Development of Scientific Applications with the Mobile Robot Programming Toolkit

The MRPT reference book

by Jose Luis Blanco Claraco

Machine Perception and Intelligent Robotics Laboratory

University of Malaga

Version: October 31, 2008

ii

Note:

This book is uncompleted. The most up-to-date version will always be available online at:

http://mrpt.sourceforge.net/mrpt-book/

Contents

Listings

Part I

First steps

Chapter 1

Introduction

1.1 Why a new library?

Many good scientific programs and programming libraries exist out there. When working with matrixes, vectors, and graphical representations, applications like MATLAB or Octave excel. If one's needs are efficient image algorithms under C and C++, OpenCV or VXL are good bets. Other libraries provide Bayesian inference or random number generators for a variety of probability distributions. When interfacing a variety of sensors, a low-level language as C is probably one of the best ways to develop a robust and efficient implementation. A problem raises only when a project requires performing many or all of these tasks under a single and sensible development framework, since each library declares its own data structures. For example, an image grabbed by an OpenCV program cannot be *directly* sent to a MATLAB program which detects features.

The development of mobile robotics software is one of those complex projects that require having at hand a variety of heterogeneous tools: a robot may capture an image from an IEEE1394 camera, extract features from it, read odometry information from wheel encoders through a serial port, and then fuse all these data using a Kalman filter in matrix form. This contains tasks which range from low-level code (close to hardware), up to linear algebra.

1.2 What is MRPT?

To face the development of such software, we have created the *Mobile Robot Programming Toolkit*, or MRPT. This framework acts as the *glue* that makes

3

possible to interconnect several third-party libraries, but it also implements several features on its own.

Despite the name, MRPT currently comprises several generic libraries in C++ which can be perfectly employed for developing any kind of scientific application that requires 2D plots, linear algebra, 3D geometry, Bayesian inference, 3D scene animations, or any combination of them.

In the specific field of *mobile robotics*, MRPT is aimed to help researchers to design and implement algorithms in the areas of Simultaneous Localization and Mapping (SLAM), computer vision and motion planning (obstacle avoidance). The libraries include classes for easily managing 3D(6D) geometry, probability density functions (pdfs) over many predefined variables (points and poses, landmarks, maps), Bayesian inference (Kalman filters, particle filters), image processing, path planning and obstacle avoidance, 3D visualization of all kind of maps (points, occupancy grids, landmarks,...), and "drivers" for a variety of robotic sensors.

1.3   What is this book about?

This document tries to address the needs of two different kinds of readers:

- Firstable, **users of the MRPT programs**. The toolkit is not only a collection of libraries, but also contains some ready-to-use programs. With those applications, a user can record data from a mobile robot, manipulate the logs if needed, and create point or occupancy grid-maps using state-of-the-art algorithms *without typing a single line of source code*.

- **Developers**. Users who pretend to integrate their own algorithms into MRPT or to use it as a layer on which to develop more powerful applications or libraries.

Obviously, many readers may fit within both kinds of readers, but for reasons of clarity, this book is structured into two well-differentiated parts. Part II addresses *using* existing programs, while Part III discusses more in-deep details required for MRPT programmers.

1.4   What is this book *not* about?

The intention is that this book does not become one of those boring, and nearly useless hard copies of a library reference. This book pretends to

let a programmer know what is inside MRPT, as a birth-eye-view. Once he or she needs to handle any specific class, the reference documentation (created with Doxygen) will be an invaluable tool, and indeed one of our main concerns during the development of MRPT has been an extensive and good reference documentation, which is available online at the MRPT web site [4].

But before reading that documentation, the programmer should have a gross idea of how things are managed within MRPT, and that is precisely the aim of this book.

1.5   How much does it cost?

MRPT is **free software**. Free in both senses: you can use it without any cost, and it is an Open Source project. We have released the sources under GNU General Public License 3. Feel free to modify the sources for your needs, to the extent allowed by the aforementioned license. If you want to contribute with patches or bug reports (or even better, bug fixes!), please contact the authors through the mailing list: mrpt-help@lists.sourceforge.net

Despite its beginnings at the MAPIR Laboratory in the University of Málaga, several people world-wide have contributed in different ways to its development since its release as an Open Source project. We kindly thank everyone who has helped in any way, and hope more people continue getting involved in the future[1].

MRPT is released under GNU GPL 3.

1.6   OS restrictions

MRPT is designed to be cross-platform. It works under 32bit and 64bit systems. Thus, the good news is that any user application developed with MRPT and no other OS-dependant API will also become cross-platform without any extra effort.

The libraries are daily tested under Windows 32bit and Linux. In theory they should also work under any POSIX-compatible system equipped with a decent C++ compiler, like Mac OS X, Solaris, BSDs, etc, but we have not verified all these platforms yet[2].

---

[1]The complete list of authors can be checked out online at http://babel.isa.uma.es/mrpt/index.php/Authors

[2]An up-to-date list of systems where MRPT has been completely tested can be found in http://babel.isa.uma.es/mrpt/index.php/Supported_Platforms

1.7   Robotic software architectures

MRPT provides several ready-to-use data structures and algorithms which can be directly used to build software aimed to be run on a vehicle or robot. In fact, some MRPT applications (e.g. `rawlog-grabber`) are designed for this purpose.

However, intelligent robots usually require a much more complex software than a single application. Robotic software architectures play the role of splitting the code into independent programs (or "modules") which, as a whole, comprise the robot *software*. In such a framework MRPT might be just a "low-level" library.

A number of publicly available frameworks exist. In our group, we developed the *BABEL* system [7], available online for download at [8]. Other development environments are the Player project [2], MOOS [10] and CAR-MEN [9].

Chapter 2

Compiling

This chapter explains how to compile the MRPT libraries and applications, and also whether a user may instead prefer a pre-compiled version.

If you are sure you prefer (or have to) compile MRPT from sources, skip the next section and continue with section 2.2.

2.1   Binary distributions

Some users may want to only *use* existing MRPT applications, so they do not pretend to develop custom programs based on MRPT. For such users, precompiled binary distributions of MRPT exist and perhaps are a better choice than compiling it from sources.

For 32bit Windows XP/Vista, binary packages are available for download at the main MRPT download page[1].

There are no precompiled MRPT packages for GNU/Linux yet. However, binary packages are in the queue for being included into Debian (eventually, this implies derivate distributions such as Ubuntu will include them too), so they will be available for installing with:

    $ sudo apt-get install mrpt-apps

or manually from `synaptic` or your preferred package manager.

2.2   Prerequisites

As with any mid or large-size software collection, MRPT requires some programs and libraries to be installed in your system *before* you can compile

---

[1]http://babel.isa.uma.es/mrpt/downloads/

it. Next sections explain the required steps for each system, through in general the main requisites are:

- **CMake:** A powerful cross-platform build system.

- **wxWidgets:** An extensive GUI toolkit.

- **OpenCV:** A widely-used computer vision library.

2.2.1   GNU/Linux

Debian, Ubuntu

Invoke:
```
    sudo apt-get install build-essential cmake libwxgtk2.8-dev libwxbase2.8-dbg
libwxgtk2.8-dbg libftdi-dev libglut3-dev libhighgui-dev lib3ds-dev
libboost-program-options-dev
```
Note that if version 2.8 of wxWidgets is not available in your distribution, it would have to be installed manually.

Fedora

Invoke as root:
```
    yum install gcc gcc-c++ make cmake wxGTK-devel opencv-devel freeglut-devel
lib3ds-devel boost-dev
```

OpenSUSE

Invoke:
```
    sudo zypper install make gcc gcc-c++ cmake cmake-gui pkg-config
zlib-devel wxGTK-devel wxGTK-gl libusb-devel freeglut-devel lib3ds-devel
libboost-program-options
```

Installing OpenCV on OpenSUSE

OpenCV must be downloaded and compiled from sources manually in OpenSUSE. Download the opencv-1.0.0.tar.gz Linux sources and follow these steps:

1. Install the dependencies. This will assure some packages required by OpenCV GUI and video grabbing. Invoke:

   ```
   sudo zypper install make gcc gcc-c++ wxGTK-devel libdc1394-devel
   libraw1394-devel libpng-devel libjpeg-devel
   ```
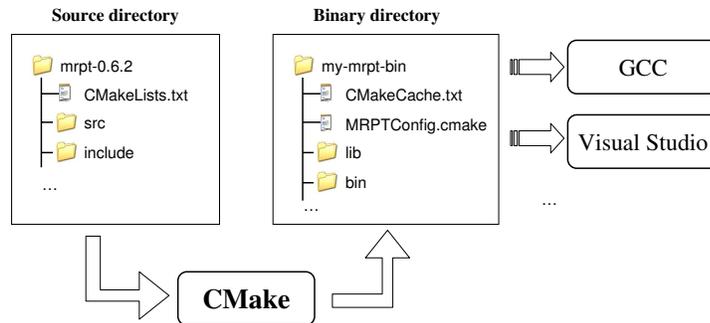
Figure 2.1: The concepts of *source* and *binary* directories with the CMake toolchain.

Optionally, if you enable the "Packman repository", the package ffm-peg should be also installed.

2. Decompress the tarball:

```
tar -xf opencv-1.0.0.tar.gz
```

3. Then go to the newly created directory and invoke the configure tool:

```
./configure
```

If everything goes fine, no error will be reported as all the dependencies are satisfied. Now compile and install OpenCV with:

```
make && sudo make install && sudo /sbin/ldconfig
```

2.2.2 Windows

2.3 Compiling

2.4 Building options

Part II

User guide

Chapter 3

Applications

3.1 RawLogViewer

3.1.1 Description

3.1.2 Usage

3.2   rawlog-grabber

3.2.1   Description

`rawlog-grabber` is a command-line application which uses a generic sensor architecture to allow collecting data from a variety of robotic sensors in real-time taking into account the different rates at each sensor may work. This program creates a thread for each sensor declared in the config file and then saves the timestamp-ordered observations to a rawlog file. The valuable utility of this application is to collect datasets from mobile robots for off-line processing.

3.2.2   Usage

This program is invoked from the command line with:

```
rawlog−grabber <config_file.ini>
```

3.2.3   Configuration files

The format of the configuration file is explained in the comments of the following prototype file. Refer also to the directory

   `MRPT/shared/mrpt/config_files/rawlog-grabber`

for more sample files and to the next sections for each specific sensor[1].

```
// −−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
//   Example  config  file  for  rawlog−grabber
//
//                   ~  The  MRPT  project  ~
//           Jose  Luis  Blanco  Claraco  (C)  2005−2008
// −−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−

//   Each  section  [XXXXX]  (except  [global])  sets  up  a  thread  in
//   the  rawlog−grabber  standalone  application.  Each  thread  collects
//   data  from  one  sensor  or  device,  then  the  main  thread  groups
//   and  orders  them  before  streaming  everything  to  a  rawlog  file.

//   The  name  of  the  sections  can  be  arbitrary  and  independent
//    of  the  sensor  label.  The  driver  for  each  sensor  is  actually
//    determined  by  the  field  "driver",  which  must  match  the  name
//    of  some  class  in  mrpt::hwdrivers  implementing  CGenericSensor.


// ══════════════════════════════════════════════════════════
//   Section:  Global  settings  to  the  application
// ══════════════════════════════════════════════════════════
```

---

[1]However, notice that the most up-to-date documentation will be always available in the reference of CGenericSensor and their derived classes.

```
[global]
// The prefix can contain a relative or absolute path.
// The final name will be <PREFIX>_date_time.rawlog
rawlog_prefix            = dataset

// Milliseconds between thread launches
time_between_launches    = 800

// SF=1: Enabled  -> Observations will be grouped by time periods.
// SF=0: Disabled -> All the observations are saved independently
//                    and ordered solely by their timestamps.
use_sensoryframes        = 1

// Only if "use_sensoryframes=1": The maximum time difference between
//  observations within a single sensory-frame.
SF_max_time_span         = 0.25        // seconds

// Observations will be processed at the main thread with this period
GRABBER_PERIOD_MS         = 1000        // ms


// Here follow sections for each sensor.
// This is one example for a Hokuyo laser scanner:

// ═══════════════════════════════════════════════════
//   SENSOR: LASER_2D
// ═══════════════════════════════════════════════════
[LASER_2D]
driver                   = CHokuyoURG
process_rate             = 90              ; Hz

sensorLabel              = HOKUYO_UTM
pose_x                   = 0      ; Laser range scaner 3D position
pose_y                   = 0      ;   on the robot (meters)
pose_z                   = 0.31
pose_yaw                 = 0      ; Angles in degrees
pose_pitch               = 0
pose_roll                = 0

COM_port_WIN             = COM3
COM_port_LIN             = ttyACM0
```

Specification for: Hokuyo Laser

Specification for: GPS

Specification for: Camera

Part III

Programming guide

Chapter 4

The libraries

The Mobile Robot Programming Toolkit (MRPT)
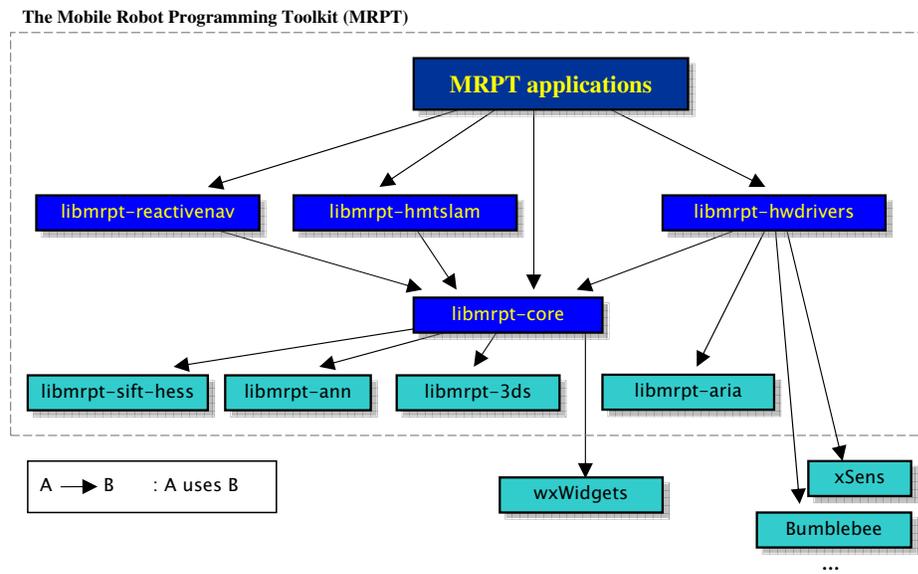
Figure 4.1: An overview of the individual libraries within MRPT.

Chapter 5

Your first MRPT program

At this point, it is assumed that MRPT has been *already compiled* in any arbitrary user directory (or, optionally, installed in the system, e.g. using `synaptic`). If this is not the case, refer to Chapter 2 for instalation instructions.

In this chapter you will learn the basics of the CMake building system and how to use it to create and compile a very simple MRPT program. The complete files of this example can be found within the MRPT packages at `MRPT/doc/mrpt_example1.tar.gz`[1].

---

[1]Or downloaded from this link: mrpt_example1.tar.gz

5.1   Source files

The first step is to include the MRPT headers in your program. Inserting the following lines will probably be enough for most applications:

```cpp
#include <mrpt/core.h>

using namespace mrpt;
using namespace mrpt::utils;
using namespace mrpt::poses;
using namespace mrpt::slam;
using namespace std;
```

If you prefer to explicitly refer to MRPT classes through their namespaces (like typing `mrpt::math::CMatrixFloat` instead of `CMatrixFloat`), remove the using namespace statements (and prepare your fingers to type a few extra characters!).

The header `mrpt/core.h` includes all the classes in a number of namespaces:

1. `mrpt::bayes`: Different particle filters and Kalman filter algorithms.

2. `mrpt::compress`: Data compression/decompression algorithms.

3. `mrpt::gui`: GUI windows for 2D and 3D visualization.

4. `mrpt::math`: A lot of functions for maths stuff, vectors, matrices, etc.

5. `mrpt::opengl`: 3D scene representation and rendering.

6. `mrpt::poses`: 2D and 3D geometry and pdf.

7. `mrpt::random`: Pseudo-random numbers generators.

8. `mrpt::scan_matching`: Scan matching-related static functions.

9. `mrpt::slam`: SLAM, localization, maps, robot's observations, etc.

10. `mrpt::synch`: Multi-threading, synchronization utilities

11. `mrpt::system`: OS abstraction layer.

12. `mrpt::utils`: Serialization, sockets, streams, etc.

13. `mrpt::vision`: Classes for computer vision, detectors, features, tracking.

If you want to include classes from additional libraries, like `mrpt-hwdrivers`, the corresponding include statements must be added:

```cpp
#include <mrpt/hwdrivers.h>  // Includes mrpt::hwdrivers namespace
using namespace mrpt::hwdrivers;
```

Now we will see a complete program. This very basic example just creates a pair of 2D (x, y, ϕ) and 3D $(x, y, z, yaw, pitch, roll)$ poses and computes the composed pose $R \oplus C$ and the distances between them:

Listing 5.1: A very simple MPRT program

```cpp
#include <mrpt/core.h>

using namespace mrpt::utils;
using namespace mrpt::poses;
using namespace std;

int main()
{
 // Robot pose: 2D (x,y,phi)
 CPose2D   R(2,1, DEG2RAD(45.0) );

 // Camera pose relative to the robot: 6D (x,y,z,yaw,pitch,roll).
 CPose3D   C( 0.5,0.5,1.5,
              DEG2RAD(-90.0),DEG2RAD(0),DEG2RAD(-90.0) );

 cout << "R:_" << R << endl;
 cout << "C:_" << C << endl;
 cout << "R+C:" << (R+C) << endl;
 cout << " |R-C|=_" << R.distanceTo(C) << endl;
 return 0;
}
```

Now, save this program as `test.cpp` and half the work is done!

5.2   The CMake project file

The simplest CMake project must contain just one file `CMakeLists.txt`. Create a file with that name and with the following contents in the same directory than the file `test.cpp`:

```
PROJECT( mrpt_example1 )

CMAKE_MINIMUM_REQUIRED(VERSION 2.4)
# ————————————————————————————————————————————
# Indicate CMake 2.7 and above that we don't want to mix relative
#  and absolute paths in linker lib lists.
# Run "cmake −−help−policy CMP0003" for more information.
# ————————————————————————————————————————————
if(COMMAND cmake_policy)
      cmake_policy(SET CMP0003 NEW)
endif(COMMAND cmake_policy)


# ——————————————————————————————————————————————————
#    The list of "libs" which can be included is:
#      − core: The main library. It will be included by default if the
#               assignment to MRPT_USE_LIBS is omitted.
#      − hmtslam         −−> libmrpt−hmtslam
#      − hwdrivers        −−> libmrpt−hwdrivers
#      − reactivenav      −−> libmrpt−reactivenav
#      − aria            −−> libmrpt−aria
#      − xsens           −−> libmrpt−xsens
# ——————————————————————————————————————————————————
SET(MRPT_USE_LIBS core)
FIND_PACKAGE( MRPT REQUIRED )

# Declare the target (an executable)
ADD_EXECUTABLE( mrpt_example1
        test.cpp
        )

# Tell the compiler to link against MRPT libraries.
TARGET_LINK_LIBRARIES( mrpt_example1 ${MRPT_LIBS})
```

There are two important steps in this CMake script: looking for the MRPT library and defining a target (which eventually will become a Visual Studio Project, or a Makefile) named `mrpt_example1` which contains only one source file `test.cpp`.

Let's review briefly how CMake look for the MRPT library. Recall Figure 2.1 and the discussion in that chapter on source vs. binary directories in CMake. With the command `FIND_PACKAGE(...)`, CMake will look for a file named `MRPTConfig.cmake`, which contains information such as where are the library header directories, or which libraries should a program link against. If you have compiled MRPT manually, this directory will be your MRPT binary directory. If MRPT has been installed in a Unix system, it will probably be at `/usr/lib/cmake/` or `/usr/local/lib/cmake/`.

5.3   Generating the native projects

Now, a native project must be created to compile your program, where *native* means a project for your preferred compiler or IDE which is supported by CMake. Some examples are: Unix makefiles, Visual Studio solutions, Code Blocks projects, Eclipse projects, etc. In any case, create a new directory to make an off-tree build, for example `first-mrpt-bin`. We will refer to the directory with the sources (`test.cpp` and `CMakeLists.txt`), as .

Under Unix or GNU/Linux, go to the new empty directory and invoke:

```
first-mrpt-bin$ ccmake {path_first-mrpt-src}
```

On Windows, execute `cmake-gui` or `cmakesetup` and select the source (`{path_first-mrpt-src}`) and binary (`first-mrpt-bin`) directories. Note that in some Linux distributions `cmake-gui` is also available.

At this point, press the button "configure" in CMake, then "generate" to build your project. If CMake complains about not finding MRPT, set manually the variable `MRPT_DIR` to the directory where you compiled MRPT (or `/usr/lib/cmake/` if it was installed through synaptic or apt).

5.4   Compile

Once generated the project for your favorite compiler, just manage it as usual. For example, for Unix Makefiles, go to the binary directory and invoke `make`. For Visual Studio, open the solution file `mrpt-example1.sln` and compile as usual.

5.5   Summary

Creating user applications with MRPT requires adding the corresponding MRPT headers to the sources and creating a CMake project which includes `MRPTConfig.cmake` using the command `FIND_PACKAGE( MRPT REQUIRED )`. The simple project presented in this chapter could be hopefuly used as a base for the user to create more complex applications.

Chapter 6

Linear algebra

In this chapter you will learn one of the most basic features of MRPT: manipulating vectors and matrices very easily. The syntax in many cases will remain very close to that used in MATLAB.

In the following, all the required classes can be included in a program with:

```cpp
#include <mrpt/core.h>

using namespace mrpt;
using namespace mrpt::math;
using namespace mrpt::utils;
using namespace mrpt::system;
```

Currently there is no support for reading/writing binary MATLAB files, but this limitation is not severe since files saved from MATLAB in plan text (with the format `--ascii`) are fully supported.

Notice that, like in C/C++ languages in general, the first element in any sequence has the index 0. This convention also applies to all matrixes and vectors in MRPT.

6.1 Matrixes

6.1.1 Declaring

Matrixes are implemented as templates in MRPT, but the following two types are provided for making programs more readable:

```cpp
typedef CMatrixTemplateNumeric<float> CMatrixFloat;
typedef CMatrixTemplateNumeric<double> CMatrixDouble;
```

Currently the only kind of matrix is variable-size, that is, there is not any class for fixed-size matrixes. However, you can create a matrix with any given size by passing it at construction time, or otherwise you can resize it later, as shown in this example:

```
CMatrixDouble    M(2,3);  // Create a 2x3 matrix
cout << M(0,0) << endl;  // Print out the left-top element

CMatrixDouble    A;  // Another way of creating
A.setSize(3,4);        //   a 2x3 matrix
A(2,3) = 1.0;          // Change the bottom-right element
```

A matrix can be resized at any time, and the contents are preserved if possible. Notice also in the example how the element at the r'th row and c'th column can be accessed through $M(r, c)$.

Sometimes, predefined values must be loaded into a matrix, and writing all the assignments element by element can be tedious and error prone. In those cases, better use this constructor[1]:

```
const double numbers[] = {
 1,2,3,
 4,5,6 };
CMatrixDouble    N(2,3,numbers);
cout << "Initialized_matrix:_" << endl << N << endl;
```

If the size of the vector does not fit exactly the matrix, an exception will raise at run-time. This example above also illustrates how to dump a matrix to the console, which is useful for debugging in case of small matrixes.

6.1.2   Storage in files

When managing large matrixes, it is useful to load or save them in files. In particular, it would be even more handful to make those files compatible with MATLAB. This format exists and is as simple as plain text files. For example, the following small program loads a matrix from a file, then compute its eigenvectors and save them to a different file:

```
CMatrixDouble H,Z,D;
H.loadFromTextFile("H.txt");   // H <- 'H.txt'
H.eigenVectors(Z,D);            // Z: eigenvectors, D: eigenvalues
Z.saveToTextFile("Z.txt");     // Save Z in 'Z.txt'
```

---

[1]This feature is available in MRPT 0.6.4 or newer.

6.2 Vectors

6.2.1 Declaring

The base class for vectors is the standard STL `std::vector`, such as a user will normally declare and manipulate objects of the types `vector_float` or `vector_double` [2], for element types being `float` or `double`, respectively:

```cpp
typedef std::vector<float> vector_float;
typedef std::vector<double> vector_double;
```

6.2.2 Resizing

To resize a vector we must use the standard `std::vector` methods, that is:

```cpp
vector_double  V(5,1); // Create a vector with 5 ones.
V.resize(10);
cout << V << endl;      // Print out the vector to console
```

6.2.3 Storage in files

There is less support yet to vector I/O than in the case of matrixes, so it is normally advisable to use matrixes when loading text files, especially when the format of the file is unknown (e.g. column vs. row vector).

Reading a vector from a text file

This works for row vectors only:

```cpp
vector_double v;
loadVector( CFileInputStream("in.txt"), v);
```

Saving to a text file

The function `vectorToTextFile` allows saving as a row, as a column, and optionally, to append at the end of the existing file:

```cpp
vector_double v(4,0); // [0 0 0 0]
vectorToTextFile(v, "o1.txt");   // Save as row
vectorToTextFile(v, "o2.txt", true); // Append a new row
vectorToTextFile(v, "o3.txt", false, true ); // Save as a column
```

---

[2]One can also use `CVectorFloat` and `CVectorDouble`, which have some useful operations implemented as methods, but most MRPT interfaces expect the simpler STL containers.

Serializing

If you prefer to serialize the vectors in binary form (see chapter 9), use the templates `vector_float_serializable` or `vector_double_serializable` exactly as a normal vector:

```
vector_double_serializable  v = linspace(0,1,100); // [0 ... 1]
CFileOutputStream("dump.bin") << v;
```

6.3   Basic operations

In this section we will go through a quick summary of unary and binary operations for matrixes, vectors, or a mix of them. Table 6.3 lists some of the most simple of these operations in common mathematical notation, in C++ using MRPT operators and alternative functional forms.

Description	Operation	MRPT C++	Alternative
Read element	$a \leftarrow M(i,j)$	a = M(i,j)	
Write element	$M(i,j) \leftarrow a$	M(i,j) = a	
Matrix inverse	M^{-1}	!M	M.inv()
Matrix transpose	$M^{\top}$	~M	
Matrix assignment	$Q \leftarrow M$	Q = M	
Matrix comparison	$Q = M?$	Q == M, Q!=M	
Matrix sum/substract	$M + Q, M - Q$	M+Q, M-Q	
In place sum	$M \leftarrow M + Q$	M+=Q	
Vector sum/substract	$v + w, v - w$	v+w, v-w	
Scalar multiplication	$M \leftarrow Ma$	M*=a	
Matrix multiplication	MQ	M*Q	
Matrix multiplication	$M \leftarrow MQ$	M = M*Q	M.multiply(Q)
Matrix/vector mult.	Mv	M*v	
Multiply by inverse	MQ^{-1}	M/Q	
Determinant	$|M|$	M.det()	

Naturally, some operations carry restrictions on the sizes of the operants (e.g. matrix multiplication). An exception will be thrown if invalid operations are found in run-time. This table does not contain all the implemented operators, for all the details please refer to:

- `mrpt::math`

- `mrpt::math::CMatrixTemplateNumeric<T>`

Other methods which may render very useful to those programmers familiarized with MATLAB are:

- `M.ones(A,B)` : Generates a $A \times B$ matrix of ones.

- `M.zeros(A,B)` : Generates a $A \times B$ matrix of zeroes.

- `M.unit(A)` : The $A \times A$ unity matrix.

- `size(M,1)` : Number of rows in M, equivalent to `M.getRowCount()`.

- `size(M,2)` : Number of columns in M, equivalent to `M.getColCount()`.

- `v=linspace(a,b,N)` : Generates a vector v with N elements in the range $[a, b]$.

- `mean(v)`, `stddev(v)` : Mean and standard deviation of the vector v. There is also a combined `meanAndStd(...)`.

- `cumsum(v)` : Cumulative sum of vector v.

- `histogram(v,...)` : Histogram of a vector. See reference documentation.

As an example of the operators described so far, the equation

$$R = H \cdot C \cdot H^\top$$

can be implemented with the next code fragment:

```
CMatrixDouble   C(3,3);
CMatrixDouble   H(5,3);

// C=diag([1 2 3])
C(0,0) = 1;
C(1,1) = 2;
C(2,2) = 3;

// randomize matrix
mrpt::random::matrixRandomUni(H,-1.0,1.0);

CMatrixDouble   R = H * C * (~H);
```

However, this operation, like many others have specialized methods which much better performance. These common expressions should be known to take advantage of them, hence they are summarized in the next section.

6.4 Optimized matrix operations

Many common operations with matrices have efficient implementations, as summarized in Table 6.3. In the table M, A, B, C represent matrixes while v, w are vectors and x is a scalar. All these elements must be of the appropriate sizes for the corresponding operations to make sense. For clarity, some terms in the "operation" column are represented in MATLAB notation.

Operation	Efficient implementation	Remarks
$M = M + A^\top$	`M.addAt(A)`	
$M = M + A + A^\top$	`M.addAAt(A)`	A must be square
$M = AB^\top$	`M.multiply_ABt(A,B)`	
$M = AA^\top$	`M.multiply_AAt(A)`	
$M = A^\top A$	`M.multiply_AtA(A)`	
$w = Ab$	`A.multiply_Ab(b,w)`	
$w = A^\top b$	`A.multiply_Atb(b,w)`	
$M = AB$	`M.multiply_result_is_symmetric(A,B)`	AB should be symmetric
$M = ABA^\top$	`A.multiplyByMatrixAndByTranspose(B,M)`	B symmetric
$M = M + ABA^\top$	`A.multiplyByMatrixAndByTranspose(B,M,false,0,true)`	B symmetric
$x = ABA^\top$	`A.multiplyByMatrixAndByTransposeScalar(B)`	B symmetric, result scalar
$M = ABC$	`M.multiplyABC(A,B,C)`	
$M = ABC^\top$	`M.multiplyABCt(A,B,C)`	
$M = AB(r0:end, c0:(c0+c))$	`A.multiplySubMatrix(B,M,c0,r0,c)`	
$M = A^{-1}$	`A.inv_fast(M)`	Contents of A are lost
$sum(A(:))$	`M.sumAll()`	

6.5   Text output

6.6   Matrixes manipulation

6.6.1   Extracting a submatrix

For example, the following MATLAB statement:

$$A = C(6:8, 7:8);$$

becomes:

```
CMatrixDouble   C(10,10);
CMatrixDouble   A(3,2);   // Set to the size of the patch to extract
C.extractMatrix(5,6,A)
```

Notice again how in MATLAB the first elements are referenced as 1 while in MRPT they have 0 as index.

6.6.2   Extracting a vector from a matrix

Extracting a column, for example $v = C(:,3)$, can be implemented with:

```
CMatrixDouble   C(10,10);
vector_double   v;
C.extractCol(2,v);
```

And equivalently for rows, for example $v = C(4,:)$:

```
CMatrixDouble   C(10,10);
vector_double   v;
C.extractRow(5,v);
```

6.6.3 Building a matrix from parts

A matrix can be also built from its 4 parts, such as:

$$M = \left(\begin{array}{cc} A & B \\ C & D \end{array} \right)$$

with:

```
 CMatrixDouble   M;
 M.joinMatrix(A,B,C,D);
```

Many other methods exist (please, see the reference for further details) with self-explaining names: `insertRow`, `appendRow`, `insertCol`, `insertMatrix` (for inserting a submatrix in a larger matrix), etc.

6.7 Matrix decomposition

Chapter 7

Mathematic algorithms

7.1 Fourier Transform (FFT)

7.2 Statistics

Mean, std, meanAndStd.

7.3 Spline interpolation

7.4 Spectral graph partitioning

7.5 Quaternions

7.6 Geometry functions

7.7 Numeric Jacobian estimation

Chapter 8

3D geometry

8.1 Introduction

8.2 Homogeneous coordinates geometry

8.3 Geometry elements in MRPT

8.3.1 2D points

8.3.2 3D points

8.3.3 2D poses

### 8.3.4 3D poses
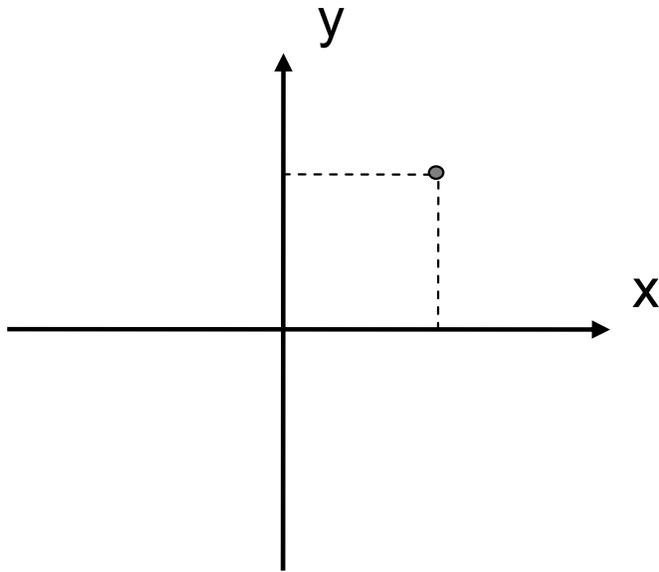
Figure 8.1: A point in 2D.
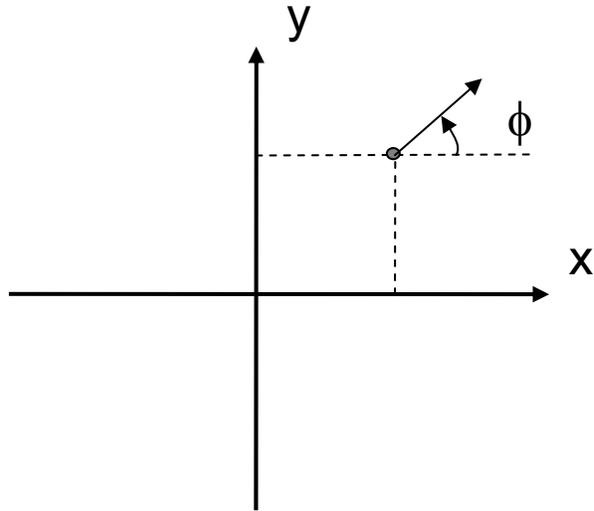
Figure 8.2: A pose in 2D.
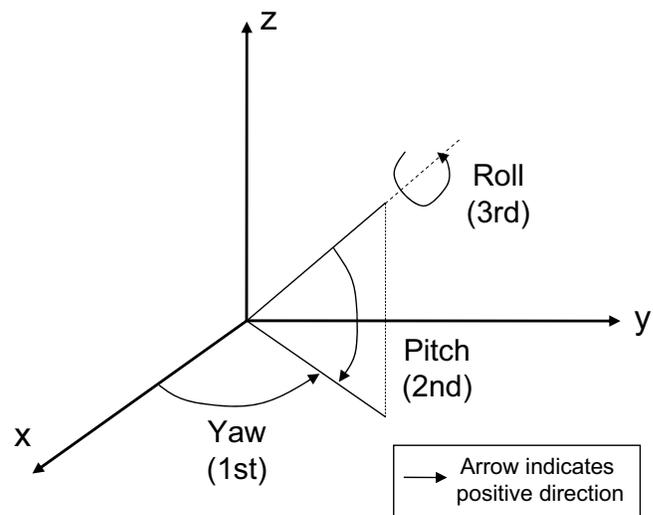
Figure 8.3: A pose in 3D.

Chapter 9

Serialization

9.1   The problem of persistence

Serializing consists of taking an existing object and converting it into a sequence of bytes, in any given format, such as the contents and state of the object can be afterward reconstructed, or deserialized.

9.2   Approach used in MRPT

There are many C++ libraries for serializing out there (e.g. boost), although the MRPT C++ library uses a simple, custom implementation with the following aims:

1. **Simplicity:** A few and small core functions only.

2. **Versioning:** If a class changes along time (something really common), a new version number will be assigned to its serialization, but old stored data can be still imported.

3. **C++ compiler independence:** Use only standardized data-lengths. For example, a data of type "int" has different lengths depending on the machine, thus it is not allowed to serialize an "int" variable without forcing it to a known length.

Currently, the only supported format for serialization is binary, i.e. there is no support for XML. The reason is that, for robotic applications, it is typically more important to save data size (and transmission times) between a running, real-time system. The actual binary frame for each serialized object is sketched in Figure 9.1[1].

---

[1]In versions before MRPT 0.5.5 the end flag was not present and the first and third fields were 4 bytes wide (instead of just 1). However, data saved in the old format can be still loaded without problems.

| 1 byte	N bytes	1 byte		1 byte
1	Class name length (N)	Class name (without final '\0')	Serialization version	Object data	End flag 0x88
MSB LSB				
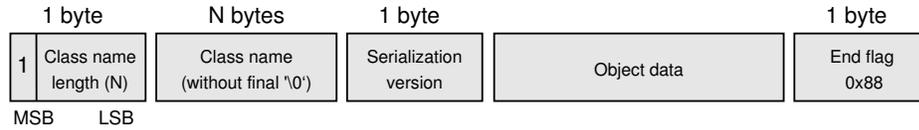
Figure 9.1: The binary format of serialized objects in MRPT.

When an object is serialized, its contents are written to a generic destination via a `CStream` class. The list of currently implemented streams are (see the reference of `utils::CStream` for more information).

The typical usage of serialization for storing an existing object into, for example, a file, is to use the `<<` operator of the `CStream` class:

```cpp
#include <mrpt/core.h>

using namespace mrpt;
using namespace mrpt::slam;
using namespace mrpt::math;
using namespace mrpt::utils;

int main()
{
 // Declare serializable objects:
 COccupancyGridMap2D    grid;
 CMatrix                M(6,6);

 // Do whatever...

 // Serialize it to a file:
 CFileOutputStream("saved.gridmap") << grid << M;

 return 0;
}
```

To restore a saved object, you can use two methods, depending of whether you are sure about the class of the object which will be read from the stream, or not. If you know the class of the object to be read, you can simply use the ¿¿ operator towards an existing object, which will be passed by reference and its contents overwritten with those read from the stream. An example:

```cpp
// Declare serializable objects:
COccupancyGridMap2D    grid;
CMatrix                M;

// Load from the file:
CFileInputStream("saved.gridmap") >> grid >> M;
```

The other situation if when you don't know the class of the object which will be read. In this case it must be declared a smart pointer to a generic `utils::CSerializable`

object (initialized as NULL to indicate that it is empty), and after using the `>>` operator it will point to a newly created object with the deserialized object:

```
// Declare serializable objects:
CSerializablePtr obj;  // NULL pointer

// Load from the file:
CFileInputStream("saved.gridmap") >> obj;

std::cout << "Object_class:" << obj->GetRuntimeClass()->className;
```

The next section explains the most important methods of `utils::CSerializable` and runtime class information. In the case of loading objects of unknown class, it is important to read the MRPT registration mechanism and when you should call it manually.

Note that these code examples do not catch potential exceptions (more about exception management in the MRPT here).

Apart from using the operators `<<` and `>>` over a `utils::CStream`, there are two independent functions, `utils::ObjectToString` and `utils::StringToObject`, which serialize and deserialize, respectively, an object into a standard STL string (`std::string`). The difference of these functions with serialization over normal `CStream`'s is that the binary data stream is encoded to avoid null characters (`'\0'`), such as the resulting string can be passed as a `char *`. Avoid using these functions but when strictly necessary, since they introduce an additional processing delay.

9.3 Run-time class identification

All serializable classes must inherit from the virtual class utils::CSerializable, which provides standard methods to manage any serializable object without knowing its real class. The most common operation is probably to check whether an object is of a given type, which can be performed by:

```
CSerializablePtr obj;
stream >> obj;

// Test if "obj" points to an object of class "CMatrix".
if ( IS_CLASS(obj,CMatrix) )
// Or (old format):
if ( obj->GetRuntimeClass() == CLASS_ID( CMatrix ) )
```

If the class to test is not in the current namespace (and there is not a `using namespace NAMESPACE;`), you can alternatively use `CLASS_ID_NAMESPACE`, for example:

```
if ( obj->GetRuntimeClass() == CLASS_ID_NAMESPACE( CMatrix, UTILS) ) ...
```

The method CSerializable::GetRuntimeClass() actually returns a pointer to a UTILS::TRuntimeClassId data structure, which contains other useful members:

1. The class name as a string:

```
obj−>GetRuntimeClass()−>className ;
```

2. Checking whether a class is a descendent of a given virtual class. An example:

```
void func ( CMetricMap ∗ aMap )
{
 if (IS_DERIVED(aMapCPointsMap))
 {
  CPointsMap ∗pMap = (CPointsMap∗) aMap;
 }
}
```

Other useful method of any serializable object is CSerializable::duplicate, which makes a copy of the object. The internal data, pointers, etc... will be really duplicated and the original object can be safely deleted.

9.4   Writing new serializable classes

9.5   Serializable STL containers

Chapter 10

Smart Pointers

10.1 Overview

Why are smart pointers (SM) needed? Aliases.

In MRPT we have chosen the wonderful implementation of smart pointers found in the STLplus C++ Library Collection [11] due to its versatility, clean interface and proven robustness. In the following we give some basic tips on SM usage which should be enough in most situations. If more in-deep information is needed, please refer to the project website in [11].

10.2 Uses within MRPT

10.3 Do's and Don'ts

10.3.1 Always create from dynamic memory

10.3.2 Do not create a SM from a local variable

10.3.3 Freeing a shared object

10.3.4 Freeing an alias only

Chapter 11

Images

11.1 The central class for images

The main class for image storage is `CMRPTImage`, which internally fully relies on the IPL format and OpenCV functions for memory management, format conversions, file I/O, etc. Basically, it is a wrapper for OpenCV C library functionality with the more attractive appearance of a C++ class and extended with many MRPT-specific algorithms.

11.2 Basic image operations

11.3 Feature extraction

11.4 SIFT descriptors

Chapter 12

GUI classes

12.1   Windows from console programs

12.2   Bitmapped graphics

See `mrpt::gui::CDisplayWindow`.

12.3   3D rendered graphics

See `mrpt::gui::CDisplayWindow3D`.

12.4   2D vectorial plots

See `mrpt::gui::CDisplayWindowPlots`.

Chapter 13

OS Abstraction Layer

13.1 Cross platform Support

To write cross-platform and cross-compiler code, we need a layer of functions that act like a minimum set of services found on any OS and compiler. In MRPT, these methods are concentrated in the namespace `mrpt::system::os`, and comprise a range of different areas as enumerated next.

13.2 Function Areas

13.2.1 Threading

13.2.2 Sockets

13.2.3 Time and date

13.2.4 String parsing

13.2.5 Files

Chapter 14

Observations

COObservationImage
Reference system on the camera:
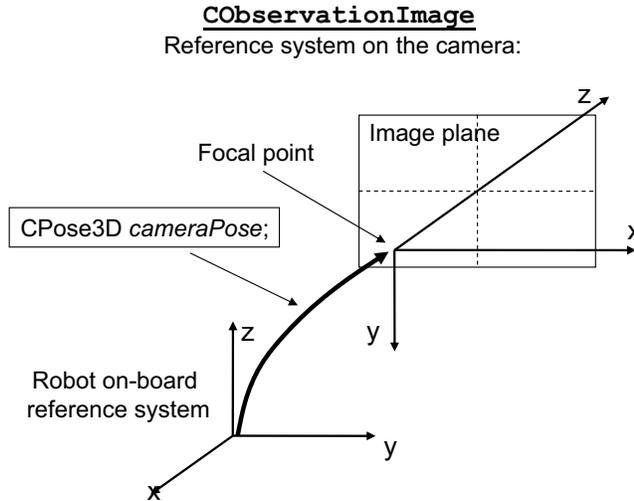
Figure 14.1: Representation of single camera observations.

COObservationStereoImages
Reference systems related to both cameras:
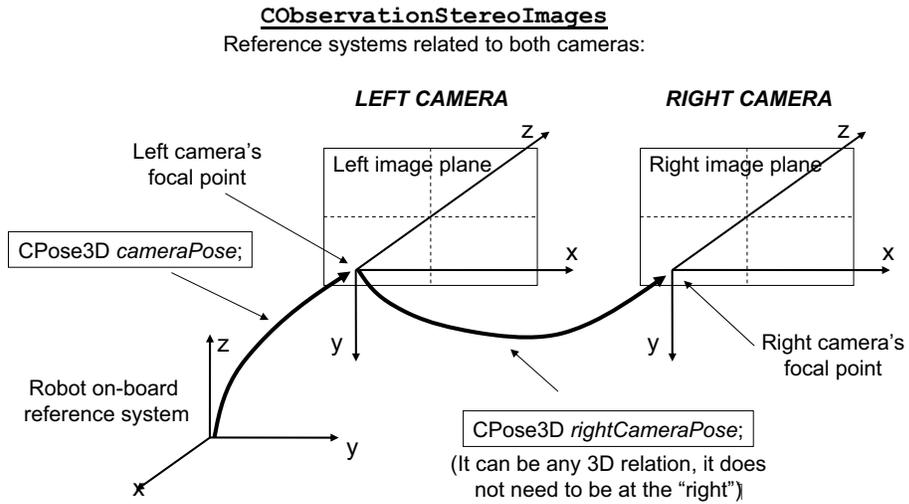

Figure 14.2: Representation of stereo image observations.

Chapter 15

Probabilistic Motion Models

15.1 Introduction

Within a particle filter, the samples are propagated at each time step using some given proposal distribution. A common approach for mobile robots is taking the probabilistic motion model directly as this proposal.

In the MRPT there are two models for probabilistic 2D motion, implemented in `mrpt::slam::CActionRobotMovement2D`.

To use them just fill out the option structure `motionModelConfiguration` and select the method in:

`CActionRobotMovement2D::TMotionModelOptions::modelSelection`.

An example of usage would be like:

```cpp
using namespace mrpt::slam;
using namespace mrpt::poses;

CPose2D actualOdometryReading(0.20, 0.05, DEG2RAD(1.2) );

// Prepare the "options" structure:
CActionRobotMovement2D                          actMov;
CActionRobotMovement2D::TMotionModelOptions opts;

opts.modelSelection = CActionRobotMovement2D::mmThrun;
opts.thrunModel.alfa3_trans_trans = 0.10f;

// Create the probability density
// distribution (PDF) from a 2D odometry reading:
actMov.computeFromOdometry( actualOdometryReading, opts );

// For example, draw one sample from the PDF:
CPose2D sample;
actMov.drawSingleSample( sample );
```

This chapter provides a description of the internal models used by these methods.
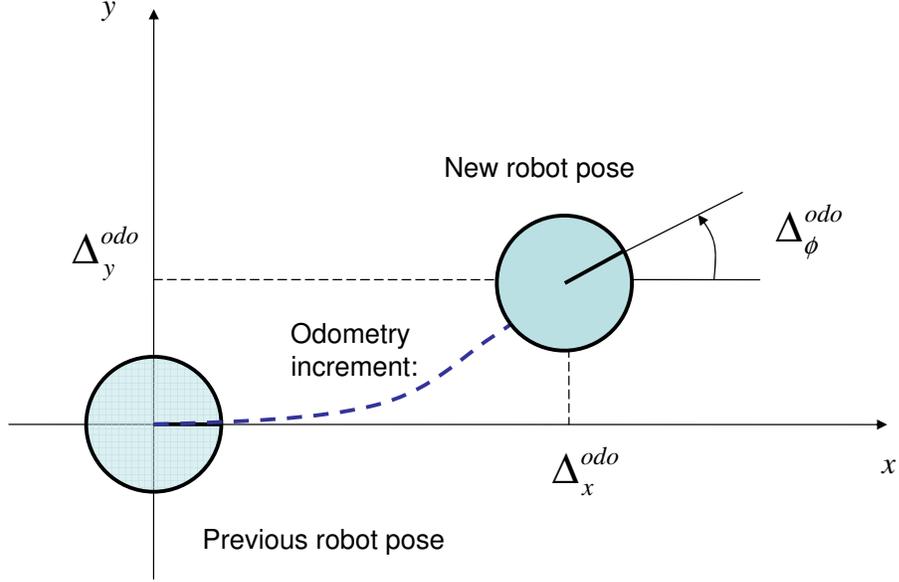
Figure 15.1: Variables in the Gaussian motion model.

15.2   Gaussian probabilistic motion model

Assume the odometry is read as incremental changes in the 2D robot pose. The odometry readings are denoted as $\left(\Delta_x^{odo} \ \Delta_y^{odo} \ \Delta_\phi^{odo} \right)$. The model for these variables is depicted in Figure 15.1.

The equations that relate the prior robot pose $(x \ y \ \phi)$ and the new pose $(x' \ y' \ \phi')$ after the incremental change are: (based on the proposal in [6])

$$
\begin{pmatrix} x' \\ y' \\ \phi' \end{pmatrix} = \begin{pmatrix} x \\ y \\ \phi \end{pmatrix} + \begin{pmatrix} \cos(\phi + \frac{\Delta_\phi^{odo}}{2}) & -\sin(\phi + \frac{\Delta_\phi^{odo}}{2}) & 0 \\ \sin(\phi + \frac{\Delta_\phi^{odo}}{2}) & \cos(\phi + \frac{\Delta_\phi^{odo}}{2}) & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \Delta_x^{odo} \\ \Delta_y^{odo} \\ \Delta_\phi^{odo} \end{pmatrix}
$$

Our aim here is to obtain a multivariate Gaussian distribution of the new pose, given that the prior pose has a known value (it is the particle being propragated). In this case we can just model how to draw samples from a prior pose of (0 0 0), and then the samples can be composed using the actual prior pose.

Using this simplification:
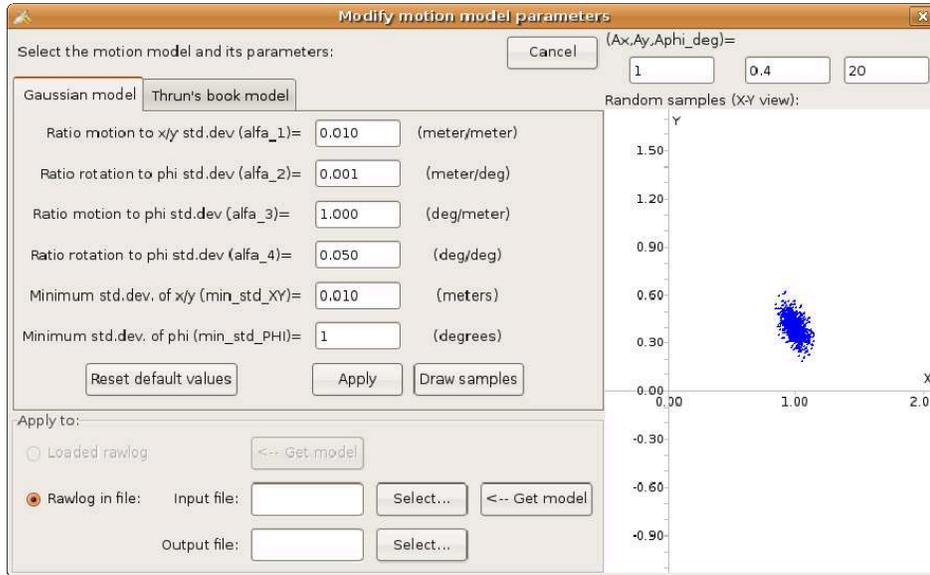
Figure 15.2: Simulation of a Gaussian motion model in RawlogViewer.

$$
\begin{pmatrix} x' \\ y' \\ \phi' \end{pmatrix} = \begin{pmatrix} \cos\frac{\Delta_\phi^{odo}}{2} & -\sin\frac{\Delta_\phi^{odo}}{2} & 0 \\ \sin\frac{\Delta_\phi^{odo}}{2} & \cos\frac{\Delta_\phi^{odo}}{2} & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \Delta_x^{odo} \\ \Delta_y^{odo} \\ \Delta_\phi^{odo} \end{pmatrix} = H \begin{pmatrix} \Delta_x^{odo} \\ \Delta_y^{odo} \\ \Delta_\phi^{odo} \end{pmatrix}
$$

The mean of the Gaussian can be simply computed from the composition of the prior and the odometry increment. For the covariance, we need to estimate the variances of the three variables of the odometry increment. We model them as having independent, zero-mean Gaussian errors. The errors will be composed of terms that capture imperfect odometry and potential drift effects.

We denote as Σ the diagonal matrix having the three variances of the odometry variables, modeled as:

$$
\sigma_{\Delta_x^{odo}} = \sigma_{\Delta_y^{odo}} = \sigma_{xy}^{min} + \alpha_1\sqrt{(\Delta_x^{odo})^2 + (\Delta_y^{odo})^2} + \alpha_2|\Delta_\phi^{odo}|
$$
$$
\sigma_{\Delta_\phi^{odo}} = \sigma_\phi^{min} + \alpha_3\sqrt{(\Delta_x^{odo})^2 + (\Delta_y^{odo})^2} + \alpha_4|\Delta_\phi^{odo}|
$$

The default parameters (loaded in the constructor and available in RawLogViewer) are:

$$\begin{aligned}
\alpha_1 &= 0.05 \ meters/meter \\
\alpha_2 &= 0.001 \ meters/degree \\
\alpha_3 &= 5 \ degrees/meter \\
\alpha_4 &= 0.05 \ degrees/degree \\
\sigma_{xy}^{min} &= 0.01 \ meters \\
\sigma_{\phi}^{min} &= 0.20 \ degrees
\end{aligned}$$

And finally, the covariance of the new pose after the odometry increment (C) is computed by means of:

$C = J \ \Sigma \ J^t$

where J stands for the Jacobian of H.

An example of samples obtained using this model with the RawLogViewer application is represented by Figure 15.2.
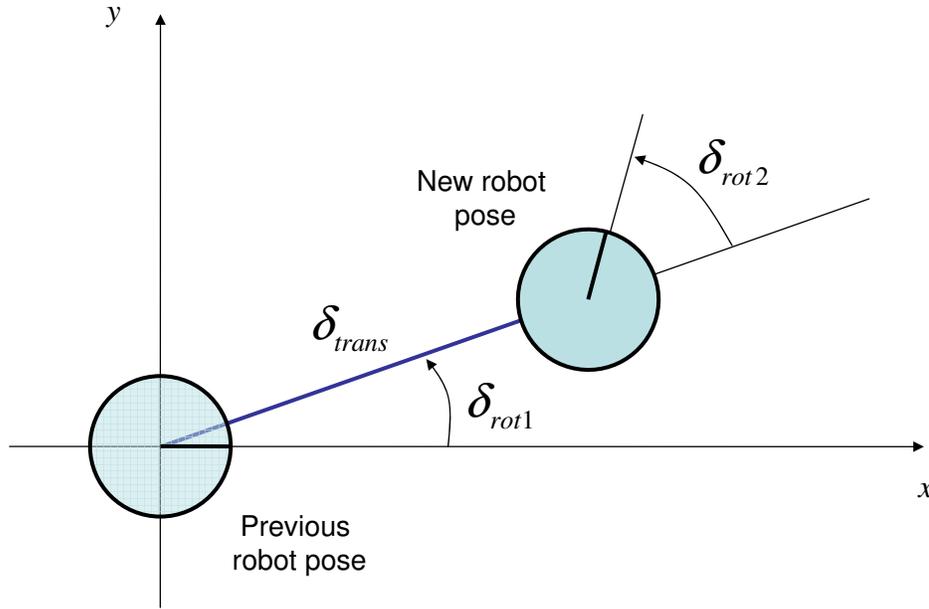
Figure 15.3: Variables in the particle-based motion model.

15.3 Thrun et al.'s book particle motion model

Like above, denote the odometry readings as $\left(\Delta_x^{odo} \ \Delta_y^{odo} \ \Delta_\phi^{odo} \right)$, and let's assume that the prior robot pose is $(0\ 0\ 0)$, which means that we want to draw samples of the robot increment, not the final robot pose (to simplify the equations without loss of generality). Then, the new robot pose, which we want to draw samples from is:

$$\begin{pmatrix} x' \\ y' \\ \phi' \end{pmatrix} = \begin{pmatrix} \cos \hat{\delta}_{rot1} & 0 & 0 \\ \sin \hat{\delta}_{rot1} & 0 & 0 \\ 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} \hat{\delta}_{trans} \\ \hat{\delta}_{rot1} \\ \hat{\delta}_{rot2} \end{pmatrix}$$

Where the variables correspond to the robot pose increment as is shown in Figure 15.3.

Here, the variables $\hat{\delta}_{trans}$, $\hat{\delta}_{rot1}$ and $\hat{\delta}_{rot2}$ are the result of adding a Gaussian, zero-mean random noise to the actual odometry readings:

$$\begin{aligned} \hat{\delta}_{trans} &= \delta_{trans} + \epsilon_{trans} & \epsilon_{trans} &\sim \mathcal{N}(0, \sigma_{trans}^2) \\ \hat{\delta}_{rot1} &= \delta_{rot1} + \epsilon_{rot1} & \epsilon_{rot1} &\sim \mathcal{N}(0, \sigma_{rot1}^2) \\ \hat{\delta}_{rot2} &= \delta_{rot2} + \epsilon_{rot2} & \epsilon_{rot2} &\sim \mathcal{N}(0, \sigma_{rot2}^2) \end{aligned}$$
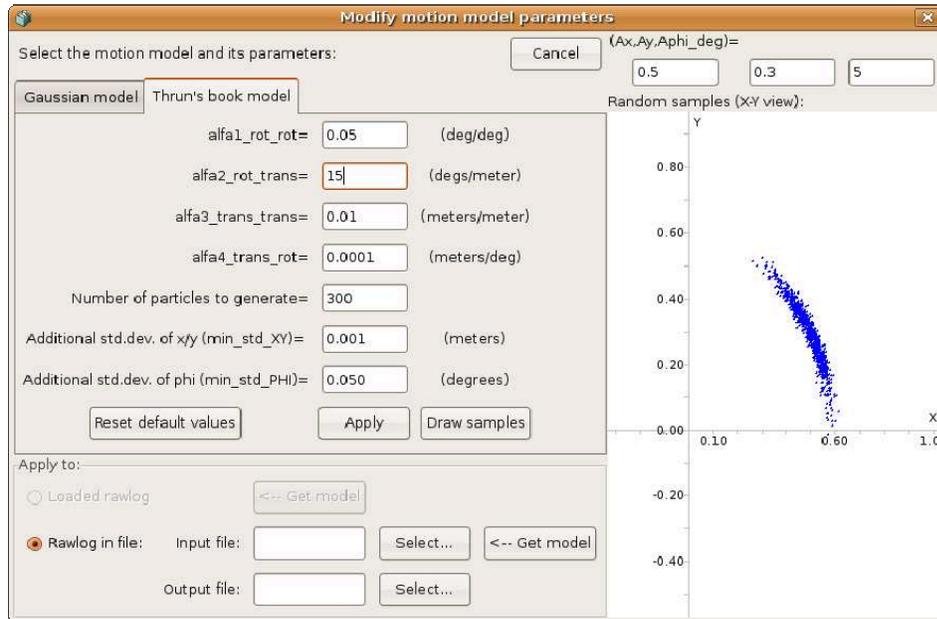
Figure 15.4: Simulation of a particles motion model in RawlogViewer.

The model described in [12] employs the following approximations for the values of the standard deviations required for the equations above:

$$\begin{aligned}
\sigma_{rot1} &= \alpha_1|\delta_{rot1}| + \alpha_2\delta_{trans} \\
\sigma_{trans} &= \alpha_3\delta_{trans} + \alpha_4(|\delta_{rot1}| + |\delta_{rot2}|) \\
\sigma_{rot2} &= \alpha_1|\delta_{rot2}| + \alpha_2\delta_{trans}
\end{aligned}$$

This is the model implemented in CActionRobotMovement2D when setting "CActionRobotMovement2D::TMotionModelOptions::modelSelection" to "mmThrun". Actually, a small additional error is summed to each pose component (x, y, ϕ) to avoid that for a null odometry increment the movement for all the particles become exactly zero, which may lead a particle filter to degenerate.

Figure 15.4 shows an example of samples generated using this model, for an excessively large value of α_2 (a very large "slippage"), generated by the application RawLogViewer.

Chapter 16

Sensor Interfaces

This chapter describes the two parts in which classes of the library `mrpt-hwdrivers` are divided: those providing the basis of communications (USB, serial), and the sensors themselves.

16.1   Communications

16.1.1   Serial ports

Even nowadays, lots of devices offer serial ports (or embedded USB-to-serial converters) as interfaces due to their simplicity of use. In MRPT, a serial port can be managed with the class `hwdrivers::CSerialPort`. An example of usage would be as follows:

```
#include <mrpt/hwdrivers.h>
...
CSerialPort   ser;
ser.setSerialPortName("ttyS0"); // or "COM3", ...
ser.setConfig(9600 /*baud*/, 0 /*no parity*/, 8 /*8 bit words*/ );
ser.open();
if (!ser.isOpen())  { // Report error }
ser.Read( ... );
ser.Write( ... );
ser.close();   // optional: it closes on destruction anyway
```

In addition, a serial port implements the generic `CStream` interface, thus it is perfectly legal to transfer arbitrarily complex objects through a serial connection as in:

```
COccupancyGridMap2D   map;
ser << map;
```

However, the most likely use of a serial ports is to send and receive short textual messages, thus the most useful methods are `Read` and `Write`.

Names of serial ports

In Windows, serial ports appear with names `COM1`, `COM2`, `COM3`, `COM4` and `\\.\COMXX` for the rest. However, if you pass a name without the prefix `\\.\` it will be added automatically.

In Linux, a variety of names can be found such as `ttyUSB0`, `ttyS0` or `ttyACM0`. It is not required to provide the full path to the device (eg. `/dev/ttyS0`), as in Windows, it will be added transparently.

As follows from above, always keep serial port names as **strings**, not only as **a number** since it will be not enough in a cross-platform application.

Timeouts

Slight changes in the timeouts of your connection can be lead to random and hard to debug errors with no apparent reason. The proper way of setting these delays is through the method:

```cpp
void CSerialPort :: setTimeouts (
    int   ReadIntervalTimeout ,
    int   ReadTotalTimeoutMultiplier ,
    int   ReadTotalTimeoutConstant ,
    int   WriteTotalTimeoutMultiplier ,
    int   WriteTotalTimeoutConstant   )
```

where all the fields have the same meaning than in the Windows API[1].

16.1.2   USB FIFO with FTDI chipset

16.2   Summary of sensors

---

[1] Search for the COMMTIMEOUTS structure for details.

16.3   The unified sensor interface

When implementing a new sensor class, the following execution flow must be kept in mind:

1. Object constructor: Do here basic initialization only. Parameters are still not set (see next step), thus communications must not be set up at this point.

2. `CGenericSensor::loadConfig`: Load here the parameters specific to your sensor. Notice that the application `rawlog-grabber` automatically loads the following parameters (common to all the sensors), thus they must be not loaded at this point:

    (a) "process_rate": The rate in Hertz (Hz) at which the sensor thread should invoke "doProcess". Mandatory parameter.

    (b) "max_queue_len": The maximum number of objects in the observations queue (default is 100). If overflow occurs, an error message will be issued at run-time.

3. `CGenericSensor::initialize`: Initialize here your connections, send initial commands to the device, etc.

4. `CGenericSensor::doProcess`: This method is called over and over again while the application is running. Your code must not delay too much and must always return, i.e. do not insert infinite loops. If a new piece of information from the sensor is gathered (which may not always occur), use the helper method `CGenericSensor::appendObservation` to add it to the "output queue". That is all `rawlog-grabber` expects from each sensor's class. Observations must be inserted in the list in the form of smart pointers (refer to Chapter 10).

16.4   How `rawlog-grabber` works

Chapter 17

Kalman filters

17.1 Introduction

17.2 Algorithms

17.3 How to implement a problem as a KF

The example bayesianTracking.

A more complicated model, the problem of 6D SLAM, is discussed in detail in [3] and implemented as the application `kf-slam` within MRPT.

Chapter 18

Particle filters

18.1 Introduction

A good tutorial can be found in [1].

18.2 Algorithms

18.2.1 SIR

18.2.2 Auxiliary PF

18.2.3 Optimal PF

18.2.4 Optimal-rejection sampling PF

The method presented in the paper [5].

18.3 Resampling schemes

18.4 Implementation examples

Bibliography

[1] M.S. Arulampalam, S. Maskell, N. Gordon, T. Clapp, D. Sci, T. Organ, and SA Adelaide. A tutorial on particle filters for online nonlinear/non-Gaussian Bayesian tracking. *IEEE Transactions on Signal Processing*, 50(2):174–188, 2002.

[2] Several authors. The Player project, 2008. `http://playerstage.sourceforge.net/`.

[3] J.L. Blanco. Derivation and Implementation of a Full 6D EKF-based Solution to Bearing-Range SLAM. Technical report, 2008.

[4] J.L. Blanco. The Mobile Robot Programming Toolkit (MRPT) website, 2008. http://mrpt.sourceforge.net/.

[5] J.L. Blanco, J. Gonzalez, and J.A. Fernández-Madrigal. An optimal filtering algorithm for non-parametric observation models in robot localization. In *IEEE International Conference on Robotics and Automation (ICRA'08)*, pages 461–466, May 2008.

[6] A.I. Eliazar and R. Parr. Learning probabilistic motion models for mobile robots. *ACM International Conference Proceeding Series*, 2004.

[7] Juan-Antonio Fernández-Madrigal. The babel development system for integrating heterogeneous robotic software. Technical report, System Engineering and Automation Dpt. - University of Málaga (Spain), July 2003.

[8] Juan-Antonio Fernández-Madrigal. The BABEL website, 2008. `http://babel.isa.uma.es/babel2/`.

[9] Michael Montemerlo, Nicholas Roy, Sebastian Thrun, Dirk Haehnel, Cyrill Stachniss, and Jared Glover. The CARMEN robot navigation toolkit, 2008. `http://carmen.sourceforge.net/`.

[10] Paul Newman et al. The MOOS website, 2008. `http://www.robots.ox.ac.uk/ pnewman/TheMOOS/`.

[11] Andy Rushton. The STLplus website, 2008. `http://stlplus.sourceforge.net/`.

[12] S. Thrun, W. Burgard, and D. Fox. *Probabilistic Robotics*. The MIT Press, September 2005.