ILQGames Documentation

David Fridovich-Keil

August 2020

Contents

1 Introduction . . . **2**
1.1 Paper and Other Resources . . . 2
1.2 Language . . . 2
1.3 Testing . . . 2
1.4 Style . . . 2

2 Installation . . . **4**
2.1 Dependencies . . . 4
2.2 Building the workspace . . . 4
2.3 Docker . . . 4
2.4 How to use this code base in another project . . . 5
2.5 Running Tests . . . 5
2.6 Running an Executable . . . 5

3 Organization . . . **6**
3.1 Dynamics . . . 6
3.2 Costs . . . 7
3.3 Constraints . . . 7
3.4 Solvers . . . 7
3.5 Problems . . . 8
3.6 Receding Horizon Operation . . . 8

4 Example . . . **9**
4.1 Getting Started . . . 9
4.2 Anatomy of an Example . . . 9
4.3 Step 1: Setting the cost weights . . . 10
4.4 Step 2: Setting the nominal speed for each car . . . 10
4.5 Step 3: Adding input constraints . . . 10
4.6 GUI . . . 10

1 Introduction

This is the official ILQGames documentation. Please note that each push to the `master` branch of this repository triggers an automatic build of class- and function-level documentation, which may be found here. Other documentation is available in the repository's main website. This manual includes all the information you'll need to download, build, use, and improve the ILQGames code base. Most of the rest of this manual is structured as a quick start guide, with step-by-step instructions to follow.

1.1 Paper and Other Resources

For a more complete description of the algorithmic details of this project, please refer to the paper. The underlying LQ game solutions can be found in the wonderful book by Başar and Olsder. I have also provided more in-depth derivations of both feedback and open-loop (which this library largely ignores) Nash equilibria of LQ games here.

1.2 Language

This code base is almost entirely written in C++, and anything not in C++ should be treated as either deprecated or temporary. If you are not familiar with C++, there are lots of great online resources for learning the basics and it might be a good idea to consult with them before attempting to add new functionality. A Julia implementation is also available here, although we do not attempt to keep the two in sync.

1.3 Testing

Testing is an important piece of any large engineering project, hardware or software. Here, the `test/` directory contains a number of unit tests to audit the functionality of individual classes and functions. The ILQGames project uses `gtest` (Google's C++ unit testing framework) to specify, accumulate, and run tests. From the `build/` directory (after installation), one can run tests by typing

```
    ./run_tests
```

1.4 Style

As with any large programming project, style is very important. I'll give a list of general style pointers here and throughout this manual, but I strongly encourage you to take a look at the Google C++ style guide if you have specific questions. Also, I strongly suggest that you use a code formatter like clang-format to automatically enforce style standards (at least, to a large degree).

- Use *inheritance* and *encapsulation* whenever possible. Inheritance is a good idea especially for classes you might need multiple versions of. For example, it makes sense for different dynamics models (each of which is its own class) to inherit from a single parent class that specifies a single unified interface. Encapsulation is the other key to effective abstraction, since it lets users of a particular class not have to worry about how particular functions are implemented or data are stored.

- *Modularity* is another key to effective software design. If you have a big complicated class or even just a function with lots of moving parts, chances are you can make your (and everyone else's) life easier by breaking it up into multiple different pieces that do one or maybe two things, and do them reliably.

- *Unit testing* is another important way to ensure that modules behave reliably in a large project. The main idea behind writing good unit tests is to test the key functionality of each class in isolation to make sure it behaves as expected.
- Comments are what let yourself and other users figure out what's going on without having to parse the code itself. Comment early and comment often.
- Classes, functions, and variables should all be named clearly and consistently. For example, throughout this repository class and function names are `CamelCased`, but variable names have `under_scores`. In C++, private member variables typically end in an underscore, while public ones do not.
- It's good to get in the habit of using the `const` specifier whenever possible, both for variables and for a class' member functions. If a value is known at compile-time, it is a good idea to use the specifier `constexpr`.
- When developing large projects, it is helpful to minimize the functions and classes visible to clients (anyone outside the file you're working on). There are a couple ways to do this:
  1. Make member variables and functions private. This keeps them from being accessed by instances of other classes, but they still do clutter the namespace of their class.
  2. Rather than make a function a private class member, you can make it a member of an anonymous namespace in the same file, which means that no other file can access it.
  3. Alternatively, you can write the function as a `lambda` function, or *anonymous* function, inline. This gets compiled very quickly so the runtime is essentially identical to the previous two methods, but it may require less of a boilerplate than either of the other approaches.

2 Installation

This section provides step-by-step instructions for installing the code base on your own computer. I have tested it on both Mac OS {Mojave, Catalina} and Ubuntu {14, 16, 18}.04. The repository also provides a docker container with everything pre-installed.

2.1 Dependencies

If you are running Ubuntu (not sure which versions), you may notice a linker error related to aclocal-XX, which may be fixed by symlinking Ubuntu's native aclocal to the desired executable aclocal-XX. Otherwise, external dependencies are standard:

- glog (Google's logging tools)
- gflags (Google's command line flag tools)
- opengl, glut (graphics tools, probably already on your machine)
- eigen3 (linear algebra library)

2.2 Building the workspace

`ilqgames` uses the CMake build system. You can build the workspace as you would any other CMake project: navigate to the top-level directory (in this case, the `ilqgames/` directory) and type in the command

```
mkdir build
```

Now, navigate to the `build/` directory and build the project as follows:

```
cd build
cmake ..
make -j4
```

If you get any errors that say things like `Could not find <X>` then you'll know you need to install dependency `X`. If the compiler runs out of memory (might happen on an under-provisioned VM) or otherwise fails, it is a good idea to retry the build with a single worker, i.e.

```
make -j1
```

2.3 Docker

Once you've cloned the workspace, instead of installing the dependencies and building the workspace directly (or in addition), you can build a docker image and work there, although it won't have the nice GUI. This can be a nice clean way to ensure repeatability despite the dependencies.

To build a docker image, first install the docker desktop client online at the link above. Then, navigate to the `docker/` directory and run this command:

```
docker build -t ilqgames .
```

To run the image, just execute (from anywhere):

```
docker run -it ilqgames /bin/bash
```

This should give you a terminal prompt in the container for this project, which is isolated from your main machine. That way, anything you do won't mess up your main computer—sort of like running a virtual machine, but way faster. All the code is ready and compiled for you, so the main utility of using a docker container like this is if you have another project that depends on this one and you wish to run that in docker, or if you just want to edit and compile new code without installing dependencies.

2.4 How to use this code base in another project

There are several ways to use this code in another project. First, you could install it in a location searchable by other projects—to do this, run the following command (still within `build/`)

```
make install
```

which may need administrator privileges. Alternatively, you can maintain a duplicate copy of this repository in your other project as an external dependency. In fact, that's what this repository does for the DearImGui library which is used for creating the GUI.

2.5 Running Tests

As above, tests are an important part of any large engineering project. Here, to run tests (again, from `build/` please type:

```
./run_tests
```

Constructing good tests is a full-time job and one which is very hard to do right and impossible to do enough of. However, aside from the obvious benefits, one other advantage of tests is that they show how some functionality in the code is intended to be used. If you would like to find out more about a certain class, for example, consulting the tests for that class is a very good idea. Some classes and functions do not have tests—which is partially an oversight and partially because it is tough to construct a meaningful test without involving many other classes or functions. If you find testing coverage gaps, please do fix them!

2.6 Running an Executable

To run any executable (other than tests), execute the following from the `bin` directory:

```
./[name-of-executable]
```

For help, please use the command-line flag `--help`, which will display the name and description of all command-line arguments this file accepts, an the individual files (compiled into this one) which declare them. Many of the arguments do not pertain to this project and are generic for any executable which uses this command-line flag system; pay particular attention to the flags from this executable which should be displayed near the top.

3 Organization

The repository is organized around three components: dynamics, costs, and solution methods. Additionally, the repository provides a number of utilities and examples. We will walk through a custom example later in this tutorial. These major components are listed below:

```
ilqgames
  LICENSE
  README.md
  include/       # Headers.
     ilqgames/
        constraint/     # Constraints on state and input.
        cost/           # Cost structure.
        dynamics/       # Dynamical system structure.
        examples/       # Sample dynamic game problems.
        geometry/       # Planar geometry primitives.
        gui/            # GUI utilities.
        solver/         # Iterative and low-level solvers.
        utils/          # Other utilities.
  src/           # Source implementations.
  exec/          # Sample source code that compiles to executables.
  bin/           # Compiled executables.
  build/         # Workspace for compilation and installation.
  derivations/   # PDF derivations of Nash equilibria in LQ games.
  logs/          # Saved logs.
```

Most of these are self-explanatory. However, it is worth noting that most usage will likely involve modifying/adding code in two places:

1. New scenarios will be declared in the `include/ilqgames/examples/` directory and defined in the `src/` directory. Note that, as discussed later in this tutorial, new examples should derive (inherit) from the `Problem` class and, if they are intended to be rendered in the GUI, the `TopDownRenderableProblem` class (or you can build another renderable class here just so long as you figure out how you want to render trajectories of the game).
2. Any new scenario, or in `ilqgames` terminology "example" or "problem," will be compiled and run in an executable. Please find existing templates in the `exec/` directory.

Otherwise, what follows is a short summary of the main ideas behind the code.

3.1 Dynamics

Dynamics are all declared within the `include/ilqgames/dynamics/` directory and they follow a nested inheritance structure. The two main types of systems represented here are single- and multi-player systems, although there are also some feedback linearizable systems (labelled "flat" for "differentially flat"). These feedback linearizable systems offer a computational speedup in some cases; for further details please refer to the paper. This tutorial will essentially ignore them, however, since the changes they require are comparatively minimal.

Regardless, the major functions of any dynamical system class are to integrate to trajectories and to differentiate in time (and subsequently discretize in time). To help the integration stay as numerically-precise as possible, all the dynamics are expressed natively in continuous-time.

Further, a large class of multi-player dynamics are "concatenated," i.e., they are merely collections of decoupled single-player systems. There is a special class for these types of systems, the `ConcatenatedDynamicalSystem` class.

Finally, all single-player systems define—by convention—public constants for the number of x and u dimensions and the index of each one. These constants use the Google C++ style guide standard naming scheme for static constants, e.g., the number of state dimensions is always `kNumXDims`.

3.2 Costs

Costs are declared within the `include/ilqgames/cost` directory. These are primarily organized around the `PlayerCost` class, in which each player holds pointers to all of its instantaneous costs. By default, all of these costs are time-additive, however an experimental feature includes extrema-over-time as well. The `PlayerCost` class also holds both state and input constraints, although these should also be considered experimental. Further details are provided below.

The major workhorse from which all instantaneous costs (and constraints, see below) derive is the `Cost` abstract class. `Cost` instances can both evaluate to a real number and populate both gradients and Hessians. Importantly, these computations are all analytic. That way, only the nonzero entries need be considered (which saves a significant amount of time).

Relatedly, the `PlayerCost` can be instantiated with both state and input regularization, which essentially adds a positive diagonal matrix to the appropriate cost Hessian for that player. This can serve to improve the conditioning and numerical performance of the problem, though as usual over-regularization can lead to poor results.

Also, note that most costs are time-invariant and hence derive from the `TimeInvariantCost`, which itself derives from `Cost`. This way, time-invariant costs do not need to depend upon time (even though, by default, cost functions can and do).

3.3 Constraints

As above, the `Constraint` class derives from the `Cost` class, and like cost functions, constraints can be time-invariant. Constraint-handling is an experimental feature, and at the moment constraints are imposed using logarithmic barriers. The `Evaluate` and `Quadraticize` methods of the `Constraint` class refer to the logarithmic barrier of the constraint. The constraint also provides a method for testing the validity of any given point.

3.4 Solvers

The `ilqgames` methodology relies upon an iterative algorithm whose inner loop finds Nash equilibria of LQ games. High-level iterative solvers derive from the `GameSolver` class. The `LQOpenLoopSolver` and `LQFeedbackSolver` classes are low-level solvers that find Nash equilibria of LQ games in both open-loop and feedback information structures; both are compatible with the higher-level iterative methods. Both low-level algorithms use variable names to be consistent with Chapter 6 of Başar and Olsder's "Dynamic Noncooperative Game Theory," and slightly more in-depth derivations are provided in the `derivations/` directory.

The main loop of the iterative method proceeds as follows:

1. Integrate the dynamics forward in time according to the current strategy iterates for all players.
2. Linearize the dynamics along that trajectory.
3. Find a quadratic cost approximation for all players along the same trajectory.
4. Solve the resulting LQ game to a Nash equilibrium.

5. Adjust the current strategy iterates “in the direction of” these Nash strategies (i.e., does a linesearch, more details for which can be found in the original paper and which optionally respects constraints).

6. Repeat until convergence.

Finally, the struct `SolverParams` contains parameters (with descriptions and default settings) for the iterative solver. These are set in each executable, and optionally some can be set on the command-line using the `gflags` library. You can find more details on this later in the tutorial when we discuss a bare-bones example.

3.5 Problems

All scenarios (or “examples” in the `ilqgames` parlance) derive from the `Problem` class, which is an abstract class that essentially pushes all the dynamics, cost, and solver instantiations into derived derived classes. Specifically, derived classes only have to specify these things in the constructor (i.e., define member variables already declared in the base class), and for those which are “top-down renderable” (i.e., compatible with the built-in GUI), they must derive instead from the `TopDownRenderableProblem` class (a child class of `Problem`) and define functions to unpack a state x into each player’s position and heading.

The base `Problem` class wraps solver calls in the `Solve` method and sets up new receding horizon warm-started invocations in the `SetUpNextRecedingHorizon` method. Please note that this receding horizon warm-starting is merely a heuristic and you should feel free to change it if you choose. Further details of receding horizon operation are below.

3.6 Receding Horizon Operation

Besides instantiating each new `Problem` in a receding horizon setting (discussed above), it is also to splice each new solution together with the existing trajectory. This is accomplished by the `SolutionSplicer`. Throughout this process, the overarching heuristic idea is to keep the trajectory of the ego agent continuous (under the theory that continuous trajectories are easier to follow and that the only autonomous agent here is the ego), and let the other agents have discontinuous trajectories.

The repository also provides a utility for simulating receding horizon operation: the aptly-named `RecedingHorizonSimulator` class. The constants here (simulating time delays, clocks, etc.) can be changed, but the purpose of these simulations is to emulate the conditions the solver will face in real-time operation. That is, in real operation during the time a solver takes to produce a solution, the state will continue to evolve according to the current feedback law and dynamics.

Finally, it is important to mention the difficulty in generating constraint-satisfying warm-starts in this context. Since the constraints can and will change between solver invocations (e.g., if they correspond to collision-avoidance), analytic instantiation—for example, by integrating the trajectory forward in time while applying no input—may violate constraints. Currently, we address this challenge by augmenting each constraint (already enforced with a logarithmic barrier) with a second barrier which is always finite even when the constraint is violated. That way, when the constraint is violated, the solver can use the secondary barrier instead of the logarithmic barrier to (hopefully) push the trajectory back into the feasible set after a few iterations. This is, admittedly, not the ideal way to handle this sort of issue; refinement of this functionality is certainly warranted.

4 Example

The remainder of this tutorial walks through the construction of a skeleton example in which two cars will emulate the challenge of navigating an intersection where one car wants to go straight and the other wants to turn. Please find the header and implementation for this `SkeletonExample` at the following locations respectively: `include/ilqgames/example/skeleton_example.h`, and `src/skeleton_example.cpp`. We will focus on the implementation file for now, since the header ought to be fairly self-explanatory. Also please note that the code itself is fairly well documented and is intended to be readable on its own, separate from this document.

4.1 Getting Started

Begin by opening the file `src/skeleton_example.cpp`. You will notice a file-level comment near the top in which the following steps are outlined:

1. Set the cost weights.
2. Set the nominal speed for each car.
3. Add input constraints.
4. (Advanced) Try other constraints.
5. (Advanced) Add in a third player.

We will walk through the first three steps and leave the last two for you to do on your own.

4.2 Anatomy of an Example

First, an overview of example `Problem` class implementations. Examples inherit several important functions from the base `Problem` class, such that a derived class need only implement a constructor. Optionally, if an example derives from the `TopDownRenderableProblem` class (as this one does), it also needs to implement functions unpacking the game state x into individual players' position and heading.

In any case, the first block of code (apart from comments and includes) is always a list of parameters in an anonymous namespace. These specify things like each player's initial condition, cost weights, etc., and they live in an anonymous namespace in order to avoid polluting the main `ilqgames` namespace shared by all files in this project. The next block of code is the constructor. This always includes four major components:

- **Dynamics.** It usually begins by instantiating the dynamics of the game. There are a number of sample implementations of different dynamical systems. Please refer to Section 3.1 for further information.
- **Initial condition.** It is also important to specify the initial condition of the game, i.e. $x(0)$ (by default, we assume that games start at time $t = 0\,\mathrm{s}$).
- **Initial operating point and strategy.** The iterative solver requires a starting point, i.e., an initial operating point and strategy. The `OperatingPoint` is a state and control trajectory for the game, and each player's `Strategy` is a time-varying affine feedback law. Both are typically initialized to zero, so that the first iteration of the solver just integrates the initial state forward in time with no control inputs. Other initializations are certainly possible, though. For example, in some cases it can be useful to initialize the position for each player to follow the center of its lane (even though this may not be dynamically feasible).

- **Cost structure.** Arguably the most important part of the constructor is the part where it specifies the cost (and sometimes, constraint) structure of the game. This is usually what couples players together, so it is very important to get this part right and often the behavior of the solver can be sensitive to changes in the weights of some of these costs, and to whether behaviors are expressed as costs or constraints. This sensitivity is, of course, undesirable, but unfortunately the repository is not mature enough yet to handle every case reliably.

4.3 Step 1: Setting the cost weights

The first step in this skeleton example is to check the existing cost weights and modify them. To find them, search for `CostWeight`. A good rule of thumb for setting these should be to keep in mind the relative magnitude of different costs and make sure that the weights reflect how important you want them to be. For example, the steering input is in $\mathrm{rad\,s^{-1}}$ and should pretty much never exceed a magnitude of 1.5. Compare that with the nominal speed, which is in $\mathrm{m\,s^{-1}}$ and from which our deviation could very well be 10. Clearly, to have a smooth solution, the weight of the steering cost should exceed that of the nominal speed cost.

Further, it is important to remember that costs are time-additive. Some costs, like the proximity cost, are only active for a short time, whereas others like the lane cost are active the entire trajectory. Hence, the weights for the sparser costs should usually be higher to have much effect.

4.4 Step 2: Setting the nominal speed for each car

In addition to modifying cost weights, you can modify other parameters like the nominal speed for each car. This particular parameter governs (per the name) the desired speed of each vehicle. To give one car a "lead foot," just increase its nominal speed. You can also raise or lower this cost weight to make that player "care" more or less about going that particular speed. You can also change the cost weight on acceleration to see how this factors in.

4.5 Step 3: Adding input constraints

Constraints are the least mature part of this project; hence it is not recommended to use them without really understanding what's going on and when things might go wrong. For a first exposure to constraints, we will add interval constraints on the inputs for each player. As mentioned in Section 3.3, these are implemented as logarithmic barriers, though the adaptive scaling of these barriers is still somewhat primitive. To add in these constraints, please uncomment the relevant block of code (which can be found by searching for `constraint`).

4.6 GUI

Each example is accompanied by another file that defines the function `main` and gets compiled into an executable. These live in the `exec/` directory and the one for this example is located at `exec/skeleton_example/main.cpp`. Upon opening this file, you will notice that, apart from a bunch of boilerplate GUI handling, the only things going on are

1. setting solver parameters (some of which are factored out into command-line arguments using the wonderful gflags library),
2. creating an instance of the desired example,
3. solving the example, and
4. optionally saving the results to disk.