The C(anonical) Scan Matcher

1. Introduction

I created this package:

- To have a well-documented reference implementation of PL-ICP. If you are only interested in the core algorithm of PL-ICP, a separate concise implementation in C/-Matlab/Ruby is available.
- To have a **trustworthy** scan matcher to be used in the experiments for some papers on ICP covariance, the Cramer-Rao bound for range finders, and robot calibration. For batch experiments, it's also useful that it's pretty fast.
- To have a collection of utilies for command line (UNIX-style) manipulation of laser data, and creating beautiful maps and animations.

2. Content of this package

The core content is the C scan matching library which is quite polished, but this package contains a lot of software, only some of that in an usable state. In general, I am not ashamed of the prototypical code I write.

2.1. Stable things: C scan matching library

The directory `sm/csm` contains a scan matcher written in C, plus associated tools and apps. This is stable and reasonably bug-free.

There are many libraries in the `sm/lib` directory:

- Directory `egsl` : a light wrapper for GSL that makes manipulating matrices easy and efficient. This is documented in another file: see `sm/lib/egsl/docs`.
- Directory `options` : for processing command-line arguments and configuration files.
- Directory `json-c` : a library for JSON input/output. This is a slightly modified version of the original `json-c` library released under the MIT license.

2.2. Stable things: applications

There are many applications in the `sm/apps` directory:

- Application `sm2` : standard scan-matching. Reads a log, runs ICP, and writes the scan-matched output. Input can be both Carmen and JSON.
- Application `sm3` : like sm2, but instead of actual output it measures the performance. This is the application that produced the stats found in the paper submitted to ICRA'08.
- Application `sm1` : useful for running experiments. Reads scans from two different files, and outputs statistics.

Visualization apps:

- Application `log2pdf` : converts a laser log to a PDF map. To build this application, it is needed to install the Cairo graphics library.
- Application `sm_animate` : creates an animation for the ICP process, displaying the correspondences, etc. This application reads the output created by `sm2` with the `-file_jj` option. To build this application, it is needed to install the Cairo graphics library.

Miscellaneous Unix-style processing for laser data:

- Application `carmen2json` : converts a Carmen log to the JSON format.
- Application `ld_fisher` : computes the Fisher's information matrix. See http://purl.org/censi/2006/accuracy for details.
- Application `json_extract` : extract the n-th object from a JSON stream.
- Application `ld_slip` : adds some noise to the odometry field.
- Application `ld_smooth` : smooths the readings data.
- Application `ld_noise` : adds sensor noise.

- Application `ld_cluster_curv` : clusterize the rays based on the analysis of the curvature.
- Application `ld_linearize` : fits a line to each cluster (data must have been previously clustered, for example by `ld_cluster_curv`).

GUI apps:

- `apps/gtk_viewer` contains the prototype of a viewer using GTK. It does not work yet.

2.3. Unstable things: scripts

In the `scripts/` directory you can find:

- Script `json2matlab.rb` : converts a JSON object in a Matlab scripts. This is the holy grail of data exchange.

  Warning: at the moment, this script relies on some patches to the Ruby JSON library. Without them, it is limited to only 1 JSON object in each file.
- Script `fig2pics.rb` : used for converting FIG files to PDF. It has many more options than `fig2dev` (that is being used internally), including the ability to use a LaTeX preamble and to change the resulting bounding box.
- Script `create_video.rb` : displays the scan-matching process. This reads the journal files written by applications `sm1` and `sm2` . **Made obsolete by** `sm_animate`

2.4. Unstable things: Ruby and Matlab implementations

Unstable things include:

- Directory `sm_ruby_wrapper/` : a ruby wrapper for the `sm` C library. This wrapper is used for running some of the experiments. It is not documented and it needs tidying a little.
- Directory `rsm/` : a Ruby implementation of the same algorithms used in the `sm` library. Some times ago, the C and Matlab implementation were perfectly in sync. Now they differ a little. However, in the future I will try to get them back in sync, as the only way of having a good chance of making a bug-free implementation, is to make it twice.
- Directory `matlab/` and `matlab_new/` . The Matlab scripts are a mess that needs tidying. There's a lot in there. They are kept here because they are used for creating some of the figures in the submitted papers. Also, the first PLICP implementation was written in Matlab and is buried there, somewhere.

  Also, I occasionally tried to make sure that the scripts run fine in Octave. They do, except for the plotting.

3. Installation

3.1. Required software dependencies

This software has been tested on Mac OS X, Linux, and Windows XP (using Cygwin). It compiles with GCC (3.3 or 4.x) and the Intel C++ Compiler (ICC).

Required software:

- The build system is based on `cmake`, which is available at http://www.cmake.org/.
- The GSL, Gnu Scientific Library, available at http://www.gnu.org/software/gsl/.
- (optional) For `log2pdf` and other visualization applications, you will need the Cairo graphics library, available at http://cairographics.org. The recommended version is the stable 1.4.12.

Linux. CMake, Cairo, and GSL are probably already packaged for your Linux distribution. For example, in Ubuntu, you can simply enter this command to install all dependencies:

```
$ sudo apt-get install build-essential cmake libgsl0-dev libcairo2-dev
```

OS X. You can install GSL using Fink. You have to install Cairo manually.

Windows XP, using Cygwin. CSM runs fine on Cygwin, but very slow compared to Linux/OS X. Make sure you install the Cygwin packages `cairo`, `gsl`, `gsl-apps`, `gsl-devel`.

Windows XP, using Visual Studio. CSM doesn't compile yet on this platform. CMake can theoretically create Visual Studio projects, but I could not manage to do it. Also, some CMake code is probably Unix-specific.

3.2. Compiling

If you are lucky, this should be it:

```
$ cmake .
$ make
```

If you want to install this library system-wide, you could use:

```
$ cmake -DCMAKE_INSTALL_PREFIX:PATH=/usr/local .
$ make
$ make install
```

as the first step.

For installing the Ruby wrapper, refer to the separate instructions. If you want to use the Ruby wrapper, I suggest to install the source code in a `deploy` sub-directory of `csm`:

```
csm/
        docs/
        csm/
        rsm/
        deploy/     <---  here
```

To do this, use:

```
$ cmake -DCMAKE_INSTALL_PREFIX:PATH=`pwd`/deploy .
$ make
$ make install
```

(you have to give a complete path to `-DCMAKE_INSTALL_PREFIX:PATH`).

Later, remember to set your `PATH` variable to `csm/deploy/bin`.

3.3. Getting started

You might get started by doing this:

```
$ sm2 < in.log > out.log
```

where `in.log` is a Carmen-format log file.

You can find one in the top-level `experiments` directory: it is called `laserazosSM3.log`. So, if you installed the Cairo library, you can see the result with:

```
$ sm2 < in.log > out.log
$ log2pdf -use odometry -in out.log -out out-odometry.pdf
$ log2pdf -use estimate -in out.log -out out-estimate.pdf
```

3.4. Installing Ruby libraries and wrapper (optional)

This step-by-step guide is written by me, for me.

Installing with cmake:

```
$ cmake . -DCMAKE_INSTALL_PREFIX:PATH=/usr/local
```

First, set up some directories

```
$ export SMLIB=
$ cd $SMLIB
$ ls
..................
```

Create installation directory:

```
$ mkdir deploy
$ mkdir deploy/bin
$ export PATH=$PATH $SMLIB/deploy/bin
```

Create a new ruby installation

```
$ mkdir my_ruby
$ cd my_ruby
```

Download ruby:

```
$ wget ftp://ftp.ruby-lang.org/pub/ruby/ruby-1.8.5.tar.gz
$ tar xvzf ruby-1.8.5.tar.gz
$ ./configure --prefix=$SMLIB/deploy
$ make
$ make install
```

Now you should be able to use the new ruby installation

```
$ which ruby
<SMLIB>/deploy/bin/ruby
$ ruby --version
ruby 1.8.5 (2006-08-25)
```

Instructions for installing rb-gsl:

1. Get and install GSL. Make sure the command “gsl-config” is in command search path.
2. Download Ruby/GSL, ungzip and untar the archive rb-gsl-xxx.tar.gz.
3. Use: % cd rb-gsl-xxx/ % ruby setup.rb config % ruby setup.rb setup % ruby setup.rb install (as root)

Download rubygems:

```
$ cd $SMLIB/my_ruby
$ wget http://rubyforge.org/frs/download.php/11289/rubygems-0.9.0.tgz
$ tar xvzf rubygems-0.9.0.tgz
$ cd rubygems-0.9.0
$ ruby setup.rb
```

Now you should have the “gem” command installed:

```
$ which gem
<SMLIB>/deploy/bin/gem
```

4. The `laser_data` data structure

Laser data is passed around in a structure which is quite rich and in some ways redundant to achieve ease of use.

In C, the structure’s name is `struct laser_data`. In Ruby, it is `class LaserData`. In Matlab, it’s a generic structure.

A description of the fields follows (assume the structure is called `ld`).

Regarding the pose of the robot:

`ld.true_pose` Pose of the robot (m,m,rad), in world coordinates.

`ld.odometry` Odometry (`true_pose` corrupted by noise).

`ld.estimate` Estimate of `true_pose`.

Regarding the rays:

`ld.nrays` Number of rays.

`ld.min_theta` and `ld.max_theta` Minimum and maximum theta (radians).

`ld.theta[i]` Direction of i-th ray with respect to the robot (radians).

`ld.readings[i]` Sensor reading (meters). If the reading is not valid, then `ld.readings(i) == NAN`.

`ld.valid[i]` In C, it assumes values `0` and `1`. In Ruby, it assumes values `true` or `false`. (**TODO**: choose how to serialize).

This field is true if this ray is valid, and, in particular, `ld.readings[i]` is valid. Invalid rays occur when the obstacle is farther than the sensor horizon.

`ld.true_alpha[i]` Orientation of the normal of the surface (radians, relative to robot). It is `NAN` if not valid.

`ld.alpha[i]` Estimated orientation of the surface (radians, relative to robot). It is an estimate of `ld.true_alpha[i]`.

`ld.alpha_valid[i]` True if previous field is valid.

`ld.cov_alpha[i]` Estimated covariance of `ld.alpha[i]`.

Additional fields used during the computation:

`ld.cluster[i]` Cluster to which point i belongs. This is used for computing the orientation (at the moment a really dumb algorithm is used for clustering). If `cluster[i] == -1`, the point does not belong to any cluster.

`ld.points[i].p` Point coordinates (cartesian). Computed from the polar coordinates `theta[i]` and `readings[i]`.

`ld.points_w[i].p` Point coordinates (cartesian) in a "world" reference frame. Computed with the function `ld_compute_world_coords(LDP, double pose[3])`.

`ld.hostname` This is parsed from the Carmen data field.

`ld.tv` This is a `struct timeval` field giving a timestamp for the laser scan. Please see the section on parsing to learn how this is parsed from the Carmen log.

5. Input and output formats

The library understands two formats: a rich JSON format, and the old good Carmen format.

5.1. The JSON log format

See this site: http://www.json.org for general information about JSON.

This is a sample laser data structure. It has only 5 rays (which all happen to be invalid), and it has no `alpha`, `true_alpha`, `cluster` fields:

```
{
  "nrays": 5,
  "min_theta": null,
  "max_theta": null,
  "theta":    [ null, null, null, null, null ],
  "readings": [ null, null, null, null, null],
  "valid":    [ 0, 0, 0, 0, 0],

  "odometry": [ null, null, null ],
  "estimate": [ null, null, null ],
  "true_pose": [ null, null, null ]
}
```

Note that `NAN` is represented with `null` in the JSON format.

5.2. The Carmen log format

The 6 pose values in the log are interpreted as follows:

```
estimate.x estimate.y estimate.theta ....
odometry.x odometry.y odometry.theta
```

5.2.1. Regarding the timestamp

Regarding the timestamp "fields". The last three fields in a Carmen log can be:

```
integer   string   integer
```

This is interpreted as seconds, hostname, microseconds. This is good if you want to write a `timeval` struct to the log and *be sure* it won't be modified by precision problems when writing, and parsing, as a `double`.

If it doesn't look like a timestamp, then it is assumed that the fields are:

```
double string double
```

In this case, the first double is interpreted as the timestamp in seconds, while the second double is discarded.

The library will warn the user about these decisions by writing on the console this message:

```
sm2:inf: Reading timestamp as 'sec hostname usec'.
```

or this one:

```
sm2:inf: Reading timestamp as doubles (discarding second one).
```

6. Examples

6.1. Simple scan matching

Simple scan-matching:

```
$ sm2 < in.log > out.log
```

where `in.log` may be in either Carmen or JSON format.

6.2. Creating a PDF

Creating a PDF:

```
$ log2pdf -use odometry  -in in.log -out out_odometry.pdf
$ log2pdf -use estimate  -in in.log -out out_estimate.pdf
```

6.3. Examining one particular matching (video)

To zoom on one particular matching, write a "journal" using the `-file_jj` option of `sm2`:

```
$ sm2 -file_jj journal.txt < in.log > out.log
```

Extract what you are interested in from the journal. In this example, the 13th matching:

```
$ json_extract -nth 13 < journal.txt > matching13.txt
```

Create the animation:

```
$ sm_animate -in matching13.txt
```

6.4. Help! ICP doesn't work

Actually, there are a million reasons for which it shouldn't work. If it gives strange results, try the following:

1. Plot the data! Plot the input and plot the output using `log2pdf`.
2. Plot the animation! Use the procedure above and inspect the resulting videos.
3. Double-check the parameters you are using. Note that there are some like `max_correspondence_dist` which depend on the scale of your data. A value of 2m might work for a big robot making large movements, but not for a little Khepera.
4. Smooth your data – if your sensor is very noisy, like an Hokuyo, it's worth to do simple low-pass filtering. Especially for PLICP which uses the orientation information.

7. Embedding CSM in your programs

7.1. Linking to CSM

When CSM is installed, a pkgconfig `csm.pc` file is installed as well. This makes it easy to link to CSM.

For example, on my system, after installing CSM, I can run `pkgconfig` to get the C preprocessors and linker flags.

This is what I get on my system (on yours, paths will be different, of course).

```
 $ pkg-config --cflags csm
   -I/sw/include -I/Users/andrea/svn/cds/csm/deploy/include/cairo
   -I/Users/andrea/svn/cds/csm/deploy/include

 $ pkg-config --libs csm
   -L/sw/lib -L/Users/andrea/svn/cds/csm/deploy/lib
   -lcsm-static -lgsl -lgslcblas -lm
```

If you use GNU Make, a basic Makefile for your program linking to CSM would be something like:

```
CSM_FLAGS=`pkg-config --libs --cflags csm`

myprogram: myprogram.c
        gcc $(CSM_FLAGS) -o myprogram myprogram.c
```

You can download the sources for this example in the repository (directory `docs/example-linking-make`).

If you use CMake — and you should! — it is reccomended that you use something like the following in your `CMakeLists.txt`.

```
cmake_minimum_required(VERSION 2.4)
project(myproject)

# Require we have pkgconfig installed
find_package(PkgConfig REQUIRED)
# Tell pkgconfig to look for CSM
pkg_check_modules(CSM REQUIRED csm)

IF(${CSM_FOUND})
        MESSAGE("CSM_LIBRARY_DIRS: ${CSM_LIBRARY_DIRS}")
        MESSAGE("CSM_LIBRARIES: ${CSM_LIBRARIES}")
        MESSAGE("CSM_INCLUDE_DIRS: ${CSM_INCLUDE_DIRS}")

        INCLUDE_DIRECTORIES(${CSM_INCLUDE_DIRS}) # important!
        LINK_DIRECTORIES(${CSM_LIBRARY_DIRS})    # important!
ELSE(${CSM_FOUND})
        MESSAGE(FATAL_ERROR "CSM not found. Check that the environment \
        variable PKG_CONFIG_PATH includes the path containing the file 'csm.pc'.")
ENDIF(${CSM_FOUND})

add_executable(myprogram myprogram.c)

target_link_libraries(myprogram ${CSM_LIBRARIES}) # important!
```

You can download the sources for this example in the repository (directory `docs/example-linking-cmake`).

7.2. Accessing CSM functions from your applications

All functions that you would be interested in using are accessible by including one header:

```
#include <csm/csm_all.h>
```

If you are linking from C++, as opposed to C, all functions are enclosed in the `CSM` namespace. Therefore, you need something like the following.

```
#include <csm/csm_all.h>
using namespace CSM;
```

7.3. Orienting oneself in the source code

The main function to call is the following:

```
void sm_icp(struct sm_params*params, struct sm_result*result);
```

This implements matching between two laser scans. All the applications discussed above (`sm1`, `sm2`, etc.) are essentially wrapper of `sm_icp`: they fill in the `params` structure, and read from the `result` structure.

The `sm_params` structure is described in the `<csm/algos.h>` header file. It contains parameters for both ICP and other algorithms (like HSM; however, only (PL)ICP is considered stable in CSM)

Note that many of the parameters greatly influence the behavior of PLICP, so it is worth reading them all. If you run `sm2 -help` you will see the default values, which are reasonable as a starting point.

We now briefly discuss the main parameters.

- `params->laser_ref` : pointer of a structure of type `laser_data` (described before in this document) representing the "ref"erence scan (first scan).
- `params->laser_sens` : pointer of a structure of type `laser_data` representing the second scan.
- `params->first_guess` : first guess (x,y,theta).

Parameters that influence stopping:

- `max_iterations` : maximum number of iterations
- `epsilon_xy` , `epsilon_theta` : stop if change below these thresholds