# fmrbenchmark

***Release 0.0.0***

April 13, 2016

This is the User's Guide to the fmrbenchmark repository, which is part of a project to develop benchmark problems for research in so-called "formal methods for robotics." This effort is stimulated by competitions, and the main website is http://fmrchallenge.org. The two other major forms of documentation are the API manual and the benchmark specifications. Among the latter documents are competition rules. Besides these sources of documentation, there are comments in the code as well as README and similar files throughout the repository.

For newcomers, a good place to begin is the Introduction.

CHAPTER 1

# Installation

There is no generic installation yet. Instructions for particular problem domains can be found in the appropriate `README` files in the repository or on the pages of User's Guide dedicated to each. E.g., try the Problem domain: Scaling chains of integrators.

CHAPTER 2

# Introduction

This page provides orientation and an overall introduction to the repository. It is a good place to begin before studying a particular benchmark. There are two founding ambitions of the project: to develop benchmark problems for research in so-called "formal methods for robotics," and to create standard interfaces, formats, etc. for expressing problems and using tools that implement methods described in the research literature. Our effort is analogous to that of SMT-LIB, which is for research in satisfiability modulo theories.

There are four major kinds of entities in the repository:

1. **benchmarks**;
2. **analysis tools** for reviewing results from using benchmarks;
3. **examples** demonstrating components of benchmarks and solution controllers;
4. **documentation**.

Spanning all four of the above kinds is the *supporting infrastructure*. This refers to header files, message formats, etc. that may be used by more than one benchmark and that may be of independent interest, besides benchmarking.

The repository as a whole has a single version number. Depending on the eventual pace of growth and styles of usage, we may begin to version significant components separately. In any case, version numbers are of the form `M.m.u`, and changes only to `u` are not expected to break any current usage.

**Warning:** The interfaces to command-line tools, the names of important ROS topics, and other user-level aspects of the repository may change with little warning until version 0.1.0. Beginning at that time, care will be taken to ensure backwards-compatibility for and to have more gradual deprecation.

## 2.1 Formulation

A normative description of benchmarks as well as a development of notation and problem formulation is given in the Challenge Document. Below is a summary.

Benchmarks are organized into *problem domains* (sometimes also called "problem settings"), which are defined in terms of several parameters. A *problem variant* refines the domain by constraining possible values that may be assigned to a parameter, e.g., deciding that time can only be a multiple of a constant (the period). Finally, a *problem instance* is defined as a particular selection of values consistent with a problem variant. The instance is the thing that is actually to be solved. A special case of this taxonomy is a concrete benchmark from industry that is to be solved as given, i.e., there is no need to provide more details like how time progresses or what the initial state can be. In such a case, the problem domain, variant, and instance are all the same.

## 2.2 Support for platforms and programming languanges

While it may be possible to build the benchmarks and infrastructure on other platforms, the current target is Ubuntu 14.04 running Linux x86_64 and the following:

- ROS Indigo
- Gazebo as used with ROS

The benchmarks are primarily implemented in C++ and C. As of version 0.0.0, most of the examples and tools for reviewing results are in C++ and Python.

CHAPTER 3

# Problem domain: Scaling chains of integrators

Often referred to as “the first domain,” the basic problem is to find a controller for a given chain of integrators system so that all trajectories repeatedly reach several regions while avoiding others.

## 3.1 Preparations

While below we include pointers to the main websites for dependencies, many are available via packages for your OS and may already be installed, especially if you have ROS on Ubuntu 14.04. Supported platforms are described in the Introduction.

### 3.1.1 Dependencies

- Eigen
- Boost, specifically Boost.Thread and bind.

### 3.1.2 Other

While not necessary to use the benchmark per se, `plotp.py` and `tdstat.py` provide a means to examine problem instances and results of trials, as demonstrated in the tutorial below. Together with the `fmrb` Python package, which is under `tools/fmrb-pkg/` in the repository, the following additional dependencies are present:

- NumPy, which is part of the standard scientific Python stack
- Matplotlib, also part of the standard stack
- pycddlib, a Python wrapper for Komei Fukuda’s cddlib
- Python Control Systems Library

Once these are met, install `fmrb` from your copy of the repository, e.g.,

```
cd tools/fmrb-pkg
pip install -e .
```

or get it from PyPI,

```
pip install fmrb
```

## 3.2 Tutorial

In the below code, `$FMRBENCHMARK` is the absolute path to a copy of the fmrbenchmark repository on your machine.

### 3.2.1 Demonstrations of components

To build the "standalone" (i.e., independent of ROS) examples demonstrating various parts of this benchmark, go to the `dynamaestro` directory (`$FMRBENCHMARK/domains/integrator_chains/dynamaestro`) and then follow the usual CMake build instructions. On Unix without an IDE, usually these are

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

One of the resulting programs is `genproblem`, the source of which is `$FMRBENCHMARK/domains/integrator_chains/dynamaestro/examples/standalone/genproblem.cpp`. The output is a problem instance in JSON. To visualize it, try

```
dynamaestro/build/genproblem | analysis/plotp.py -
```

from the directory `$FMRBENCHMARK/domains/integrator_chains/`.

### 3.2.2 Controller examples

Note that the `lqr.py` controller requires the Python Control System Library (`control`) and a standard scientific Python stack including NumPy. These and other dependencies are described above.

Create a catkin workspace.

```
mkdir -p fmrb_demo/src
cd fmrb_demo/src
catkin_init_workspace
```

Create symbolic links to the ROS packages in the fmrbenchmark repository required for this example.

```
ln -s $FMRBENCHMARK/domains/integrator_chains/dynamaestro
ln -s $FMRBENCHMARK/examples/sci_concrete_examples
```

Build and install it within the catkin workspace.

```
cd ..
catkin_make install
```

Because the installation is local to the catkin workspace, before beginning and whenever a new shell session is created, you must first

```
source install/setup.zsh
```

To initiate the performance of a collection of trials defined by the configuration file `mc-small-out3-order3.json` in the ROS package `sci_concrete_examples` of example controllers,

```
$FMRBENCHMARK/domains/integrator_chains/trial-runner.py -l -f mydata.json src/sci_concrete_examples/t
```

This will cause trial data to be saved to the file `mydata.json` in the local directory from where the above command is executed. A description of options can be obtained from `trial-runner.py -h`.

In a separate terminal, run the example controller using:

```
roslaunch sci_concrete_examples lqr.launch
```

You can observe the sequence of states and control inputs using `rostopic echo state` and `rostopic echo input`, respectively. At each time increment, the state labeling is published to the topic `/dynamaestro/loutput` as an array of strings (labels) corresponding to the polytopes containing the output at that time.

Because we used the `-l` flag when invoking `trial-runner.py` above, two additional topics are available. The labeling without repetition is published to "/logger/loutput_norep", and several elements (up to 3) of the state vector are published to "/logger/state_PointStamped" as a PointStamped message, which can be viewed in rviz.

Once all trials have completed, the trial data can be examined using `tdstat.py`. E.g., to get a summary about the data for each trial,

```
$FMRBENCHMARK/domains/integrator_chains/analysis/tdstat.py -s mydata.json
```

To get the labeling of the trajectory for trial 0, modulo repetition,

```
$FMRBENCHMARK/domains/integrator_chains/analysis/tdstat.py -t 0 --wordmodrep mydata.json
```

To get a description of options, try `tdstat.py -h`.

CHAPTER 4

# Problem domain: Traffic network of Dubins cars

Often referred to as “the second domain,” the basic setting is navigation in a small network of roads with vehicles that follow unicycle-like dynamics.

Many components of this problem domain are not ready for users. To begin exploring the currently available pieces, a good place to begin is the file `README` in the directory `domains/dubins_traffic/` in the fmrbenchmark repository.

CHAPTER 5

# Problem domain: Factory cart clearing

(Not released yet.)

CHAPTER 6

# Contributing

There are many ways to contribute. Major concerns to keep in mind:

- Participants should adhere to the Debian Code of Conduct. (Replace references to “Debian” with “fmrbenchmark” and “fmrchallenge” as appropriate.)
- There is not a dedicated mailing list yet, but there is an announcements newsletter.
- You must hold the copyright or have explicit permission from the copyright holder for anything that you contribute. Furthermore, to be included in this project, your contributed works must be under the standard “BSD 3-clause license” or a comparable open-source license (including public domain dedication). You can find a copy at `LICENSE` in the root of the repository.

Please report potential bugs or request features using the issue tracker.

## 6.1 Proposing benchmarks

Proposals about benchmark problems or supporting infrastructure are always welcome and need not have a demonstrating implementation. Furthermore, in your proposal you can use an implementation that is not ready for immediate inclusion in the repository, e.g., if it is created entirely in MATLAB. Such implementations are still useful because they provide a reference about your original intent and can be a basis for porting, e.g., to C++ or Python. In most cases, there are three parts involved in the inclusion of a benchmark:

1. a normative description about the problem and methods of evaluation in the Challenge Document;
2. introductory and tutorial treatment in the User’s Guide, and relevant additions to the API manual;
3. details and practical considerations for using it as part of a competition.

## 6.2 Development

Please report potential bugs or request features using the issue tracker. Bugfixes and other corrections, implementations of new features, improvements to documentation, etc. should be offered as pull requests. Patches can be submitted through other media if you prefer, but please try to make it easy to use and understand your proposed changes.

The benchmarks are primarily implemented in C++ and C. Unless there are strong motivations to use a different programming language, we prefer these for well-known reasons: they are fast, mature, standard, etc. Besides C and C++, several core tools for analysis of results are in Python and rely on widely-used numerical and scientific Python packages, among others. Observe that “tools for analysis” are not part of the benchmarks per se.

Examples can be expressed in any programming language or depend on any tool, including dependencies that have restrictive licenses. However, as with everything else in the repository, the example itself must be under the standard

"BSD 3-clause license" or a comparable open-source license (including public domain dedication). If you are going to contribute examples having dependencies that are not free as in freedom, please carefully document the special requirements for running the example controller.

In terms of planning, the project is currently sufficiently small to where it is enough to have a combination of the issue tracker and direct communication via private email or at meetings.

### 6.2.1 Style

Eventually we may create official style guidelines, but for now, please skim the source code to get an indication of the preferred style.

## 6.3 Working on physical variants of the problem domains

One of our ambitions is to create benchmarks that involve physical systems. In other words, we want to create well-documented testbeds that facilitate repeatability of published experiments involving real robot hardware and are challenging with respect to the state of the art.

There are a lot of incidental costs and resource requirements to develop physical benchmarks, such as raw materials, lab space, etc. Usually these are provided by each lab group for their own internal purposes (often with little or no public disclosure of details). However, this project is a joint effort that is not under the purview of a single grant nor institution. Thus an important manner of contribution is to realize physical variants of the benchmarks in your own lab and then give feedback about missing details, subtle considerations, etc. There is not a dedicated mailing list yet, so the best ways to contribute here are the issue tracker, as noted in the *Development* section, and via email to the authors.

## 6.4 Providing computing resources

Two important aspects of benchmarking are scale and comparability of performance results. Several of the domains are designed to have problem instances that can be arbitrarily large, e.g., Problem domain: Scaling chains of integrators. To support these ambitions, we accept donations of hardware as well as of remote access to computing resources, e.g., university-managed clusters or cloud computing services.