Motion Planning Workshop

The following attempts to provide an overview of the auxiliary code supplied for your project. The documentation is intentionally asymmetric in the sense that some peripheral parts are briefly mentioned while the core part is presented in detailed. Before reading the documentation, please be familiar with the MMS framework (see [3, 2]). In order to understand the code itself you will need to familiarize yourself with CGAL [5] and BOOST [4].

1 General Background — Cgal, Kernels and Generic Programming

The code uses two main libraries: CGAL and BOOST. CGAL is used for the geometric infrastructure and algorithms and BOOST is used for its graph library. A fundamental building block used by most of CGAL's algorithms is a *kernel* which groups together geometric objects (such as points, segments, lines etc.) and predicates (such as intersections). The kernels makes use of *number types* which may be the machine built-in fixed-precision number types (such as `int` or `double`) but may be more advanced number types that allow *exact* representation of numbers which is needed for many algorithms. In order to allow separation between the types used and the algorithms implemented Cgal follows the *generic programing paradigm* [6]. In practice this means that most of the code is templated with a kernel as the templated argument. This allows the use of different kernels with the same code as long the kernel adheres to certain properties.

The code supplied to you is written with the same considerations in mind. Thus almost all of the classes are templated. The template argument is usually a kernel that is required to support *exact rational arithmetic*. In some cases the number types needed are *algebraic numbers*. For these cases we also need a way to convert between rational and algebraic numbers.

To simplify the use of the code, we provide an example where the templated arguments are given as default parameters. We recommend using these parameters as they follow the minimal requirements needed while providing the best performance.

2 General Contents

The code is subdivided into several folders, each attempting to group common parts of the infrastructure together:

- **Project**: Includes general files required for the project. There are general include files used by all components (`Includes.h`), global constants defined (`Globals.h`), compilation flags (`CompilationFlags.h`, general typedefs (`CgalTypedefs.h`) and a class that reads and stores a configuration file (`Configuration.h` and `Configuration.cpp`). There is no need to change these files.
- **Programs** Includes a simple program that calls an example of the MMS framework (we will cover this example thoroughly). The program is declared in `Path planner.h` and implemented in `Path planner.cpp`. The file `Path planner.cpp` also includes a small function `display_scene` that may be used to visualize a scene with a source configuration and one target configuration.
- **Input** Input configuration files and scenarios are placed in this folder. We supply two scenes consisting of a workspace, a robot a source configuration and one target configuration. Each folder includes a configuration file for loading the scene and solving it.
- **Utils** Includes many utilities used by the MMS framework. The utilities include: (i) geometric utilities (such as bounding volumes and point comparison), (ii) Interval utilities (such as intervals and interval sets), (iii) Number type utilities (such as conversions between algebraic and rational numbers and the approximation of square root numbers), (iv) Polygon utilities (such as intersection predicates, translations and rotations of polygons and more), (v) Utilities supporting random generation of numbers and geometric objects, (vi) Rotation utilities (such as representing a rational rotation and converting

between angles and rotations), (vii) UI utils (a temporary GUI, a final more sophisticated GUI will be supplied). We will review several of these utils in the section 3.

- **Graph** A wrapper class around the graph library provided by Boost. Includes self implementation of connected components in `ConnectedComponents.h` and `ConnectedComponents.cpp`.
- **Configuration spaces** Includes the means to construct two types of configuration spaces: (i) AnglePrimitive — the construction of the two-dimensional configuration space representing translational movement only. (ii) PointPrimitive — the construction of the one-dimensional configuration space representing rotational movement only. We will review their interface in section 4.
- **Manifolds** Includes a unified interface for the decomposition of configuration spaces into FSCs. We will review this part in section 5.
- **Path planning** Includes local planners in different FSCs. We will review this part in section 6.

3 Utils

We will overview the class `Interval` located in the Interval_utils sub-folder, the main polygon utilities, the class `Rotation` located in the Rotation_utils sub-folder and the class `Reference_point` located in the main utilities directory. All other classes are either self explanatory or of less importance and may be overlooked.

3.1 `Interval` class

The class `Interval` represents an interval (not necessarily bounded). It is templated with two parameters: the first is the number type used for the boundary representation and the second a converter to a possibly different number type. The interface is straightforward except for the predicates `contains`. There are two implementations of contains predicate: the first operates on the number type used by the `Interval` class while the second operates on approximations of the number types as provided by the converter. This is done as the approximated operations are less computationally consuming.

We will use this class as a way to store FSCs for the point primitive (rotational movement only). $\mathcal{C}_{\text{free}}$ decomposes in this case into intervals whose endpoints are algebraic numbers. As their representation is complex, we will want to use rational numbers when possible hence the two number types and the need to convert between the two.

3.2 Polygon utils

The two main classes that are in this subdirectory are `Extended_polygon` and `Smart_polygon_with_holes` which both extend Cgal's `Polygon_2` class.

The class `Extended_polygon` stores a polygon and allows efficient translation and rotation of the polygon by additionally storing its reference point. A movement of the polygon is simply an update of the reference point. Only when an explicit representation of the polygon is required the vertices' coordinates are computed.

The class `Smart_polygon_with_holes` stores a polygon and allows queries on the polygon such as its area or number of vertices. Once such a query has been performed, the result is stored so that the next query will not have to perform the same computation over again but return the stored value.

3.3 `Rotation` class

As we need to exactly rotate coordinates we represent a rotation as a pair of rational number representing the sine and cosine of an angle. This representation (implemented in the class `Rotation`) allows for consistent rotations of points, adding rotations exactly and exact comparison. The implementation is quite trivial and simply stores the sine and cosine of an angle.

What is slightly more interesting is the conversion between an angle to a rotation instance . The angle 45° has sine and cosine values of $\frac{\sqrt{2}}{2}$ which cannot be represented by rational numbers. We used the method by Canny et al. [1] to approximate rotations. This is out of the scope of the project but is an interesting method on its own.

3.4 `Reference_point` class

A robot reference point is defined by its location and orientation. This is simply implemented by storing a point (for the location) and a rotation (for the orientation). The class is trivial but will be used again and again thus mentioned here.

4 configuration spaces

4.1 Angle Primitive

As discussed in class the means to compute the configuration space of a translating robot is by computing the Minkowski sum of all the obstacles with the robot reflected about the origin. The is implemented in a straightforward manner in the class `FixedRotation::Configuration_space`. The class stores two internal data structures: The first is CGAL's `CGAL::Polygon_set_2` class which allows efficient operations on sets of polygons such as union, intersection and point location. The second is a vector of `Smart_polygon_with_holes` which stores each of the cells represented in the polygon set. This redundant storage of the cells is done intentionally to allow quick access to either a general cell (for point location queries) and a specific cell for queries supported by the `Smart_polygon_with_holes` class.

The interface of the class allows to decompose an entire configuration space into free cells via the function `decompose` or adding an obstacle to an already decomposed configuration space via the function `add_obstacle` for a workspace obstacle or via the function `add_c_space_obstacle` for an obstacle in the configuration space.

Each cell has an internal id and can be accessed by supplying the id. The only cell that cannot be accessed is the unbounded cell which represents the world out of the bounding box of the workspace. A cell's id may be retrieved by providing a point in that cell via the function `get_fsc_id`.

4.2 Point Primitive

The configuration space of a robot rotating about a fixed point is computed by considering the set of *critical angles* — the angles where a feature (edge or vertex) of the robot is contact with a feature (vertex or edge) of an obstacle. All the critical angles are inserted into a one dimensional array (this is a complex way of saying a set of intervals along a line). The intervals along this line are cells completely in $\mathcal{C}_{\text{free}}$ or $\mathcal{C}_{\text{forb}}$. The only non-straightforward part of this process is computing the critical angles, this is described in [2] but what should be stated is that the angles are not computed explicitly but only via a parametrization: $t = tan(\frac{\theta}{2})$. Thus the line describing this configuration space spans from $-\infty$ to $+\infty$.

The file `Fixed_point_utils.h` provides conversions between angles and their parametrization. The file `ConfigurationSpace.h` provides the interface to computing the actual one-dimensional configuration space. There are two internal files `CriticalValues.h` and `CriticalValuesConstructor.h` that may be overlooked. The file `ConfigurationSpace.h` contains the class `Configuration_space` which we will briefly overview.

The class provides the ability to consider either the complete range of angles (values of t from $-\infty$ to $+\infty$.) or a subset of these angles (values of t bounded between two restricting values t_a and t_b). This gives the ability to consider only a restricted part of the one dimensional configuration space which can speed up the computation time. This restriction of angles is called the *region of interest* and is implemented by a class named `Rotation_range` supplied in the utils folder.

The class is templated with a kernel `K` for the use of rational numbers, a kernel `AK` for the use of algebraic numbers and a class `AK_conversions` that is used to convert between the numbers. This is needed as the parametrized critical angles are algebraic numbers. The main utilities provided by the class are the `decompose` function which decomposes the configuration space of a robot fixed at a given position in a workspace cluttered with obstacles into cells. These cells are completely in $\mathcal{C}_{\text{free}}$ or $\mathcal{C}_{\text{forb}}$ but when they are created we lack the knowledge to determine if such is cell is free or forbidden. Thus the class has an additional function `free_space_location_hint` which allows a user to supply the class with the information that a specific configuration is free. We will see how this is used when we cover the example supplied. Additionally the class allows to add an obstacle to an already decomposed configuration space via the function `add_obstacle`.

Retrieving an interval is somewhat more complex than what would have been expected. There reason is that the first interval (from $-\infty$ to the first critical angle t_0) and the last interval (from the last critical angle t_k to $+\infty$) are actually connected. Thus when a user requests an interval there are three options of the return type: (i) a simple interval $[t_a \ldots t_b]$ (ii) two intervals in the form $[-\infty \ldots t_0]$ and $[t_k \ldots \infty]$ (iii) one complete interval (relating to the fact that the robot has no obstacles blocking his rotational motion) $[-\infty \ldots +\infty]$ As a result, there are two (small) technicalities: the first is that there is an internal id for every interval and this is simple the interval's index in a vector storing all critical angles. The external id (what the user knows) is not related and there is an internal mapping between the internal and external ids. The second technicality is that the the function get interval constructs the requested interval in an `Interval_set` (a class that is declared in the utils section) which may be a single interval but can also cover the other cases mentioned.

5 Manifolds

The folder Manifolds contains the infrastructure needed to represents manifolds in a unified manner. A manifold is defined via a constraint on the configuration space (and a possible region of interest to restrict the part of the configuration space where the manifold is constructed) and the structure of an FSC in the decomposed manifold. The manifold serves as a *container* of such cells. Additionally the MMS algorithm requires a container of manifolds. These four structures are virtually implemented by four base classes in the folder Manifolds
Base.

The folder Manifolds
FixedAngle contains the derived classes appropriate for the representation of a robot translating without rotating. The folder Manifolds
FixedPoint contains the derived classes appropriate for the representation of a robot translating without rotating.

There are two additional files that are important in the context of manifolds: (i) `FSC_indx` — we wish to store a representation of an FSC in the connectivity graph. The way we do this is by defining an index to represent each FSC. The index stores the FSC type (`FIXED_ANGLE` or `FIXED_POINT`, the manifold id in the appropriate manifold container and the FSC id in the appropriate manifold). (ii) `Intersect manifolds` —Given two manifolds, compute their intersection (if exists) and returns the ids of the FSCs that intersect.

6 Path Planning

After the connectivity graph has been constructed, one can query it for paths in $\mathcal{C}_{\text{free}}$. In order to determine if a path exists one simply needs to check if the node of the FSC containing the source and the node of the FSC containing the target are in the same connected component. In order to construct an actual path one needs to plan a path within each FSC. Planning within cells is implemented in this folder.

Planning a path within an interval is implemented in the file `IntervalPathPlanning.h` and planning a path within a polygon (using the visibility graph technique presented in class) is implemented in the file `PolygonPathPlanning.h`.

7 Example code — putting it all together

The example we supply consists of two parts, the first is a planner using all the building block supplied. The second is a program that calls the planner for a specific scenario.

The planner is implemented in the class `Mms_path_planner_example` which is located in the file `MMS_example.h`. This class contains many small details that you should familiarize yourself with. This is a naïve implementation of the MMS algorithm, your planner may reuse the code in this class in order to construct your best algorithm.

The main internal variables stored by the class are the workspace, the robot, the connectivity graph and the manifolds (`Layers` — manifolds representing translations and `C_space_lines` — manifolds representing rotations). The main functions that the class provides the user are `preprocess` and `query` which we will cover:

`preprocess` The function constructs a set of angles that will serve as the constraints defining the layers. This is done in the function `generate_rotations`, the result is stored in the member variable `_rotations`. Next, each such rotation is used to construct a layer using the private function `add_layer`. Afterwards the lines are generated using the private function `generate_connectors`.

Generating a layer is straightforward and we omit the details. Generating a vertical line in the configuration space is slightly more complex: Generating a connector consists of the following parts: (i) Choosing a free configuration on a layer that will define the fixed position of the robot. (ii) Defining a region of interest used — this is done via a heuristic considering the cell of the layer from the line is constructed. If the cell has a small area (according to pre-defined constants) then a small region of interest will be used. If the cell has a large area, then the whole one-dimensional configuration space will be constructed. (iii) Possibly, choosing not to construct the configuration space. This is termed filtering and is done by considering which cells of layers *may* intersect the line. If all of them are in the same connected component in the connectivity graph, then the contribution of the connector is limited and it is not constructed. (iv) The connector is constructed. (v) The connectivity graph is updated.

`query` — The function receives two reference points of the robot that represent a source and target configurations. It attempts to construct a sequence of configurations in $\mathcal{C}_{\text{free}}$ connecting the source to the target. Initially the source and target configurations are connected to the connectivity graph by constructing the point primitive. If they fail to connect to the graph then no path is returned. If they do not lie in the same connected component of the graph then no path is returned.

If a path in the connectivity graph is found then a path in $\mathcal{C}_{\text{free}}$ exists. As the path that was found in the connectivity graph is a list of `FSC_ indx`'s that correspond to a list of intersecting FSCs, a path in each such FSC needs to be constructed. This is done by converting each `FSC_indx` into a unified representation of an FSC (implemented by a class FSC in the file `FSC.h`) and planning a path within each such FSC (implemented in the file `Fsc_path_planning.h`).

References

[1] John Canny, Bruce Donald, and Eugene K. Ressler. A rational rotation method for robust geometric algorithms. In *Symposium on Computational Geometry*, pages 251–260, New York, NY, USA, 1992. ACM.

[2] Oren Salzman, Michael Hemmer, and Dan Halperin. On the power of manifold samples in exploring configuration spaces and the dimensionality of narrow passages. *CoRR*, submit/0423333, 2012.

[3] Oren Salzman, Michael Hemmer, Barak Raveh, and Dan Halperin. Motion planning via manifold samples. In *ESA*, pages 493–505, 2011.

[4] Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual.* Addison-Wesley Professional, 2001.

[5] The CGAL Project. *CGAL User and Reference Manual.* CGAL Editorial Board, 4.0 edition, 2012. http //www.cgal.org/.