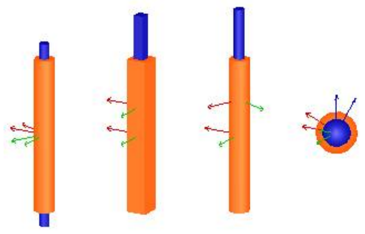
Joint types and operation

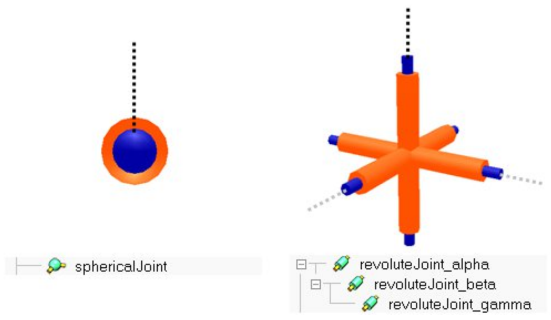
Compared to another object, a joint has two reference frames (visible only if the joint is selected). The first one is the regular reference frame that is fixed and that other objects also have. The second reference frame is not fixed, and will move relative to the first reference frame depending on the joint position (or joint value) that defines its configuration.

Joint types

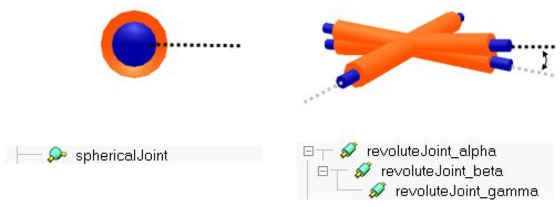
4 types of joints are supported:


[Revolute joint, prismatic joint, screw and spherical joint]

- **Revolute joints**: revolute joints have one DoF and are used to describe rotational movements (with 1 DoF) between objects. Their configuration is defined by one value that represents the amount of rotation about their first reference frame's z-axis. They can be used as passive joints, or as active joints (motors).
- **Prismatic joints**: prismatic joints have one DoF and are used to describe translational movements between objects. Their configuration is defined by one value that represents the amount of translation along their first reference frame's z-axis. They can be used as passive joints, or as active joints (motors).
- **Screws**: screws, which can be seen as a combination of revolute joints and prismatic joints (with linked values), have one DoF and are used to describe a movement similar to a screw. A pitch parameter defines the amount of translation for a given amount of rotation. A screw configuration is defined by one value that represents the amount of rotation about its first reference frame's z-axis. Screws can be used as passive joints, or as active joints (motors).
- **Spherical joints**: spherical joints have three DoF and are used to describe rotational movements (with 3 DoF) between objects. Their configuration is defined by three values that represent the amount of rotation around their first reference frame's x-, y- and z-axis. The three values that define a spherical joint's configuration are specified as Euler angles. In some situations, a spherical joint can be thought of as 3 concurrent and orthogonal to each other joints, that are parented in a hierarchy-chain. The analogy is however only valid while all revolute joints keep an orientation distinct from any of the two others: indeed, should two joints come close to coincide, a singular situation might appear and the mechanism might lose one DoF. This does never happen with spherical joints that are internally handled to avoid this kind of situation. Spherical joints are always passive joints, and cannot act as motors.



[Two equivalent mechanisms (in this configuration): spherical joint (left) and 3 revolute joints (right)]



[Two non-equivalent mechanisms: the right configuration is close to a singularity]

A joint is used to allow for a relative movement between its parent and its children. When a parent-child relationship is built between a joint and an object, the object is attached to the joint's second reference frame, thus, a change of the joint's configuration (intrinsic position) will directly be reflected onto its children. New joints can be added to a scene with [Menu bar --> Add --> Joints].

Joint modes

A joint can be in one of following modes:

- **Passive mode**: in this mode the joint is not directly controlled and will act as a fixed link. The user can however change the joint's position

with appropriate API function calls (e.g. sim.setJointPositon or sim.setSphericalJointMatrix).
- **Inverse kinematics mode**: in this mode, the joint acts as a passive joint, but is used (adjusted) during inverse kinematics calculations.
- **Dependent mode**: in this mode, the joint position is directly linked (dependent) to another joint position through a linear equation.
- **Motion mode**: this mode is deprecated and should not be used anymore. A similar and more flexible behaviour can be obtained with the **passive mode** and a child script appropriately updating the joint.
- **Torque or force mode**: in this mode, the joint is simulated by the dynamics module, if and only if it is dynamically enabled (refer to the section on designing dynamic simulations for more information). When dynamically enabled, a joint can be free or controlled in Force/torque, in velocity or in position. Screws cannot be operated in torque or force mode (however it is possible to obtain a similar behaviour by linking a revolute and prismatic joint programmatically), and spherical joints can only be free in torque or force mode.
  - When the joint motor is disabled, the joint is free and only constrained by its limits.
  - When the joint motor is enabled and the control loop is disabled, then the joint will try to reach the desired target velocity given the maximum torque/force it is capable to deliver. When that maximum torque/force is very high, the target velocity is instantaneously reached and the joint operates in velocity control, otherwise it operates at the specified torque/force until the desired target velocity is reached (torque/force control).
  - When the joint motor is enabled and the control loop is enabled, then the user has 3 control modes available:
    - *Custom control*: a joint callback function will be in charge of controlling the dynamic behaviour of the joint, allowing you to control the joint with any imaginable algorithm.
    - *Position control (PID)*: the joint will be controlled in position via a PID controller that will adjust the joint velocity in following way (the *Δt* divider is to keep the controller independent of the selected controller time step):

$$\text{Velocity} = \frac{K_p e_i + K_i \sum e_i \Delta t + K_d (e_i - e_{i-1})/\Delta t}{\Delta t}$$

$$\text{Force is constant}$$

    - *Spring-damper mode*: the joint will act like a spring-damper system via a force/torque modulation:

$$\text{Velocity is constant}$$

$$\text{Force} = K e_i + C(e_i - e_{i-1})/\Delta t$$

When the joint is in passive mode, inverse kinematics mode or dependent mode, it can optionally also be operated in a hybrid fashion: hybrid operation allows the joint to operate in a regular way, but additionally, just before dynamics calculations, the current joint position will be copied to the target joint position, and then, during dynamics calculations, the joint will be handled as a motor in position control (if and only if it is dynamically enabled (refer to the section on designing dynamic simulations for more information)). This feature allows for instance to control the leg of a humanoid robot by simply specifying the desired foot position (as an inverse kinematics task); the corresponding calculated joint positions will then be applied as position control values for the leg dynamic motion.

Joint controllers

There are many different ways a joint can be controlled. In following section, we differentiate betwen a ***loose*** controller and a ***precise*** controller: a ***loose*** joint controller will not be able to provide new control values in each possible regulation step (e.g. some regulation steps might/will be skipped, but control is still possible). A ***precise*** joint controller on the other hand, will be able to provide control values in each possible regulation step.

First, the approach to take for controlling a joint will depend on the joint mode:

- The joint is not in force/torque mode.
- The joint operates in force/torque mode.

The differentiation comes from the fact that a joint that operates in force/torque mode will be handled by the physics engine. And the physics engine will perform by default 10 times more calculation steps than the simulation loop: the simulation loop runs at 20Hz (in simulation time), while the physics engine runs at 200Hz (also in simulation time). That default behaviour can entirely be configured if required.

If the joint is not in force/torque mode: if the joint is not in force/torque mode, then you can directly (and instantaneously) set its position via the sim.setJointPosition API function (or similar, e.g. simxSetJointPosition for the B0-based remote API, or simxSetJointPosition for the legacy remote API). You can do this from a child script, from a plugin, from a ROS node, from a BlueZero node, or from a remote API client. If you do this from a child script, then it should be done inside of the *actuation section* of the non-threaded child script, or from a threaded child script that executes before the *sensing phase* of the main script (default). In the latter case however, make sure to have your threaded child script synchronized with the simulation loop for ***precise*** control.

In following threaded child script example, the joint is controlled ***loosely*** in position, and there is no synchronization with the simulation loop:

```
-- Following script should run threaded:

jointHandle=sim.getObjectHandle('Revolute_joint')

sim.setJointPosition(jointHandle,90*math.pi/180) -- set the position to 90 degrees
sim.wait(2) -- wait 2 seconds (in simulation time)
sim.setJointPosition(jointHandle,180*math.pi/180) -- set the position to 180 degrees
sim.wait(1) -- wait 1 second (in simulation time)
sim.setJointPosition(jointHandle,0*math.pi/180) -- set the position to 0 degrees
etc.
```

In following threaded child script example, the joint is controlled **precisely** in position in each simulation step, i.e. the thread is synchronized with the simulation loop:

```
-- Following script should run threaded:

sim.setThreadSwitchTiming(200) -- Automatic thread switching to a large value (200ms)
jointHandle=sim.getObjectHandle('Revolute_joint')

sim.setJointPosition(jointHandle,90*math.pi/180) -- set the position to 90 degrees
sim.switchThread() -- the thread resumes in next simulation step (i.e. when t becomes t+dt)
sim.setJointPosition(jointHandle,180*math.pi/180) -- set the position to 180 degrees
sim.switchThread() -- the thread resumes in next simulation step
sim.setJointPosition(jointHandle,0*math.pi/180) -- set the position to 0 degrees
sim.switchThread() -- the thread resumes in next simulation step
```

```
-- etc.

-- In above code, a new joint position is applied in each simulation step
```

When you try to control a joint that is not in force/torque mode from an external application (e.g. via the remote API, ROS or BlueZero), then the external controller will run asynchronously to V-REP (i.e. similar to the non-synchronized code of a threaded child script). This is fine most of the time for ***loose*** control, but if you wish to control the position of the joint ***precisely*** in each simulation loop, you will have to run V-REP in synchronous mode, and the external controller (e.g. the remote API client) will have to trigger each simulation step explicitely.

Following illustrates a C++ B0-based remote API client that does this:

```
bool doNextStep=false;

void simulationStepDone_CB(std::vector<msgpack::object>* msg)
{
    doNextStep=true;
}

int main(int argc,char* argv[])
{
    ...
    client.simxSynchronous(true); // enable the synchronous mode
    client.simxGetSimulationStepDone(client.simxDefaultSubscriber(simulationStepDone_CB)); // callback when step finished
    client.simxStartSimulation(client.simxDefaultPublisher()); // start the simulation

    client.simxSetJointPosition(jointHandle,90.0f*3.1415f/180.0f,client.simxDefaultPublisher()); // set the joint to 90 degrees
    client.simxSynchronousTrigger(); // start one simulation step
    while (!doNextStep) // wait until simulation step finished
        client.simxSpinOnce();

    doNextStep=false;
    client.simxSetJointPosition(jointHandle,180.0f*3.1415f/180.0f,client.simxDefaultPublisher()); // set the joint to 180 degrees
    client.simxSynchronousTrigger(); // start one simulation step
    while (!doNextStep) // wait until simulation step finished
        client.simxSpinOnce();

    doNextStep=false;
    client.simxSetJointPosition(jointHandle,0.0f*3.1415f/180.0f,client.simxDefaultPublisher()); // set the joint to 0 degrees
    client.simxSynchronousTrigger(); // start one simulation step
    while (!doNextStep) // wait until simulation step finished
        client.simxSpinOnce();
    ...
}
```

Refer to this page for details on how the B0-based remote API synchronous mode operates exactly. The approach is similar with ROS or BlueZero.

Following does the same, however with a legacy remote API client:

```
...
simxSynchronous(clientId,1); // enable the synchronous mode (client side). The server side (i.e. V-REP) also needs to be enabled.
simxStartSimulation(clientId,simx_opmode_oneshot); // start the simulation
simxSetJointPosition(clientId,jointHandle,90.0f*3.1415f/180.0f,simx_opmode_oneshot); // set the joint to 90 degrees
simxSynchronousTrigger(clientId); // trigger next simulation step. Above commands will be applied
simxSetJointPosition(clientId,jointHandle,180.0f*3.1415f/180.0f,simx_opmode_oneshot); // set the joint to 180 degrees
simxSynchronousTrigger(clientId); // next simulation step executes. Above commands will be applied
simxSetJointPosition(clientId,jointHandle,0.0f*3.1415f/180.0f,simx_opmode_oneshot); // set the joint to 0 degrees
...
```

Refer to this page for details on how the legacy remote API synchronous mode operates exactly. The approach is similar with ROS or BlueZero.

If the joint is in force/torque mode: if the joint operates in force/torque mode and is dynamically enabled, then it will be indirectly handled by the physics engine. If your joint's motor is not enabled, then your joint is not controlled (i.e. it will be free). Otherwise, your joint can be in following two dynamic modes:

- The joint's motor is enabled, but the control loop is disabled. Use this mode when you want to ***precisely*** custom control your joint from an external application (e.g. force/torque control, PID, etc.). Use this mode also, when you want to ***loosely*** control your joint in force/torque mode, or for velocity control (e.g. robot wheel motors).
- The joint's motor is enabled, and the control loop is enabled. Use this mode when your joint needs to act as a spring/damper, or if you want to ***precisely*** custom control your joint from within V-REP, or if you want to ***loosely*** control your joint in position control from an external application.

If your **joint's motor is enabled, but the control loop is disabled**, then the physics engine will apply the specified **Maximum force/torque**, and accelerate the joint until the **target velocity** is reached. If the load is small and/or the maximum force/torque high, that target velocity will be reached quickly. Otherwise, it will take some time, or, if the force/torque is not large enough, the target velocity will never be reached. You can programmatically adjust the target velocity with sim.setJointTargetVelocity (or for example, in case of the B0-based remote API: simxSetJointTargetVelocity, or, in case of the legacy remote API: simxSetJointTargetVelocity), and the maximum force/torque with sim.setJointForce (or for example, in case of the B0-based remote API: simxSetJointForce, or in case of the the legacy remote API: simxSetJointForce). You should be very careful before writing a ***precise*** joint controller for a joint in force/torque mode from a child script for following reason:

By default, the simulation loop runs with a time step of 50ms (in simulation time). But the physics engine will run with a time step of 5ms, i.e. 10 times more often. A child script will be called in each simulation step, but not in each physics engine calculation step. This means that if you control a joint from a child script in a *regular way*, you will only be able to provide new control values once for 10 physics engine calculation steps: you will be missing 9 steps. One way to overcome this would be to change the default simulation settings and to specify a simulation **time step** of 5ms, instead of 50ms. This works fine, but remember that all other calculations (e.g. vision sensors, proximity sensors, distance calculations, IK, etc.) will also run 10 times more often, and finally slow down your simulation (most of the time you won't need such a high refresh rate for the other calculation modules. But the physics engine requires such a high refresh rate). Another, much better option, would be to use a joint callback function (or a dynamics callback function) as will be explained further down.

If, one the other hand, you want to run a ***precise*** and regular joint controller externally (e.g. from a remote API client, a ROS node or a BlueZero node), then you have no other option than to set the simulation loop to the same rate as the physics engine rate, then run V-REP in synchronous mode, and the external controller (e.g. the remote API client) will have to trigger each simulation step explicitely.

Following illustrates a C++ B0-based remote API client that does this:

```
bool doNextStep=false;

void simulationStepDone_CB(std::vector<msgpack::object>* msg)
{
    doNextStep=true;
}

int main(int argc,char* argv[])
{
    ...
    client.simxSynchronous(true); // enable the synchronous mode
    client.simxGetSimulationStepDone(client.simxDefaultSubscriber(simulationStepDone_CB)); // callback when step finished
    client.simxStartSimulation(client.simxDefaultPublisher()); // start the simulation

    // set the desired force and target velocity:
    client.simxSetJointForce(jointHandle,1.0f,client.simxDefaultPublisher());
    client.simxSetJointTargetVelocity(jointHandle,180.0f*3.1415f/180.0f,client.simxDefaultPublisher());
    client.simxSynchronousTrigger(); // start one simulation step
    while (!doNextStep) // wait until simulation step finished
        client.simxSpinOnce();

    doNextStep=false;
    // set the desired force and target velocity:
    client.simxSetJointForce(jointHandle,0.5f,client.simxDefaultPublisher());
    client.simxSetJointTargetVelocity(jointHandle,180.0f*3.1415f/180.0f,client.simxDefaultPublisher());
    client.simxSynchronousTrigger(); // start one simulation step
    while (!doNextStep) // wait until simulation step finished
        client.simxSpinOnce();

    doNextStep=false;
    // set the desired force and target velocity:
    client.simxSetJointForce(jointHandle,2.0f,client.simxDefaultPublisher());
    client.simxSetJointTargetVelocity(jointHandle,180.0f*3.1415f/180.0f,client.simxDefaultPublisher());
    client.simxSynchronousTrigger(); // start one simulation step
    while (!doNextStep) // wait until simulation step finished
        client.simxSpinOnce();
    ...
}
```

Refer to this page for details on how the B0-based remote API synchronous mode operates exactly. The approach is similar with ROS or BlueZero.

Following does the same, however with a legacy remote API client:

```
...
simxSynchronous(clientId,1); -- enable the synchronous mode (client side). The server side (i.e. V-REP) also needs to be enabled.
simxStartSimulation(clientId,simx_opmode_oneshot); // start the simulation
simxSetJointForce(clientId,jointHandle,1.0f,simx_opmode_oneshot); // set the joint force/torque
simxSetJointTargetVelocity(clientId,jointHandle,180.0f*3.1415f/180.0f,simx_opmode_oneshot); // set the joint target velocity
simxSynchronousTrigger(clientId); // trigger next simulation step. Above commands will be applied
simxSetJointForce(clientId,jointHandle,0.5f,simx_opmode_oneshot); // set the joint force/torque
simxSetJointTargetVelocity(clientId,jointHandle,180.0f*3.1415f/180.0f,simx_opmode_oneshot); // set the joint target velocity
simxSynchronousTrigger(clientId); // next simulation step executes. Above commands will be applied
simxSetJointForce(clientId,jointHandle,2.0f,simx_opmode_oneshot); // set the joint force/torque
simxSetJointTargetVelocity(clientId,jointHandle,180.0f*3.1415f/180.0f,simx_opmode_oneshot); // set the joint target velocity
...
```

Refer to this page for details on how the legacy remote API synchronous mode operates exactly. The approach is similar with ROS or BlueZero.

If your **joint's motor is enabled, and the control loop is also enabled**, then the physics engine will handle the joint according to the setting: your joint can operate in position control (i.e. PID control), in a spring/damper mode, or in custom control. PID and spring/damper parameters can be updated from a child script, from a remote API client, from a ROS or BlueZero node. Refer to object parameter IDs 2002-2004, and 2018-2019. Desired target positions can be set with sim.setJointTargetPosition (or, for example, from a B0-based remote API client: simxSetJointTargetPosition, or from a legacy remote API client: simxSetJointTargetPosition). When you need a ***precise*** custom controller, then you should use a joint callback function instead (or a dynamics callback function).

Finally, if you need a precise PID or custom controller that is implemented in an external application, you need to make sure that the simulation step is the same as the physics engine calculation step: by default, V-REP's simulation loop runs at 20Hz (in simulation time), while the physics engine runs at 200Hz. You can adjust the simulation step size in the simulation setting. You also need to make sure you run V-REP in synchronous mode.

Following illustrates a C++ B0-based remote API client that does this:

```
bool doNextStep=false;

void simulationStepDone_CB(std::vector<msgpack::object>* msg)
{
    doNextStep=true;
}

int main(int argc,char* argv[])
{
    ...
    client.simxSynchronous(true); // enable the synchronous mode
    client.simxGetSimulationStepDone(client.simxDefaultSubscriber(simulationStepDone_CB)); // callback when step finished
    client.simxStartSimulation(client.simxDefaultPublisher()); // start the simulation

    // set the desired target position:
    client.simxSetJointTargetPosition(jointHandle,90.0f*3.1415f/180.0f,client.simxDefaultPublisher());
    client.simxSynchronousTrigger(); // start one simulation step
    while (!doNextStep) // wait until simulation step finished
        client.simxSpinOnce();

    doNextStep=false;
    // set the desired target position:
    client.simxSetJointTargetPosition(jointHandle,180.0f*3.1415f/180.0f,client.simxDefaultPublisher());
    client.simxSynchronousTrigger(); // start one simulation step
    while (!doNextStep) // wait until simulation step finished
        client.simxSpinOnce();
    doNextStep=false;
    // set the desired target position:
    client.simxSetJointTargetPosition(jointHandle,0.0f*3.1415f/180.0f,client.simxDefaultPublisher());
    client.simxSynchronousTrigger(); // start one simulation step
    while (!doNextStep) // wait until simulation step finished
        client.simxSpinOnce();
    ...
}
```

Following does the same, however with a legacy remote API client:

```
    ...
    simxSynchronous(clientId,1); -- enable the synchronous mode (client side). The server side (i.e. V-REP) also needs to be enabled.
    simxStartSimulation(clientId,simx_opmode_oneshot); // start the simulation
    simxSetJointTargetPosition(clientId,jointHandle,90.0f*3.1415f/180.0f,simx_opmode_oneshot); // set the desired joint position
    simxSynchronousTrigger(clientId); // trigger next simulation step. Above commands will be applied
    simxSetJointTargetPosition(clientId,jointHandle,180.0f*3.1415f/180.0f,simx_opmode_oneshot); // set the desired joint position
    simxSynchronousTrigger(clientId); // next simulation step executes. Above commands will be applied
    simxSetJointTargetPosition(clientId,jointHandle,0.0f*3.1415f/180.0f,simx_opmode_oneshot); // set the desired joint position
    ...
```

You can also have a remote API client provide control values for a custom joint controller implemented in a joint callback function, by providing values, for instance via signals, to that joint callback function. For example, from a C++ B0-based remote API client:

```
    bool doNextStep=false;

    void simulationStepDone_CB(std::vector<msgpack::object>* msg)
    {
        doNextStep=true;
    }

    int main(int argc,char* argv[])
    {
        ...
        client.simxSynchronous(true); // enable the synchronous mode
        client.simxGetSimulationStepDone(client.simxDefaultSubscriber(simulationStepDone_CB)); // callback when step finished
        client.simxStartSimulation(client.simxDefaultPublisher()); // start the simulation

        // set the desired target position:
        simxSetFloatSignal("myDesiredTorque",1.0f,client.simxDefaultPublisher());
        simxSetFloatSignal("myDesiredTarget",90.0f*3.1415/180.0f,client.simxDefaultPublisher());
        client.simxSynchronousTrigger(); // start one simulation step
        while (!doNextStep) // wait until simulation step finished
            client.simxSpinOnce();
        ...
    }
```

In above example, your joint callback function could fetch those two signals (with sim.getFloatSignal) before doing the control.

Following does the same, however with a legacy remote API client:

```
    ...
    simxSynchronous(clientId,1); -- enable the synchronous mode (client side). The server side (i.e. V-REP) also needs to be enabled.
    simxStartSimulation(clientId,simx_opmode_oneshot); // start the simulation
    simxSetFloatSignal(clientId,"myDesiredTorque",1.0f,simx_opmode_oneshot); // set the signal value
    simxSetFloatSignal(clientId,"myDesiredTarget",90.0f*3.1415/180.0f,simx_opmode_oneshot); // set the signal value
    simxSynchronousTrigger(clientId); // trigger next simulation step. Above commands will be applied
    ...
```

Recommended topics

- Joints
- Joint properties
- Inverse kinematics module
- Dynamics module
- Designing dynamic simulations