RobWork – A Programming Primer

This collection of tutorials and exercises are provided to ease and aid the introduction to programming applications using RobWork. Currently the tutorial should be compatible with RobWork 0.5

**Table of contents**

[Introduction 1](#_Toc293571578)

[Exercises 2](#_Toc293571579)

[Tutorial 1 – My first cmake project 3](#_Toc293571580)

[Tutorial 2 – Math jogling 4](#_Toc293571581)

[Exercise 1 - Transformations 5](#_Toc293571582)

[Exercise 2 – Point transform 5](#_Toc293571583)

[Tutorial 3 – WorkCell, Devices and forward kinematics 5](#_Toc293571584)

[Exercise 1 – WorkCell 6](#_Toc293571585)

[Exercise 2 – Forward kinematics 6](#_Toc293571586)

[Exercise 3 – More Forward kinematics 6](#_Toc293571587)

[Tutorial 9 – RobWorkStudio plugin 7](#_Toc293571588)

[Exercise 1 – Updating the states 9](#_Toc293571589)

[Exercise 2 – Saving the state 10](#_Toc293571590)

# Introduction

RobWork is a framework/library for simulation and control of robotics with emphasis on industrial robotics and their applications.

The project was started in 2006 by Ph.D. students and master students at the Maersk Mc-Kinney Moeller Institute, University of Southern Denmark.

Today the project is more mature and used by both researchers, Ph.D. students, master students and robotics students. RobWork is mainly devided into 2 different parts. The basic framework named RobWork consist of basic mathematics used for robots and algorithms — e.g. motion planning and inverse kinematics. The second part of the framework named RobWorkHardware contains drivers and code for communicating with robots, cameras, canbus and others.

The major goal of the framework is to:

* Provide a single framework for offline and online robot programming including modelling, simulation and (realtime)control of robotics

The target users are:

* Researchers who needs a common framework for experimental robotics
* Students (you) who wants to experiment with the concepts of robotics
* Implementers of robot applications

RobWork is currently being used for research and student exercises at the University of Southern Denmark. You can read a lot more information at the RobWork website:

*http://www.robwork.dk.*

The following tutorials/exercises assume a functional installation of both RobWork and RobWorkStudio. Please go to the homepage for installation tutorials.

## Exercises

The exercises in the tutorials are often created with the intent of the user to retrieve information/functionality by using the API-DOC. This should enable the user to become more familiar with the base functionality of RobWork and also encurage the use of apidoc to find functionality. The apidoc is available on the homepage.

# Tutorial 1 – My first cmake project

This tutorial will demonstrate how to create a simple RobWork application using cmake.

First create a directory for your project. Create a “CMakeLists.txt” file and a “HelloWorld.cpp” in the directory. Insert the following cmake script into the CMakeLists.txt file:

The script should be pretty much self explaining. However, it is important that the RW\_ROOT is set to the correct paths of robwork.

CMAKE\_MINIMUM\_REQUIRED(VERSION 2.6.0)

# The name of the project. (EDIT THIS)

PROJECT(TutorialCollection)

# Used to resolve absolute path names

SET(ROOT ${CMAKE\_CURRENT\_SOURCE\_DIR})

##### IMPORTANT EDIT THESE PATHS TO REFLECT YOUR ROBWORK INSTALLATION #####

# Now set the RW/RWS root (edit this if necessary)

SET(RW\_ROOT "${ROOT}/../../../RobWork")

# set some robwork settings (EDIT THESE TO FIT YOUR ENVIRONMENT)

SET(CMAKE\_BUILD\_TYPE Release)

#Include default settings for constructing a robwork dependent project

SET(CMAKE\_MODULE\_PATH ${RW\_ROOT}/build ${CMAKE\_MODULE\_PATH})

FIND\_PACKAGE(RobWork)

LINK\_DIRECTORIES( ${ROBWORK\_LIBRARY\_DIRS} )

# if you have additional libraries or include dirs then add them here

INCLUDE\_DIRECTORIES(${ROOT} ${ROBWORK\_INCLUDE\_DIR})

# And now we add any targets that we want

add\_executable(HelloWorld HelloWorld.cpp)

target\_link\_libraries(HelloWorld ${ROBWORK\_LIBRARIES})

Now edit you “HelloWorld.cpp” file and type in a small main application.

This is very basic but the reader should notice two important aspects. Firstly, the header file “rw/common/Log.hpp” is included. This file includes only the Log class functionality which is the only class used in the code. Secondly the “using namespace” clause is used such that we can call “Log::infoLog()” instead of “rw::common::Log::infoLog()”.

**#include** <rw/common/Log.hpp>

**using** **namespace** rw::common;

**int** **main**(**int** argc, **char**\*\* argv) {

Log::*infoLog*() << "Hey, we are printing to the RobWork log!\n";

std::cout << "Which should just be standard out for now!" << std::endl;

}

When many different namespaces and classes from robwork are used, it can be somewhat tedious to write the “using namespace” and includes in every file. Instead a general header file “rw.hpp” and a macro can be used to assemble all classes into one namespace: “robwork”. We rewrite the code snippet from above:

**#include** <rw/rw.hpp>

USE\_ROBWORK\_NAMESPACE

**using** **namespace** robwork;

**int** **main**(**int** argc, **char**\*\* argv) {

Log::*infoLog*() << "The using namespace enables us to call Log directly!\n";

rw::common::Log::*infoLog*() << "We can still use the native namespace!\n";

robwork::Log::*infoLog*() << "but also the general namespace!\n";

}

Notice that when using this type friendly shortcut, the risk of name clashes between robwork classes and other libraries become much higher. Also, NEVER use “using namespace” in headerfiles, unless its within a function scope.

# Tutorial 2 – Math jogling

This tutorial will demonstrate some of the basic math functionality available in RobWork. This is mostly related to homogenous transformations, rotations, convertions and so on.

First add a new file “MathJogling.cpp” to your cmake project from tutorial 1. Make sure that the file is added as an executable in the end of the CMakeList.txt file.

# add another executable

add\_executable(MathJogling MathJogling.cpp)

target\_link\_libraries(MathJogling ${ROBWORK\_LIBRARIES})

Add the standard static main code body in the “MathJogling.cpp” and we are ready to play.

**#include** <rw/rw.hpp>

USE\_ROBWORK\_NAMESPACE

**using** **namespace** robwork;

**int** **main**(**int** argc, **char**\*\* argv) {

// main body, add your code here

}

The main use of the math package is homogenous transformations, rotations, vectors. However, before venturing into mathematical expressions we need to look at the different rotation representations. The most user friendly format is probably euler angles where RobWork use a fixed axis ZYX euler representation using the class “RPY” (Roll Pitch Yaw). The following snippet illustrates conversions between the rotation formats.

RPY<> rpy(0, 0, 90\*Deg2Rad); // 90 degree rotation around x-axis

Rotation3D<> rot = rpy.toRotation3D(); // create Rotation3D matrix

EAA<> eaa( rot ); // construct eaa form rotation3d

Quaternion<> quat( rot ); // construct quaternion from rotation3d

// there are streaming operators for all math types

Log::*infoLog*() << rpy << std::endl;

Log::*infoLog*() << rot << std::endl;

Log::*infoLog*() << eaa << std::endl;

Log::*infoLog*() << quat << std::endl;

Operators are used throughout the math package to enable intuitive math expressions and all so streaming as shown above. Now lets look at some of the most used functions in the math package.

// rotate a vector (0,1,0) 90 degrees around x-axis

Log::*infoLog*() << rot\*Vector3D<>(0,1,0) << std::endl;

// transform a vector

Transform3D<> t1( Vector3D<>(0,0,1), rot);

Log::*infoLog*() << t1\*Vector3D<>(0,1,0) << std::endl;

// calcualte the inverse rotation

Log::*infoLog*() << inverse( rot ) << std::endl;

// calculate the inverse transform

Log::*infoLog*() << inverse( t1 ) << std::endl;

// get the rotation and translation part of a transform

Log::*infoLog*() << t1.R() << t1.P() << std::endl;

## Exercise 1 - Transformations

Try to set up two transformations T1 and T2. Set T1 with a position (x; y; z) = (1; 1; 1) and rotation (r; p; y) = (90; 0; 0). Set T2 with a position (x; y; z) = (0; 0; 1) and rotation (r; p; y) = (0; 0; 0).

RobWork uses radians and not degrees. Take this into account.

Now calculate T1T2. What is the result? Construct a drawing of the frames (by hand)!

## Exercise 2 – Point transform

Now consider that T1 and T2 are frames in a robotic system. T2 is described relative to T1 and T1 relative to the world frame T0.

Using the two transformations from the previous exercise, calculate the position of pT2 = (0:5; 1; 0) with respect to frame T1 and T0

.

What is the result?

# Tutorial 3 – WorkCell, Devices and forward kinematics

Instead of hardcoding robotic descriptions and scene descriptions into C++ files, devices and scenes can be described using an XML file.

This section with exercises will guide you through the usage of these files. First extend the cmake project from tutorial 1 with a file “WorkCellAndDevices.cpp” and create the file with an empty main body.

Loading a WorkCell in RobWork is fairly simple:

WorkCell::Ptr wc = WorkCellLoader::load("SimpleWorkCell.wc.xml");

The typical functions of the workcell is to create a default State and add/remove/find frames or devices. The templated find functions are especially usefull if specific frame or device types need to be found.

Log::*infoLog*() << "Name of workcell: " << wc->getName() << std::endl;

// get the default state

State state = wc->getDefaultState();

Frame\* worldFrame = wc->getWorldFrame();

// find a frame by name, remember NULL is a valid return

Frame\* frame = wc->findFrame("FixedFrameName");

// find a frame by name, but with a specific frame type

FixedFrame\* fframe = wc->findFrame<FixedFrame>("FixedFrameName");

MovableFrame\* mframe = wc->findFrame<MovableFrame>("MovableFrameName");

// find a device by name

Device\* wc->findDevice("SerialDeviceName");

SerialDevice\* wc->findDevice<SerialDevice>("SerialDeviceName");

The basic building blocks of a workcell are Frames. These are ordered in a tree-structure where the ROOT node allways is the WORLD frame. All frames has a parent which thier position and orientation is relative to. Descending in this tree accumulating frame transformations is basically forward kinematics. The “Kinematics” class is a utility class for calculating forward kinematics, reattaching frames(gripping),

// calculate the transform from one frame to another

Transform3D<> fTmf = Kinematics::*frameTframe*(frame, mframe, state);

// calculate the transform from world to frame

Transform3D<> wTmf = Kinematics::*worldTframe*( mframe, state );

// we can find the world to frame transform by a little jogling

Transform3D<> wTf = wTmf \* inverse(fTmf);

// test if frame is a dynamic attachable frame

**if**( Kinematics::*isDAF*( \*mframe ) ){

// attach mframe to end of serial device

Kinematics::*gripFrame*(state, \*mframe, \*sdevice->getEnd() );

}

The device class also define utility functions for calculating forward kinematics, at least those that relate to the device. Additionally the Device has functionality to compute the Device Jacobian, setting and getting the joint configurations and getting the joint limits.

// get device base frame

Frame \*base = sdevice->getBase();

// get device end effector

Frame \*end = sdevice->getEnd();

// calculate base to end transform

Transform3D<> bTe = sdevice->baseTend(state);

// or just base to any frame

Transform3D<> bTmf = sdevice->baseTframe(mframe, state);

// get device name

std::string sdevicename = sdevice->getName();

// the degrees of freedom of this device

**int** dof = sdevice->getDOF();

// set the configuration of the device to zero

sdevice->setQ( Q::*zero*(dof) , state );

## Exercise 1 – WorkCell

Construct code, that load a workcell into your software and print the workcell name. Use the workcell to retrieve all the devices in the scene. Print the name of each device to the standard output (terminal).

To solve this exercise, you should consider the following classes: WorkCellLoader, Work-

Cell and Device.

## Exercise 2 – Forward kinematics

Using the code from the previous exercise, you have loaded a workcell into your program. Extract the first device of the workcell. Set the robot in a configuration of:

(0:451; 1:4; 0:976; 0:0; 0:76; 0:0). This configuration is given in radians.

Calculate the forward kinematics of the robot in the configuration given. You should only consider the robot base to end effector for this calculation.

## Exercise 3 – More Forward kinematics

Now, you should consider, that the robot might not be placed in a non-zero position and orientation relative to the world frame.

Calculate the transformation from world frame to end effector frame. Are the transformations equivalent? Why or why not?

For this exercise you should consider the class Kinematics.

# Tutorial 9 – RobWorkStudio plugin

This tutorial will go through the creation of a RobWorkStudio plugin using cmake. The plugin is very simple and for illustrative purposes includes two buttons with event handling.

Compared to tutorial 1, we need a bit more work. Specifically we need to create a class that implements the RobWorkStudioPlugin class interface. Let us start by defining the CMakeLists.txt file

CMAKE\_MINIMUM\_REQUIRED(VERSION 2.6.0)

# The name of the project.

PROJECT(SamplePluginApp)

# Used to resolve absolute path names

SET(ROOT ${CMAKE\_CURRENT\_SOURCE\_DIR})

# Now set the RW/RWS root (edit this if necessary)

SET(RW\_ROOT "${ROOT}/../../../RobWork")

SET(RWSTUDIO\_ROOT "${ROOT}/../../../RobWorkStudio")

# We use the settings that robwork studio uses

SET(CMAKE\_BUILD\_TYPE Release)

#Include default settings for constructing a robwork dependent project

INCLUDE(${RW\_ROOT}/build/FindRobWork.cmake)

#Include default settings for constructing a robworkstudio dependent project

INCLUDE(${RWSTUDIO\_ROOT}/build/FindRobWorkStudio.cmake)

INCLUDE\_DIRECTORIES( ${ROBWORK\_INCLUDE\_DIR} ${ROBWORKSTUDIO\_INCLUDE\_DIR} )

LINK\_DIRECTORIES( ${ROBWORK\_LIBRARY\_DIRS} ${ROBWORKSTUDIO\_LIBRARY\_DIRS} )

######################### From here we add the plugins

QT4\_WRAP\_CPP(MocSrcFiles SamplePlugin.hpp)

QT4\_ADD\_RESOURCES(RccSrcFiles resources.qrc)

# The shared library to build:

ADD\_LIBRARY(SamplePlugin MODULE SamplePlugin.cpp ${MocSrcFiles} ${RccSrcFiles})

TARGET\_LINK\_LIBRARIES(SamplePlugin ${ROBWORKSTUDIO\_LIBRARIES} ${ROBWORK\_LIBRARIES})

As can be seen we need to make three files.

*resources.qrc*: Is a QT resource file. We will not use this right now but later it will be used for embedding images/icons in your plugins and exefiles.

<!DOCTYPE RCC><RCC version="1.0">

<qresource>

<!--file>pa\_icon.png</file -->

</qresource>

</RCC>

*SamplePlugin.hpp*: The plugin header file declares the basic functionalities that we inherit from the RobWorkStudioPlugin interface. Beside the inherited interface we add two QPushButton pointers.

**#ifndef** SAMPLEPLUGIN\_HPP

**#define** SAMPLEPLUGIN\_HPP

**#include** <rw/rw.hpp>

**#include** <rws/RobWorkStudioPlugin.hpp>

**class** SamplePlugin: **public** rws::RobWorkStudioPlugin

{

Q\_OBJECT

**Q\_INTERFACES**( rws::RobWorkStudioPlugin )

**public**:

**SamplePlugin**();

**virtual** **~SamplePlugin**();

// functions inherited from RobworkStudioPlugin, are typically used but can be optional

**virtual** **void** **open**(rw::models::WorkCell\* workcell);

**virtual** **void** **close**();

**virtual** **void** **initialize**();

**private** slots:

**void** clickEvent();

**void** **stateChangedListener**(**const** rw::kinematics::State& state);

**private**:

QPushButton\* \_btn0,\*\_btn1;

};

**#endif** /\*SAMPLEPLUGIN\_HPP\*/

SamplePlugin.cpp: The plugin source file implements the basic functionality.

**#include** "SamplePlugin.hpp"

**#include** <QPushButton>

**#include** <RobWorkStudio.hpp>

USE\_ROBWORK\_NAMESPACE

**using** **namespace** robwork;

**using** **namespace** rws;

**SamplePlugin::SamplePlugin**():

RobWorkStudioPlugin("SamplePluginName", QIcon(":/pa\_icon.png"))

{

QWidget\* base = **new** QWidget(**this**);

QGridLayout\* pLayout = **new** QGridLayout(base);

base->setLayout(pLayout);

**this**->setWidget(base);

**int** row = 0;

\_btn0 = **new** QPushButton("Button0");

pLayout->addWidget(\_btn0, row++, 0);

connect(\_btn0, SIGNAL(clicked()), **this**, SLOT(clickEvent()));

\_btn1 = **new** QPushButton("Button1");

pLayout->addWidget(\_btn1, row++, 0);

connect(\_btn1, SIGNAL(clicked()), **this**, SLOT(clickEvent()));

pLayout->setRowStretch(row,1);

}

**SamplePlugin::~SamplePlugin**(){ /\* deallocate used memory \*/ }

**void** **SamplePlugin::open**(WorkCell\* workcell){ /\* do something when workcell is openned \*/}

**void** **SamplePlugin::close**() { /\* do something when the workcell is closed \*/}

**void** **SamplePlugin::initialize**() {

/\* do something when plugin is initialized \*/

getRobWorkStudio()->stateChangedEvent().add(

boost::bind(&SamplePlugin::stateChangedListener, **this**, \_1), **this**);

}

**void** **SamplePlugin::stateChangedListener**(**const** State& state) {

log().info() << "State changed!";

}

**void** **SamplePlugin::clickEvent**() {

QObject \*obj = sender();

**if**(obj == \_btn0){

log().info() << "Button 0 pressed!\n";

} **else** **if**(obj == \_btn1){

log().info() << "Button 1 pressed!\n";

}

}

Q\_EXPORT\_PLUGIN(SamplePlugin);

## Exercise 1 – Updating the states

Extend the plugin such that any device in scene will add 0.01 to all its joints when \_btn0 is pushed and substract 0.01 each time \_btn1 is pushed. Remember to update the RobWorkStudio state after changing it. In this exercise you should look at setState/getState on the RobWorkStudio interface and getRobWorkStudio() on RobWorkStudioPlugin interface.

## Exercise 2 – Saving the state

Extend the plugin example with two new buttons. One that saves the current state in a member variable of the plugin, and a second button that restores the saved state.