to the vector `results` of type `Coord_type`. Since `std::transform` takes a bounded function as its fourth parameter. The parameter to the bind function (in this case, `Point& p`) is passed as a dummy variable. The Delaunay triangulation `T_function` and value function `value_function_function` are passed to the bind function. The `boost::bind` function is used to create the bounded function. It is important to ensure that the vector types of the first and second parameter vectors are similar to that of the dummy variable.

(f) Timing
The `std::chrono` functions are used to measure the time of the function.