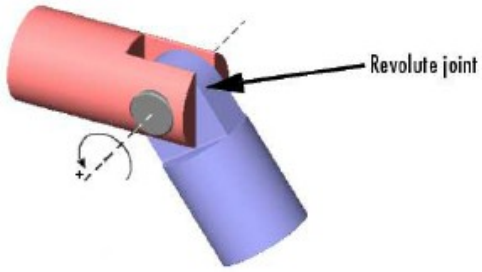
ROBOTIC ARM SIMULATION – DOCUMENTATION

1) FORWARD KINEMATICS

Introduction:

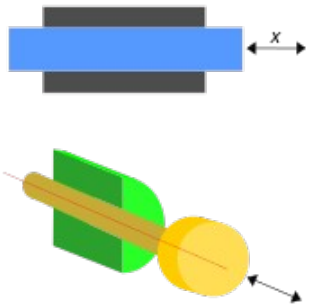
Robotic arms in more general terms can be referred to as kinematic chains. A kinematic chain is a combination of links that are connected together by joints. Joints are generally modelled to have a single degree of freedom. A degree of freedom, in the kinematics context, refers to the number of independent variables/co-ordinates necessary to completely describe the position of a system in space. In case a joint is seen to have more than one degree degree of freedom, we model that joint as a combination of multiple joints have one degree of freedom each. The different types of joints are as follows:

Revolute joint:



Revolute joints (also called pin joints or hinge joints) are single degree of freedom kinematic pairs, which can rotate (or hinge) about a single axis. Ex: in door hinges & folding mechanisms. They can be used as a passive or an active (i.e., driven by a motor) joint.

Prismatic joint



Like revolute joints, prismatic joints also have a single degree of freedom. However, the difference between the two lies in the type of motion they permit. Prismatic joints are used to define translational motion along an axis or sliding motion along an axis (hence sometimes referred to as a slider joint). Here rotational motion is completely restrained. Ex. Slider in a slider-crank mechanisms. They can be used as both passive and active joints (i.e., driven by a motor).

Joint value: In order to define a joint, we use a joint value. It can either be the angle by which rotation has taken place about the joint axis(in case of a revolute joint), or the amount of translation that has occurred along an axis (in case of a prismatic joint).

In the context of robotic arms, we need to figure out the valeus of the several joints that are present in the kinematic chain, so that the end effector ends up at a specific point in space. Thus, in forward kinematics, we determine where the end effector will be placed in 3D space given certain specific joint values for all the joints in the system under consideration. However, this doesn't mean we don't consider the positions of the other links. (For example, we need to ensure that the elbow of a robotic arms doesn't collide with any obstacle in the process of getting the end effector to a certain position in space). Each link has an associated coordinate frame associated with it. We also need to determine what the transform is from the base link of the robot to the end effector link of the robot. In most cases, industrial robots have joint encoders, and thus the robot's firmware will let the user knowthe joint values.

In summary for FK,

- **Input:** Joint angles

- **Output:** Co-ordinates of end effector / Transform from base to end effector

- Transforms for the links depends on the design of the link and therefore doesn't change as long as the design of the link doesn't change

- Tranforms for the joints depend on the joint angles, that continuously changes during run time of the robotic arm, and thus it is a variable.

INVERSE KINEMATICS:

In inverse kinematics, we will already know where the end effector of the robotic arms needs to be. In other words, we will already know what the transform from the base to the end effector must be. Using this, we will need to calculate the joint values. It is therefore, as the name suggests, the opposite of forward kinematics.

-**Input:** Co-ordinates of end effector / Transform from base to end effector

-**Output:** Joint angles

Robot description: There are two notations that we use to describe a robot:

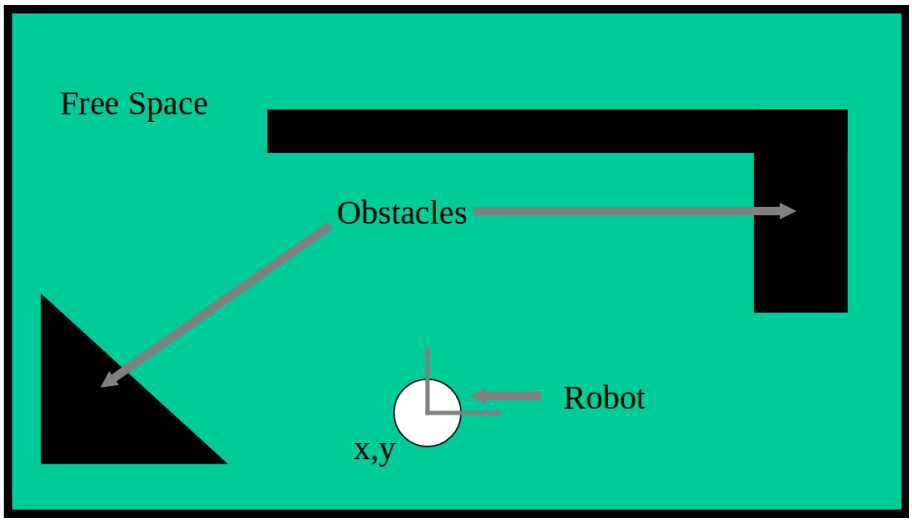
1) **URDF** – This is an abbreviation for universal robot description format, and is typically used in research. It is not as common in the industry. The reason is that since URDF is quite descriptive and lengthy, the storage space it consumes is a problem. Also, it supports solving for FK and IK using numerical techniques rather than analytical techniques. However, with increasing computational power and memory storage devices, this will not be a problem in the future, and we can see an increasing dominance of URDF.

2) **DH** – This is an abbreviation for Denavit-Hartenberg notation. This is used by industrial robots and is more compact in its representation and also supports analytical solutions to FK and IK problems. It works on a set of assumptions defined and known by the robot manufacturer and the user.

MOTION PLANNING

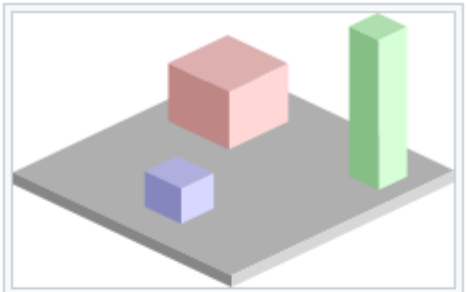
The world in which a robot is placed can be seen to be composed of two distinct features: (i) Obstacles (Ci)(i.e., regions of space in the world that have been occupied. In other words, a robot cannot go there) (ii) Free space (i.e., regions of space where a robot 'might' be able to go).

FS = W -(Uci)

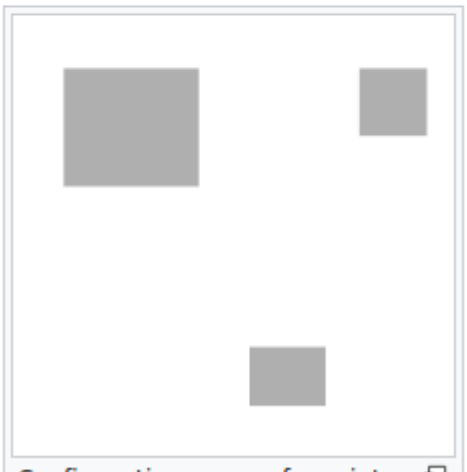


There are many different ways to measure how good a path is: (i) Time (ii) Distance (Iiii) Cost ..etc.

Motion planning is a process of breaking down a desired movement task into a set of smaller discrete motions, which satisfy a set of constraints and also optimise certain aspects of motion using parameters as discussed above (to define "how good" a path is).



Example of a workspace.



Configuration space of a point-sized robot. White = C_{free}, gray = C_{obs}.

The basic motion planning problem is to determine a continuous path between a starting point and the final destination to which the robot needs to go. The robot and obstacle geometry is described in a 2D/3D workspace and the path is described in a “configuration space”.

A “configuration” describes the pose of the robot and the “configuration space” describes the set of all possible poses that a robot can be in. There are several alogrithms which are used for motion planning including grid based search, interval based search, geometric algorithms, reward-based algorithms, A*, D* etc., each with a set of mertis and demerits. The selection of an algorithm depends on the complexity of the problem, computational power available, and time available to solve problem.

ROS CONCEPTS USED

What is ROS?

- It is a meta-operating system & software framework for programming robots
- Maintained by open source robotics foundation
-Consists of infrastructure, tools, capabilities & ecosystems

It has several advantages such as making it easier to build on work that other users have already done rather than creating the same application from scratch (saves you tremendous amount of time), it has a large community of users, and ROS is free & opensource.

ROS nodes:

-ROS nodes are single-purpose executable programs
-These nodes are individually compiled, executed and managed
-They are organized into packages

Packages: Any sofware we write in ROS needs to be organised into a package. Each package can contain libraries, executables, scripts, etc.

Manifests:A manifest is a description of a package. It defines dependencies between packages, and to capture meta-data about a package such as version, lisence, etc.

ROS topics: Streams of messages sent between nodes are called ROS topics and it is the mode by which several nodes communicate with one another. A node can “subscribe” to a topic (or receive data being sent) “published” by another node, or it can “publish” a topic (send data), that can be accessed by other nodes.
Examples of rostopic command line:

-rostopic list : lists the active topics
-rostopic echo /topic : print information about the contents of a topic
-rostopic type : print topic type
-rostopic hz : print publishing rate
-rostopic pub, etc.

ROS Master: It stores information about a network of nodes and manges the communication that occurs between several nodes.

Catkin workspace:

“catkin” is the official build system of ROS and has replaced original ROS build system, “rosbuild”. catkin combines CMake macros and Python scripts to provide some functionality on top of CMake's normal workflow. catkin was designed to be more conventional than rosbuild, allowing for better distribution of packages, better cross-compiling support, and better portability. catkin's workflow is very similar to CMake's but adds support for automatic 'find package' infrastructure and building multiple, dependent projects at the same time.

The package must contain a catkin compliant package.xml file:

- That package.xml file provides meta information about the package.
- The package must contain a CMakeLists.txt which uses catkin.
- If it is a catkin metapackage it must have the relevant boilerplate CMakeLists.txt file.
- Each package must have its own folder
- This means no nested packages nor multiple packages sharing the same directory.

- catkin_ws is organized into three distrinct folders: 1) Src (All executable nodes are placed here) 2) Devel (Executable files not, which aren’t tampered with) 3) Build (contains cache information and other intermediate files). The catkin_make command or

Roslaunch:

- It is a tool for launching multiple nodes at once (as well as setting parameters)
-Launch files are written in .xml format
-If not yet running, launch automatically starts a ROS core

Using roslaunch: roslaunch <package name> <launch_file>

Git: Git is a version control system which allows source code software developement such that multiple users can work on the same software and allows coordinating and controlling of changes made to the source code. As with most other distributed version-control systems, every Git directory on every computer is a full-fledged repository that contains the history and full version-tracking abilities. As a distributed revision-control system, it is aimed at speed, data integrity, and support for distributed, non-linear workflows.

Git command line:

git add <name of file>
git add . // adds all the files in the folder you are in
git commit -m “First commit” //create first commit
git push origin master //push the commit into master