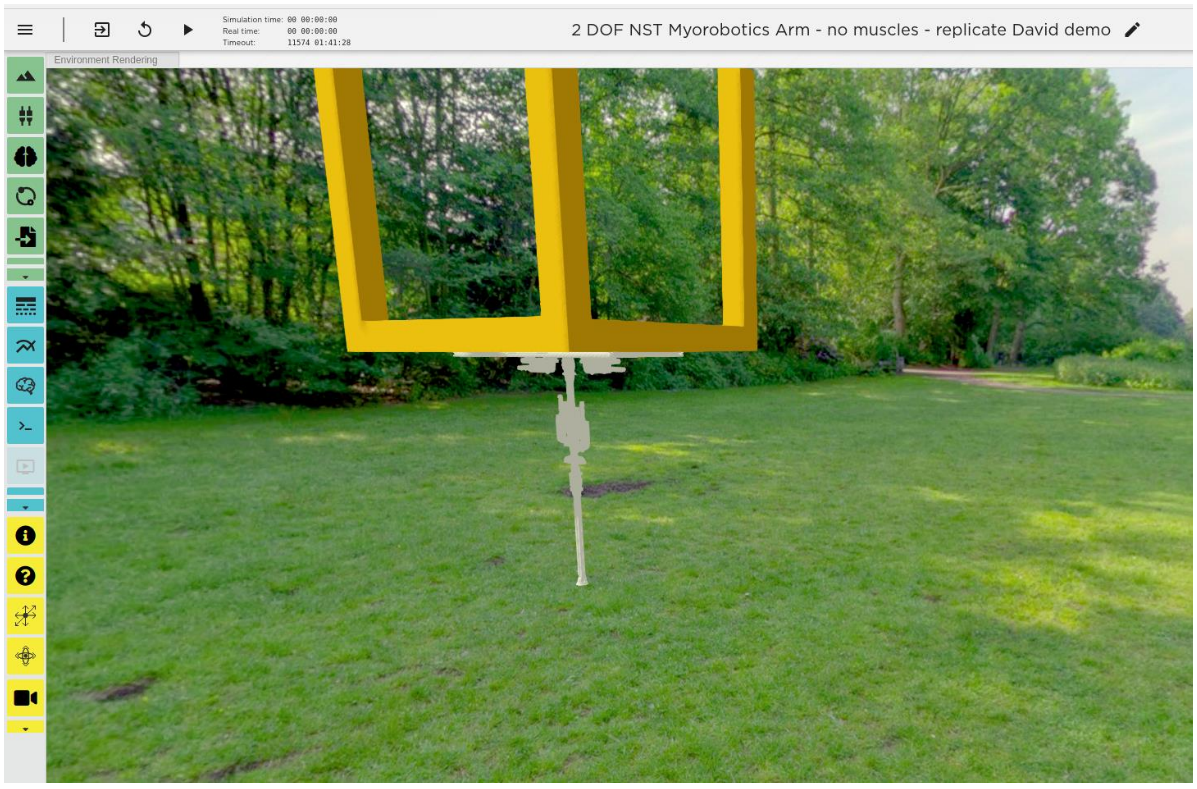
User Documentation: Mode Extraction

Introduction

In order to make sufficient use of compliant elements in robots, we developed a simple, yet energy efficient controller. The controller is able to drive multiple actuated joints with one control signal, while automatically adapting the weights to the relative forces of each actuator to match the ever-changing mechanical conditions of the robot and its environment. The controller is able to extract the eigenfrequencies of the mechanical system and drives the motion along this frequencies. In this way, the intrinsic motions of the system are excited leading to a close to optimal energy efficiency. This manual explains the implementation of this controller in the Neurorobotics Platform and how to use it.

Setup

Exemplary, the controller is implemented in the Myoarm, controlling its two joints. After cloning and launching the experiment, the following setup should show up:


Simulation time: 00 00:00:00
Real time: 00 00:00:00
Timeout: 11574 01:41:28
2 DOF NST Myorobotics Arm - no muscles - replicate David demo
Environment Rendering

TF Definitions

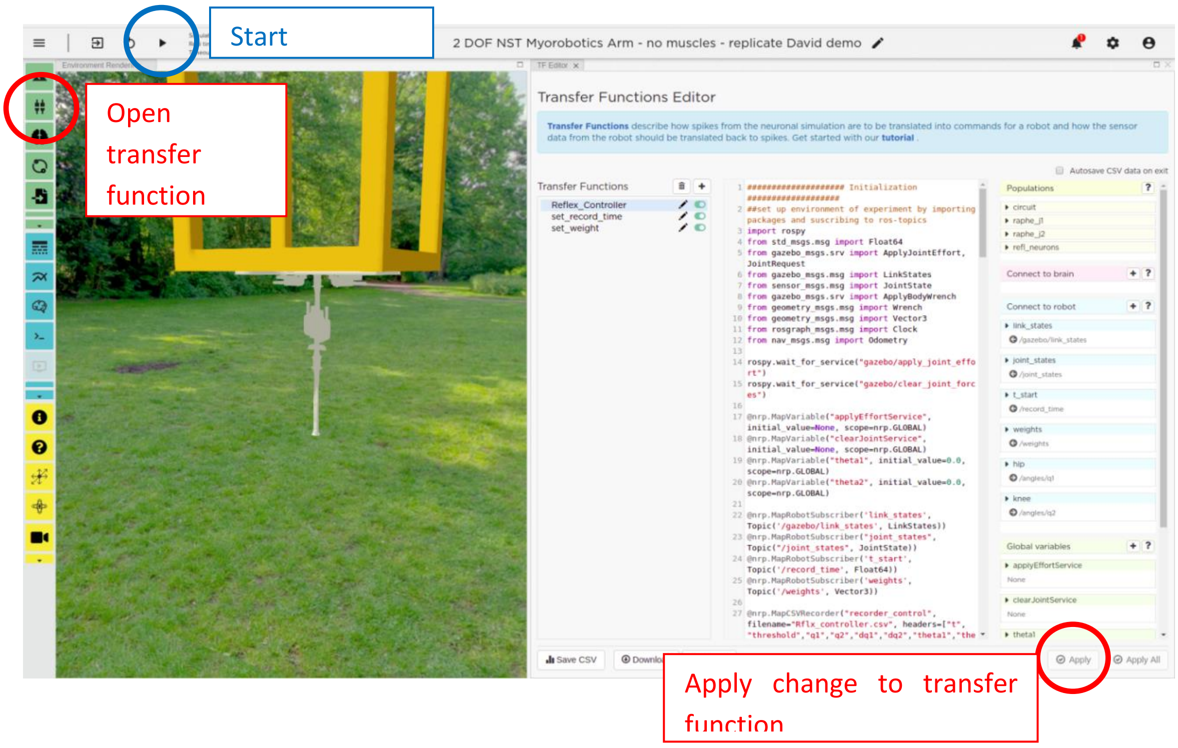
To open the transfer function, click the second green button from the top. A panel opens, in which the transfer functions defined in the .bibi-file can be found. In the Default Setup, the three functions that show up are:

- `Reflex_Controller`: defines how the control signal is applied to robot
    - Inputs:
        - `weights`: to tune control signal for different joints (calculated in `set_weight` and published through ROS)
        - `joint_states`: to read out joint positions to calculate torques in joints (Default gazebo topic, /gazebo/joint_states, alternative: read out `joint effort`)
        - OPTIONAL: `link_states`: to read out coordinates of links, e.g. to look at body height (Default gazebo topic: /gazebo/link_states)
        - OPTIONAL : `t_start`: start time to start recording when saving .csv –file (defined in `set_record_time` and published through ROS)
    - Outputs:
        - `apply_joint_effort` sending torques to the defined joints (tau1, tau2) (ROS service: /gazebo/apply_joint_effort)

- `set_weight`: define how the weights are adjusted (Default: 3 → Oja Rule)
    - Inputs:
        - `joint_states`: to read out joint positions to calculate weights with gradient descent method or Oja rule. (Default gazebo topic, /gazebo/joint_states)
        - OPTIONAL : `t_start`: start time to start recording when saving .csv –file (defined in `set_record_time` and published through ROS)
    - Outputs:
        - `weights`: needed to tune control signal for different joints (published through ROS as topic /weights)

- `set_record_time`: time from which a recording a recrding starts, when saving a .csv (helpful for debugging) (Default: 0, from beginning)
    - Inputs:
        - `NONE`
    - Outputs:
        - `t_start`: define start time to start recording when saving .csv –file (published through ROS as topic /record_time)

Use



Transfer Functions describe how spikes from the neuronal simulation are to be translated into commands for a robot and how the sensor data from the robot should be translated back to spikes. Get started with our tutorial .

```
##################### Initialization ###################
##set up environment of experiment by importing packages and suscribing to ros-topics
import rospy
from std_msgs.msg import Float64
from gazebo_msgs.srv import ApplyJointEffort, JointRequest
from gazebo_msgs.msg import LinkStates
from sensor_msgs.msg import JointState
from gazebo_msgs.srv import ApplyBodyWrench
from geometry_msgs.msg import Wrench
from geometry_msgs.msg import Vector3
from rosgraph_msgs.msg import Clock
from nav_msgs.msg import Odometry

rospy.wait_for_service("gazebo/apply_joint_effort")
rospy.wait_for_service("gazebo/clear_joint_forces")

@nrp.MapVariable("applyEffortService", initial_value=None, scope=nrp.GLOBAL)
@nrp.MapVariable("clearJointService", initial_value=None, scope=nrp.GLOBAL)
@nrp.MapVariable("theta1", initial_value=0.0, scope=nrp.GLOBAL)
@nrp.MapVariable("theta2", initial_value=0.0, scope=nrp.GLOBAL)

@nrp.MapRobotSubscriber('link_states', Topic('/gazebo/link_states', LinkStates))
@nrp.MapRobotSubscriber("joint_states", Topic("/joint_states", JointState))
@nrp.MapRobotSubscriber('t_start', Topic('/record_time', Float64))
@nrp.MapRobotSubscriber('weights', Topic('/weights', Vector3))

@nrp.MapCSVRecorder("recorder_control", filename="Rflx_controller.csv", headers=["t", "threshold","q1","q2","dq1","dq2","theta1","the
```

When opening the transfer function panel, on the left side of the panel all the available funtions and on the right side you see the code and can change them. Whenever you change something, make sure to press the "Apply" button in the right bottom corner.

The function `set_weight` defines if and with which method the weights applied the motor signals are determined to map the 1D signal to multiple joints. For more information on this, please refer to the *Tech Documentation* of this experiment.

[0]: the weighting is zero, so the control signal is scaled to zero.
[1]: the weights are set to a constant of 1 and do not adapt automatically; the sign (±) may vary depending on the employed model.
[2]: constant weights, but a difference of pi*alpha between the weights.
[3]: the weights are adapted using the Oja rule
[4]: the weights are adapted using the Gradient Descent method

To apply one of the options either change the transfer function, so that

```
weight_change = [number]
```

and click "Apply" in the right lower corner or use the ROS to publish the weight through the terminal:

```
rostopic pub /set_weight std_msgs/Int64 '[number]'
```

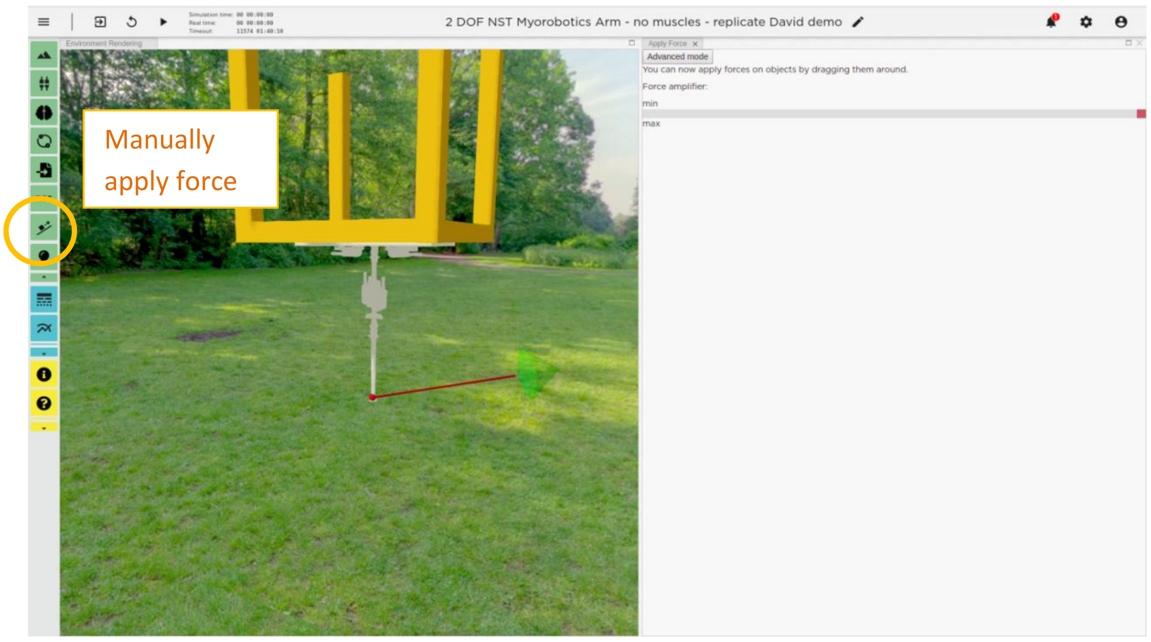
If needed, the time to start a recording to save as .csv-file can be set in the set_record_time transfer function by setting

```
t_start = [time]
```

To start the simulation press the "Play" button on the top.

When started, the arm will not start moving. To initiate the movement, you have to manually apply a force to the arm to trigger the controller, which is based on a threshold (for more information see *Tech Documentation*)

To apply a manual force press the button on the side panel with the sliding square (see below). A force panel opens. Slide the slider all the way to the right. Now click on the tick of the robot arm, keep the left mouse button pushed and pull to the side.



You can now apply forces on objects by dragging them around.

Force amplifier:

min

max

This will trigger the initial motion of the robot. The controller will now continue to drive the motion along its eigenfrequency and the weights will adapt to maximize the energy efficiency.

For more information on the working principle of the controller see the *Tech Documentation.*

IBA

Instead of using the transfer functions, it is also possible to use the IBA, which already implement the functions of set_weight.py and Reflex_Controller.py. The recording time cannot be manually set in this case. The inputs and outputs of the IBA functions:

- `FreqExtr_Module`: corresponds to `Reflex_Controller`
    - Inputs:
        - `joint_states`: same as TFs.
        - `Weights`: This is now sent via the IBA.
          The input array should look as follows:
            - At synced_data[0]: ID of sending module
            - At synced_data[1]: weight[0]
            - At synced_data[2]: weight[1]
            - At synced_data[3]: weight[2]
    - Outputs:
        - Same as the TFs, but also a float array inside the IBA called module_data.
          This array contains:
            - At module_data[0]: ID of module
            - At module_data[1]: Effort applied to "myoarm::HumerusBone_shoulder"
            - At module_data[2]: Effort applied to "myoarm::RadiusBone_ellbow"

- `setWeights_Module`: corresponds to `set_weight`
    - Inputs:
        - Same as the TFs.
    - Outputs:
        - Same as the TFs, but also a float array inside the IBA called module_data.
          This array contains:
            - At module_data[0]: ID of module
            - At module_data[1]: Weight[0]
            - At module_data[2]: Weight[1]
            - At module_data[3]: Weight[2]

For the [number] the values correspond to the following settings:

[0]: the weighting is zero, so the control signal is scaled to zero.
[1]: the weights are set to a constant of 1 and do not adapt automatically
[2]: the weights are adapted using the Oja rule
[3]: the weights are adapted using the Gradient Descent method

ATTENTION:
Option "2" from the Transfer Function is left out, since it will not be relevant for most users