WBM-Library Documentation

Release 0.9beta

Martin Neururer

Jun 25, 2018

CONTENTS

CHAPTER

ONE

INTRODUCTION

The *Whole-Body Model Library* (WBML) is an open source library for Matlab and provides, outside of of Simulink, fast *dynamics computation methods* for prototyping and testing purposes. The methods of the library are mainly based on the open source C++ program `mexWholeBodyModel`, which is a Matlab executable subroutine (MEX) and provides for Matlab a high-level interface to the `yarpWholeBodyInterface` for YARP-based robots, such as the iCub humanoid robot.

The `yarpWholeBodyInterface`[1] is a fast C++ interface to a *Whole-Body Abstraction Layer* for floating-base robots with YARP that contains subinterfaces for the *kinematic/dynamic model*, *actuators*, *sensor measurements* and *state estimations*.

The WBM-Library is an object-oriented extension for the Matlab MEX whole-body model interface, which was initially developed and supported by the FP7 EU-project *CoDyCo* (www.codyco.eu) at the *Istituto Italiano di Tecnologia* (IIT) in Genoa in Italy.

Currently, the documentation covers only the most important classes of the WBM-Library. The *utility functions* and the *interface classes* have not yet been documented in the library. The library documentation will be gradually extended over time. The function names of the utility functions are mostly oriented on the function names of the *Coordinate System Transformations*[2] of the *Robotics System Toolbox* of Matlab.

The additional interface classes of the WBM-Library ensures the ease of integration in other frameworks or research projects. The library offers two implemented interface classes:

- `iCubWBM` – An interface for the *iCub humanoid robot* for general purposes.
- `MultChainTree` – A special adapted interface for the *Robotics Toolbox* of *Peter Corke* (www.petercorke.com). With this special interface, the iCub robot can be easily integrated into research projects that are using for their experiments fixed-base robots of the toolbox. The method names and input arguments of the interface class are almost the same as the methods of the `SerialLink` class of the Robotics Toolbox.

1.1 Example

The given example in Listing 1.1 shows the basic usage of the WBM-Library. The robot model will be loaded from the given URDF description into the `yarpWholeBodyInterface` that is embedded in the MEX whole-body model interface (`mexWholeBodyModel`) for Matlab:

Listing 1.1: `examples/integrateForwardDynamics.m`

```
% namespaces:
import WBM.*
import WBM.utilities.ffun.*
import WBM.utilities.tfms.*
import WBM.RobotModel.iCub.*

```

(continues on next page)

[1] See https://github.com/robotology/yarp-wholebodyinterface.
[2] See https://www.mathworks.com/help/robotics/coordinate-system-transformations.html.

(continued from previous page)

```
%% Initialization of the WBM for the iCub robot:
wf2fixlnk    = true; % set the world frame to a fixed link
wbm_icub     = initRobotICub(wf2fixlnk);
icub_model   = wbm_icub.robot_model;
icub_config = wbm_icub.robot_config;

%% State variable:
%  Create the initial condition of the state variable "chi" for the integration of
%  the forward dynamics in state-space form. The state-space form reduces (through
%  variable substitution) the inhomogeneous second-order ODE to a first-order ODE.
chi_init = wbm_icub.init_stvChi;

%% Control torques:
%  Setup the time-dependent function "fhTrqControl" which describes a forcing
%  function on the ODEs to control the dynamics of the equation-system. It
%  refers to the control torques of each time-step t and is needed to calculate
%  the constraint (contact) forces which influences the outcome of each equation,
%  the generalized acceleration dv (ddot_q).
[p_b, R_b] = frame2posRotm(wbm_icub.init_vqT_base);
g_init = wbm_icub.generalizedBiasForces(R_b, p_b, icub_config.init_state_params.q_
↪j, zeros(icub_model.ndof,1), zeros(6,1));
len = size(g_init,1);

% Function handle (fh) for the torque controller function:
% Note: This function handle has only a very simple dummy-function as controller
%       (zero torques) that works in this case. For complex scenarios it is
%       advisable to use a real controller function instead, to avoid integration
%       errors.
fhTrqControl = @(t)zeroTrqsController(size(g_init(7:len,1)));

%% ODE-Solver:
%  Setup the function handle of the form f(t,chi), where chi refers to the dynamic
%  state of the system. It evaluates the right side of the nonlinear first-order
%  ODEs of the form chi' = f(t,chi) and returns a vector of rates of change
%  (vector of derivatives) which will be integrated by the solver.
fhFwdDyn = @(t, chi)fastForwardDynamics(t, chi, fhTrqControl, wbm_icub.robot_model,
↪ wbm_icub.robot_config);
%fhFwdDyn = @(t, chi)wbm_icub.forwardDynamics(t, chi, fhTrqControl); % optional

% specifying the time interval of the integration ...
sim_time.start = 0.0;
sim_time.end   = 2.0; %1.5;
sim_time.step  = 0.01;
tspan = sim_time.start:sim_time.step:sim_time.end;

disp('Start the numerical integration...');

ode_options = odeset('RelTol', 1e-2, 'AbsTol', 1e-4); % setup the error tolerances
[t, chi]    = ode15s(fhFwdDyn, tspan, chi_init, ode_options); % ODE-Solver
% or, optional:
%[t, chi] = fastIntForwardDynamics(fhTrqControl, tspan, chi_init, wbm_icub.robot_
↪model, wbm_icub.robot_config, ode_options);
%[t, chi] = wbm_icub.intForwardDynamics(tspan, chi_init, fhTrqControl, ode_
↪options);

save('testTrajectory.mat', 't', 'chi', 'chi_init', 'fhTrqControl', 'icub_model',
↪'icub_config');
disp('Numerical integration finished.');

noi = size(chi,1);
fprintf('Number of integrations: %d\n', noi);
```

(continues on next page)

(continued from previous page)

```
%% iCub-Simulator -- Setup the window, the environment and the draw parameters for
%  the WBM-simulator:

% create some geometric volume bodies for the simulation environment ...
rotm_r = eye(3,3); % rectangular orientation
rotm_2 = [-0.9      0  -0.1;
           0    -0.9      0;
          -0.1      0   0.9];

vb_objects       = repmat(WBM.vbCuboid, 3, 1);
vb_objects(1,1) = WBM.vbCuboid(0.1, [0.15; 0.10; 0.61], rotm_r);
vb_objects(2,1) = WBM.vbCylinder(0.1, 0.2, [-0.2; 0.4; 0.4], rotm_2);
vb_objects(3,1) = WBM.vbSphere(0.1, [-0.3; 0.3; 0.2], rotm_r);

show_light = true;
sim_config = initSimConfigICub(vb_objects, show_light); % shows the simulation␣
↪with a light scene as default.
%sim_config = initSimConfigICub(vb_objects, 'DarkScn', show_light); % optional,␣
↪shows the simulation with a dark scene.
sim_config = wbm_icub.setupSimulation(sim_config);

% Define some payload links and link each payload link to a specified volume body:
% Note: Payload links are usually links of a manipulator (end-effector) or in
%       special cases, links on which additionally special payloads are mounted
%       (e.g. battery pack or knapsack at torso, tools, etc.).
pl_lnk_l.name     = 'l_gripper'; % --> l_hand_dh_frame --> l_hand
pl_lnk_l.lnk_p_cm = [0; 0; -0.05];
pl_lnk_l.vb_idx   = 1;
%pl_lnk_l.m_rb     = sim_config.environment.vb_objects(1,1).m_rb; % optional
%pl_lnk_l.I_cm     = sim_config.environment.vb_objects(1,1).I_cm;

wbm_icub.setPayloadLinks(pl_lnk_l);

% setup the payload stack to be processed:
% (link the index of the volume body object to the left hand (manipulator))
sim_config.setPayloadStack(pl_lnk_l.vb_idx, 'lh');
% set the utilization time indices (start, end) of the object:
sim_config.setPayloadUtilTime(1, 1, 35);
% optional:
%pl_lnk_r.name     = 'r_gripper'; % --> r_hand_dh_frame --> r_hand
%pl_lnk_r.lnk_p_cm = [0; 0; 0.05];
%pl_lnk_r.m_rb     = sim_config.environment.vb_objects(2,1).m_rb;
%pl_lnk_r.I_cm     = sim_config.environment.vb_objects(2,1).I_cm;
%pl_lnk_r.vb_idx   = 2;

%pl_lnk_data = {pl_lnk_l, pl_lnk_r};
%wbm_icub.setPayloadLinks(pl_lnk_data);

% link each vb-index of the objects to one hand (manipulator) ...
%sim_config.setPayloadStack([pl_lnk_l.vb_idx, pl_lnk_r.vb_idx], {'lh', 'rh'});
% utilization time indices of the payloads ...
%sim_config.setPayloadUtilTime(1, 1, 35);
%sim_config.setPayloadUtilTime(2, 1, 33);

% get the positions data of the integration output chi:
x_out = wbm_icub.getPositionsData(chi);

% show the trajectory curves of some specified links:
lnk_traj = repmat(WBM.wbmLinkTrajectory, 3, 1);
lnk_traj(1,1).urdf_link_name = 'l_gripper';
```

(continues on next page)

(continued from previous page)

```
lnk_traj(2,1).urdf_link_name = 'r_gripper';
lnk_traj(3,1).urdf_link_name = 'r_lower_leg';

lnk_traj(1,1).jnt_annot_pos = {'left_arm', 7};
lnk_traj(2,1).jnt_annot_pos = {'right_arm', 7};
lnk_traj(3,1).jnt_annot_pos = {'right_leg', 4};

lnk_traj(1,1).line_color = WBM.wbmColor.forestgreen;
lnk_traj(1,1).ept_color  = WBM.wbmColor.forestgreen;
lnk_traj(2,1).line_color = WBM.wbmColor.tomato;
lnk_traj(2,1).ept_color  = WBM.wbmColor.tomato;
lnk_traj(3,1).line_color = 'magenta';
lnk_traj(3,1).ept_color  = 'magenta';

sim_config.trajectories = wbm_icub.setTrajectoriesData(lnk_traj, x_out, [1 1 1],␣
↪[35 35 40]);
sim_config.show_legend  = true;

% zoom and shift some specified axes (optional):
%sim_config.zoomAxes([1 4], [0.9 1.1]);   % axes indices, zoom factors (90%, 110%)
%sim_config.shiftAxes(4, [0 0.05 -0.12]); %             , shift vectors (x, y, z)

% show and repeat the simulation 2 times ...
nRpts = 2;
wbm_icub.simulateForwardDynamics(x_out, sim_config, sim_time.step, nRpts);

%% Plot the results -- CoM-trajectory:
wbm_icub.plotCoMTrajectory(x_out);

% get the visualization data of the forward dynamics integration for plots
% and animations:
vis_data = wbm_icub.getFDynVisData(chi, fhTrqControl);

% alternatively, or if you have to plot other parameter values, use e.g.:
%stp = wbm_icub.getStateParams(chi);

%figure('Name', 'iCub - CoM-trajectory:', 'NumberTitle', 'off');

%plot3(stp.x_b(1:noi,1), stp.x_b(1:noi,2), stp.x_b(1:noi,3), 'Color', 'b');
%hold on;
%plot3(stp.x_b(1,1), stp.x_b(1,2), stp.x_b(1,3), 'Marker', 'o', 'MarkerEdgeColor',
↪'r');

%grid on;
%axis square;
%xlabel('x_{xb}');
%ylabel('y_{xb}');
%zlabel('z_{xb}');
```

CHAPTER

TWO

THE WHOLE-BODY MODEL CLASSES

+WBM is the *namespace* (and main folder) for all classes and functions of the WBM-Library to avoid conflicts with other classes and functions in Matlab.

The namespace folder contains all classes for the *model*, *configuration* and *model state* of the robot. In addition, the folder provides also all necessary classes to create *graphics objects*, *volume bodies*, *target points* and *link trajectories* in a simulation scenario of a given robot model.

2.1 State, Model and Configuration Classes

class `+WBM.`**`wbmStateParams`**

`wbmStateParams` is a *data type* (class) to represent the state parameters of the current state of a given floating base robot.

`x_b`
double, matrix – Cartesian position of the base in Euclidean space $\mathbb{R}^3$.

`qt_b`
double, matrix – Orientation of the base in quaternions (global parametrization of $\mathbf{SO}^3$).

`q_j`
double, matrix – Joint positions (angles) of dimension $\mathbb{R}^{n_{dof}}$ in $[\mathrm{rad}]$.

`dx_b`
double, matrix – Cartesian velocity of the base in Euclidean space $\mathbb{R}^3$.

`omega_b`
double, matrix – Angular velocity describing the orientation of the base in $\mathbf{SO}^3$.

`dq_j`
double, matrix – Joint velocities of dimension $\mathbb{R}^{n_{dof}}$ in $[\mathrm{rad/s}]$.

Note: In dependency of the situation, the state variables can represent either a *single state* with only *column-vectors* or, a *time sequence of states* with only *matrices*, where each row-vector represents a state.

class `+WBM.`**`wbmSimEnvironment`**

Bases: `handle`

`wbmSimEnvironment` is a *data type* (class) that defines the environment for the robot simulation. It defines the background, the floor and the geometric volume bodies for the simulation environment.

`bkgrd_color_opt`
char, vector – Sets the color scheme for the axis background, axis lines and labels, and the figure background to one of the color options: `'white'`, `'black'` or `'none'`.

`grnd_shape`
double, matrix – Defines the shape of the ground-floor plane, as a polygon. The vertical X, Y and Z triplets, combined into a matrix, specify the vertices of the polygon.

grnd_color
: *double/char, vector* – Color of the ground, specified by a RGB-triplet or a color name.

grnd_edge_color
: *double/char, vector* – Edge color of the ground, specified by a RGB-triplet or a color name.

orig_pt_color
: *double/char, vector* – Color of the origin point on the floor (RGB-triplet or color name).

orig_pt_size
: *double, scalar* – Size of the origin point, specified as a positive value in points.

vb_objects
: `vbObject`, *vector* – Column-vector of *volume body objects* that representing an environment scenario for the simulation.

See also:

`wbmSimConfig` and `genericSimConfig`.

class +WBM.**wbmSimConfig**

Bases: `handle`

`wbmSimConfig` is an *abstract data type class* and represents the *base configuration interface* for the visualizer of the robot simulation.

The abstract class increases the flexibility and enables for the users to derive user specific configuration classes for the visualization of the robot.

custom_view
: *double, vector* – Custom viewpoint specification in terms of *azimuth* (az) and *elevation* (el) as a row-vector $[az, el]$.

axes_vwpts
: *cell, matrix* – List of predefined viewpoint positions for the orientation of the axes (*readonly*). The list is a 2-dimensional array where in the first column are the names of the axes orientations and in the second column the corresponding viewpoint vectors.

DF_WND_POS
: *double, vector* – Default window position of the form $[x$ (left), y (bottom)$]$, specified in *pixels* (*constant*).

DF_WND_SIZE
: *double, vector* – Default window size of the figure, specified as a vector of the form $[width, height]$ in *pixels* (*constant*).

DF_AXES_NBR
: *uint8, scalar* – Default number of axes graphics objects in the figure window, to divide the figure into a $(m \times n)$ grid (*constant*).

DF_AXES_POS
: *double, matrix* – Default positions of each axes graphics object in the figure window (*constant*).

Note: The axes positions must be specified in a $(n \times 4)$ matrix, where each row of the matrix represents an axes position of the form $[x$ (left), y (bottom), wid (width), hgt (height)$]$. The row number of the matrix must be the same as the number of axes.

DF_AXES_COLORS
: *double, matrix* – Default colors for the axes back planes and for the axis lines, tick values and labels (*constant*).

Note: The colors must be specified in a (4×3) matrix with horizontal RGB-triplets. The first triplet defines the color for the axes back planes and all other colors are for the axis lines tick values and labels in the x, y and z direction.

DF_AXES_VWPTS
: *cell, matrix* – List of default viewpoint positions to determine the orientation of each axes (*constant*).

Note: The list must be an 2-dimensional array with the viewpoint names in the first column and in the second column the specified viewpoints (row-vectors) with *azimuth* (az) and *elevation* (el) values as terms.

`DF_AXES_VIEWS`
cell, vector – Default row-array of string matching viewpoint names that shows the robot simulation at each axes with a different view (*constant*).

`DF_AXIS_LIMITS`
double, vector – Default (6×1) vector of the form $[x_{min}, x_{max}, y_{min}, y_{max}, z_{min}, z_{max}]$, to specify the axis limits of every Cartesian axes in the figure window (*constant*).

`DF_GROUND_SHAPE`
double, matrix – Default $(3 \times n)$ polygon matrix consisting of a set of vertical XYZ-triplets, which form the shape of the ground floor plane (*constant*).

`DF_GROUND_COLOR`
double, vector – Default color of the ground, specified by a RGB-triplet (*constant*).

`robot_body`
`wbmSimBody` – Data object to define the geometric shape for the body of the simulated robot.

`environment`
`wbmSimEnvironment` – Data object to define the environment settings for the robot simulation.

`trajectories`
`wbmLinkTrajectory`, *vector* – Array of trajectory objects to show the trajectory curves of specific links of the robot.

`target_pts`
`wbmTargetPoint`, *vector* – Array of target points that should reached in the simulation by specific links of the robot.

`hWndFigure`
`matlab.ui.Figure` – Figure window for the visualization of the robot simulation.

`wnd_title`
char, vector – Title (name) of the figure window.

`wnd_pos`
double, vector – Location of the left inner corner of the figure window with the origin at the left bottom corner of the primary display, specified as a row-vector of the form $[pos_{left}, pos_{bottom}]$ (default location: *`DF_WND_POS`*).

`wnd_size`
double, vector – Size of the figure window, specified as a row-vector of the form $[width, height]$ (default size: *`DF_WND_SIZE`*).

`show_wnd`
logical, scalar – Boolean flag to determine if the figure should be displayed on the screen. If the value is set to *false*, then the figure is after the simulation setup disabled and invisible on the screen. This can be helpful for performance reasons, e.g., to increase the speed of creating a video of the current simulation (default: *true*).

`nAxes`
uint8, scalar – Number of axes graphics objects in the figure window (default number: *`DF_AXES_NBR`*).

`hAxes`
double, vector – $(1 \times n_{axes})$ array with handles of the created axes objects.

`axes_pos`
double, matrix – Matrix with size and position of each axes within the figure window, specified as row-vectors of the form $[pos_{left}, pos_{bottom}, width, height]$ (default positions: *`DF_AXES_POS`*).

`axes_colors`
double, matrix – Colors for the axes back planes and for the axis lines, tick values and labels, specified by RGB-triplets as row-vectors (default colors: *`DF_AXES_COLORS`*).

`axes_views`
cell, vector – Row-array of different string matching viewpoint names to define the robot simulation at each axes in a distinctive view (default views: *`DF_AXES_VIEWS`*).

`axis_limits`
double, vector – (6×1) vector of the form $[x_{min}, x_{max}, y_{min}, y_{max}, z_{min}, z_{max}]$, to specify the axis limits of every Cartesian axes in the figure window (default limits: *`DF_AXIS_LIMITS`*).

`vwpts_annot`
cell, vector – Row-list with short annotation strings (or titles) for the given viewpoints in the figure window.

`gfx_objects`
cell, vector – $(1 \times n_{axes})$ array of *graphics object handles* as data container for each axes in the figure window to store, manipulate and visualize all graphics objects (environment, robot body, trajectories and target points) of the simulation.

`show_light`
logical, scalar – Boolean flag to enable a point light in the simulation environment that radiates from a specified location in all directions (default: *false*).

`light_pos`
double, vector – Location of the point light, specified by a Cartesian vector $[x, y, z]$.

`show_titles`
logical, scalar – Boolean flag to show the viewpoint titles in the left bottom corner of each axes (default: *true*).

`tit_font_sz`
double, scalar – Font size of the viewpoint titles, specified as a scalar numeric value.

`tit_font_color`
double/char, vector – Font color of the viewpoint titles, specified by a RGB-triplet or a color name.

`show_legend`
logical, scalar – Boolean flag to show in the figure window the legend for the link trajectories of the robot (default: *true*).

`lgd_font_sz`
double, scalar – Font size of the legend, specified as a scalar numeric value.

`lgd_font_color`
double/char, vector – Font color of the legend, specified by a RGB-triplet or a color name.

`lgd_bkgrd_color`
double/char, vector – Background color of the legend, specified by a RGB-triplet or a color name.

`lgd_edge_color`
double or char, vector – Edge color of the box outline of the legend, specified by a RGB-triplet or a color name.

`lgd_location`
char, vector – Location of the legend with respect to the axes. The location values are the same as listed in the table of the *legend properties* of Matlab.

`lgd_orient`
char, vector – Orientation of the legend, specified by one of these values: `'vertical'`, `'horizontal'`.

`mkvideo`
logical, scalar – Boolean flag to create a video of the current robot simulation (default: *false*).

Note: If the flag is enabled by setting the value to *true*, then the figure window will not shown on the screen, to speed up the creation of the video.

`vid_axes_idx`
uint8, scalar – Index of an axes graphics object in the figure window to capture an image or video only from a particular subplot frame. If the value is set to 0, then the entire figure window (all axes) will be captured for the image or video (default value: 0).

`vid_filename`
char, vector – The filename of the video with the `.avi` extension and the full path where the file will be written.

Note: Currently only the *uncompressed AVI* file format can be used for creating video files, because only this file format will be supported by Matlab on all platforms.

`vid_fps`
double, scalar – Frame rate of the video playback in frames per second [fps], specified by a positive number.

`pl_stack`
cell, matrix – Index stack (list) for the volume body objects that are defined as payloads for particular manipulators (hands) of the robot. Each payload object of the stack will be processed successively and is assigned to a manipulator of the robot, specified by one of these values:

- `'lh'`: Use the *left hand* as manipulator.
- `'rh'`: Use the *right hand* as manipulator.
- `'bh'`: Use *both hands* to manipulate the object.

Note: The payload stack is a 2-dimensional array. In the first column are listed the reference indices to the volume body objects of the environment and in the second column is each object linked to a particular manipulator (hand) for processing.

`pl_time_idx`
uint32, matrix – Utilization time matrix of the given payload objects in the stack to determine the points in time for the simulation when each object is grabbed and released by the specified manipulator.

Note: The rows of the index matrix representing time index vectors of the form, $[idx_{start}, idx_{end}]$, where idx_{start} denotes the time index when the object is grabbed and idx_{end} is the time index when the object is released by the manipulator.

`zoom_axes`
double, matrix – Zoom index matrix to specify which axes of the figure window should be zoomed by a particular zoom factor.

Note: Each row of the index matrix is of the form $[idx_{ax}, zfac]$, where idx_{ax} denotes the reference index of an axes graphics object in the figure and $zfac$ is the corresponding zoom factor. If $zfac > 1$, the scene appears larger (*zoom in*); if $zfac \geq 0$ and $zfac < 1$, the scene appears smaller (*zoom out*). If $zfac = 1$, then the scene keeps its original size. The row number of the matrix can grow at most n_{axes} long.

`shift_axes`
cell, matrix – Shift index array to specify which axes of the figure window should be shifted in a particular direction within the axis limits that are scaled to a given zoom factor. This is very useful if only a special area of the scene is of interest.

Note: The index array is a 2-element row-array of the form $\{idx_{ax}, v_{sh}\}$. The first element idx_{ax} of the array represents a column-vector with reference indices to the axes graphics objects in the figure window. The second element v_{sh} describes a $(n \times 3)$ matrix of Cartesian vectors of the form $[x, y, z]$. Each row-vector of the matrix specifies the *shift direction* of the scene of the current axes. The row numbers of both array elements must be equal and can grow at most n_{axes} long.

`setPayloadStack`(*vb_idx*, *manip*)
Sets the stack of payload objects (solid volume body objects) that will be assigned to specific manipulators (hands) of the robot for successive processing.

Parameters

- **vb_idx** (*int, vector*) – $(n \times 1)$ vector of *reference indices* of the solid volume body objects that are defined as payloads.
- **manip** (*cellstr, vector*) – String array to assign each payload object with a specific *manipulator* (hand) of the robot, specified by one of these values:
  - `'lh'`: Use the *left hand* as manipulator.
  - `'rh'`: Use the *right hand* as manipulator.
  - `'bh'`: Use *both hands* to manipulate the object.

setPayloadUtilTime(*obj_idx*, *start_idx*, *end_idx*)

Sets the utilization time of each payload object in the stack, to determine the time indices for the simulation when an object is grabbed and released by the specified manipulator (hand).

Parameters

- **obj_idx** (*int, scalar*) – Index position of the payload object in the stack.
- **start_idx** (*int, scalar*) – Time index position of the integration output matrix $\mathcal{X}$ when the robot has grabbed the object.
- **end_idx** (*int, scalar*) – Time index position of the integration output matrix $\mathcal{X}$ when the robot has released the object.

See also:

`genericSimConfig`, `wbmSimBody`, `wbmSimEnvironment`, `wbmLinkTrajectory` and `wbmTargetPoint`.

class +WBM.**wbmSimBody**(*jnt_lnk_names*, *jnt_pair_idx*, *draw_prop*)

Bases: `handle`

`wbmSimBody` is a *data type* (class) that defines the *body model*, i.e. the geometric shape, of the simulated robot.

shape_geom

struct – Data structure to define the shape and the size of all *body parts* (patches) for the links and the feet of the robot's skeleton.

Note: The structure has two data fields, `size_sf` and `faces`. The field `size_sf` represents a $(n_{lnks} \times 2)$ data matrix with *constant scale factors* of the form $[sf_{width}, sf_{height}]$, to define the shape sizes for the body parts of all links of the robot. The row index of the matrix is equal to the joint index number of the robot. The second field `faces` denotes a (6×4) *vertex connection matrix* to define which vertices are to connect for the polygons of the patches. The patches are defining the body parts of the robot.

shape_size_sf

double, matrix – $(n_{lnks} \times 2)$ matrix to set only the *scale factors* $[sf_{width}, sf_{height}]$ for the shape sizes of the body parts of the robot (see *shape_geom*).

shape_faces

uint8, matrix – (6×4) vertex connection matrix to set only the *vertex connections* for the polygons of the patches that are forming the body parts for the robot (see *shape_geom*).

foot_geom

struct – Data structure to define the *geometric form* of each foot of the robot.

Note: The data structure is a nested structure and contains following fields:

- `joints` (*uint8, vector*): (2×1) vector to specify the joint indices where the left and right foot is connected.
- `base_sz` (*struct*): Structure with the fields `width` and `height`, to specify the *base size values* (double) for the feet of the robot. Optional, instead of the structure, a row-vector of the form $[width, height]$ can also be used to set the size values.

- `shape_ds` (*double, matrix*): $(n \times 3)$ matrix with *size values* of the form $[length, width, height]$ to define the *foot dimensions* in the x, y and z directions.

`foot_joints`
uint8, vector – (2×1) vector to set the joint indices for the left and the right foot (see *`foot_geom`*).

`foot_base_sz`
struct/vector – Data structure, or optional a vector, to set the *base size values*, i.e. the `width` and the `height`, for the feet of the robot (see *`foot_geom`*).

`foot_shape_ds`
double, matrix – $(n \times 3)$ matrix with size values of the form $[length, width, height]$ to set the *dimensions* for the feet (see *`foot_geom`*).

`draw_prop`
`wbmRobotDrawProp` – Data object to control the draw properties for the body of the simulated robot.

`jnt_lnk_names`
cell, vector – Column-array of *link* and *frame names* that are related to specific parent joints or parent links of the robot model[1].

`jnt_pair_idx`
uint8, matrix – $(n \times 6)$ matrix of *joint pair indices* for the x, y and z-positions to describe the configuration, i.e. the *connectivity graph*, of the robot's skeleton in 3D-space. In general $n = n_{jnts}$, but if the given robot also has a *root link* and/or a *com link* that have no parent joint, then either $n = n_{jnts} - 1$ or $n = n_{jnts} - 2$.

Note: Each joint pair is connected with a rigid link to form a kinematic chain of the robot. The joint index pairs in the matrix denoting the index positions of the given joint names in *`jnt_lnk_names`*. The index pairs are specified in the matrix as row-vectors of the form $[x_1, x_2, y_1, y_2, z_1, z_2]$. The matrix will be used to create a set of position parameters that describes the skeleton configuration of the robot.

`nJnts`
uint8, scalar – Number of joints of the robot model[2].

`nLnks`
uint8, scalar – Number of links of the robot model[2].

`nFeet`
uint8, scalar – Number of feet of the robot model[2].

`wbmSimBody` (*jnt_lnk_names*, *jnt_pair_idx*, *draw_prop*)
Constructor.

Parameters

- **`jnt_lnk_names`** (*`cellstr, vector`*) – Column-array of *link* and *frame names* that are deduced from their parent joints or parent links of the robot model[1] (see *`jnt_lnk_names`*).
- **`jnt_pair_idx`** (*`uint8, matrix`*) – $(n \times 6)$ matrix of *joint pair indices* of the form $[x_1, x_2, y_1, y_2, z_1, z_2]$, to describe the configuration, i.e. the *connectivity graph*, of the robot's skeleton in 3D-space (see *`jnt_pair_idx`*).
- **`draw_prop`** (`wbmRobotDrawProp`) – The draw properties for the body of the simulated robot.

Returns *obj* – An instance of the `wbmSimBody` data type.

class `+WBM.`**`wbmRobotParams`**
Bases: `handle`

[1] Details about the joint, link and frame names in particular for the *iCub robot model* are available at the *iCub Model Naming Conventions*: http://wiki.icub.org/wiki/ICub_Model_naming_conventions.
[2] If the value is *zero*, then the robot model is undefined.

`wbmRobotParams` is a *data type* (class) to export the model and configuration parameters of a given floating-base robot between interfaces.

`model`
`wbmRobotModel` – Model object with the model parameters of the given floating-base robot.

`config`
`wbmRobotConfig` – Configuration object with the configuration settings of the given floating-base robot.

`wf2fixlnk`
logical, scalar – Boolean flag to indicate if the world frame (wf) is set to a fixed reference link frame.

`copy``()`
Copy function to create deep object copies.

Returns *newObj* – A full copy of the robot parameters.

See also:

`WBM`, `Interfaces.IWBM` and `Interfaces.IMultChainTree`.

class `+WBM.`**`wbmRobotModel`**
Bases: `handle`

`wbmRobotModel` is a *data type* (class) that specifies the model parameters of a given floating-base robot.

`yarp_robot_type`
char, vector – String to specify the type of a YARP-based robot (default type: `'iCub'`).

`urdf_robot_name`
char, vector – String matching *name* of an existing robot model[3], or a string with the full path to a specific URDF-file[4] to be read.

`ndof`
uint16, scalar – Number of degrees of freedom of the given robot.

`urdf_fixed_link`
char, vector – String matching URDF name of the *fixed reference link frame* w.r.t. a world frame (wf). The given reference link of the robot will be used as the *floating-base link*.

Note: The default fixed link (floating-base link) of each YARP-based robot can be different and the selection of the floating-base link depends also from the situation. For example the default fixed link of the iCub humanoid robot is `l_sole`.

`wf_R_b_init`
double, matrix – (3×3) rotation matrix as *initial orientation* from the base frame *b* to the world frame *wf*. The default orientation is the *identity matrix*.

`wf_p_b_init`
double, vector – (3×1) vector to specify the *initial position* from the base frame *b* to the world frame *wf*. The default position is $[0, 0, 0]^T$.

`g_wf`
double, vector – (3×1) Cartesian gravity vector in the world frame *wf*. If the gravity is not defined, then the gravity vector is by default a 0-vector.

`frict_coeff`
struct – Data structure for the friction coefficients of of the joints of the given robot model, specified by the field variables:

- `v`: $(n_{dof} \times 1)$ coefficient vector for the *viscous friction*.
- `c`: $(n_{dof} \times 1)$ coefficient vector for the *Coulomb friction*.

[3] The URDF-file of the given robot model must exist in the directory of the *yarpWholeBodyInterface* or *CoDyCo-Superbuild*.
[4] The Unified Robot Description Format (URDF) is an XML specification to describe a robot model.

jlmts

struct – Data structure for the *joint positions limits* of the given robot model, specified by the field variables:

- `lwr`: $(n_{dof} \times 1)$ vector for the *lower* joint limits.
- `upr`: $(n_{dof} \times 1)$ vector for the *upper* joint limits.

See also:

`WBMBase` and `WBM`.

class `+WBM.`**wbmRobotDrawProp**

Bases: `handle`

`wbmRobotDrawProp` is a *data type* (class) to control the draw properties for the body of the simulated robot.

joints

struct – Structure to control the appearance of the joint nodes, specified by following fields:

- `line_width` (*double, scalar*): Line width of the marker, specified as a positive value in points.
- `marker` (*char, vector*): Marker symbol for the joint nodes. The symbols for the marker are the same as specified in Matlab.
- `marker_sz` (*double, scalar*): Marker size, specified as a positive value in points.
- `color` (*double/char, vector*): Color of the joint nodes, specified by a RGB-triplet or a color name.

links

struct – Structure to control the appearance of the links (edges) of the robot's skeleton:

- `line_width` (*double, scalar*): Line width of the links, specified as a positive value in points.
- `color` (*double/char, vector*): Link color, specified by a RGB-triplet or a color name.

com

struct – Structure to control the appearance of the center of mass node:

- `marker` (*char, vector*): Marker symbol for the node of the center of mass. The symbols for the marker are the same as specified in Matlab.
- `marker_sz` (*double, scalar*): Marker size, specified as a positive value in points.
- `color` (*double/char, vector*): Node color of the center of mass, specified by a RGB-triplet or a color name.

shape

struct – Structure to control the appearance of the body shape of the robot:

- `line_width` (*double, scalar*): Line width of the polygons (edges) for the body parts of the robot, specified as a positive value in points.
- `edge_color` (*double/char, vector*): Edge color for the body parts of the robot, specified by a RGB-triplet or a color name.
- `face_color` (*double/char, vector*): Face color for the body parts of the robot, specified by a RGB-triplet or a color name.
- `face_alpha` (*double, scalar*): Face transparency of the body parts, specified by a scalar in range $[0, 1]$.

class `+WBM.`**wbmRobotConfig**

Bases: `handle`

`wbmRobotConfig` is a *data type* (class) that specifies the configuration parameters of a given floating-base robot.

nCstrs
uint8, scalar – Number of contact constraints of those links, where the robot model is in contact with the environment (ground or object).

ccstr_link_names
cellstr, vector – $(1 \times n_{cstrs})$ string array of *contact constraint link names* to specify which link of the robot is in contact with the ground or an object.

nPlds
uint8, scalar – Number of payloads that are assigned to specific links of the robot (default: 0).

payload_links
`wbmPayloadLink`, *vector* – $(1 \times n_{plds})$ array of *payload link objects*, where each payload is assigned to a specific link of the robot (default: *empty*).

Note: The indices 1 and 2 of the array are always reserved for the *left* and the *right hand* of a given humanoid robot. The default index that will be used by the `WBM` class is always 1. All further indices of the array can be used for other links, on which additionally special payloads are mounted (e.g. a battery pack or a knapsack at the torso, special tools, etc.).

nTools
uint8, scalar – Number of tools that are assigned to specific links of the robot. The maximum number of tools that a humanoid robot can use simultaneously, is by default 2 (see `WBM.MAX_NUM_TOOLS`). The default number is 0.

tool_links
`wbmToolLink`, *vector* – $(n \times 1)$ array of *tool link objects*, where each tool is assigned to a specific link of the robot. The length n of the array can not exceed the limit `WBM.MAX_NUM_TOOLS`. By default, the initial array is *empty*.

Note: The *default tool link* for the robot is always the first element of the object list.

init_state_params
`wbmStateParams` – Data object to define the initial state parameters of a given floating-base robot.

stvLen
uint16, scalar – Length of the state parameter vector, i.e. the length of all state parameters concatenated in one vector. The length can be either $2n_{dof} + 13$ (default), $2n_{dof} + 6$ (without `x_b` and `qt_b`) or 0 if *`init_state_params`* is empty[5].

body
`wbmBody` – Data object to define and configure the body components of the given robot.

See also:

`wbmHumanoidConfig`.

class `+WBM.`**wbmHumanoidConfig**
Bases: `WBM.wbmRobotConfig`

`wbmHumanoidConfig` is an extended *data type* (class) of the class `wbmRobotConfig` and represents the general configuration structure of a *humanoid robot*.

jpos_head
double, vector – Initial joint positions for the head.

jpos_torso
double, vector – Initial joint positions for the torso.

jpos_left_arm
double, vector – Initial joint positions for the left arm.

jpos_left_hand
double, vector – Initial joint positions for the left hand.

[5] The state parameter object is defined as *empty*, if all variables of the given data type are empty.

jpos_left_leg
double, vector – Initial joint positions for the left leg.

jpos_left_foot
double, vector – Initial joint positions for the left foot.

jpos_right_arm
double, vector – Initial joint positions for the right arm.

jpos_right_hand
double, vector – Initial joint positions for the right hand.

jpos_right_leg
double, vector – Initial joint positions for the right leg.

jpos_right_foot
double, vector – Initial joint positions for the right foot.

Note: The given *joint positions vectors* for the initial body pose of the robot must be specified in the data type as *column-vectors* and the corresponding *angle positions* of the joints are defined in degrees ([deg]).

See also:

`wbmRobotConfig`.

class `+WBM.`**`wbmFltgBaseState`**
Bases: `handle`

`wbmFltgBaseState` is a *data type* (class) to represent the actual state parameters of the robot's floating base *b* related to the world frame *wf*.

wf_R_b
double, matrix – (3×3) orientation matrix of the base (in axis-angle representation).

wf_p_b
double, vector – (3×1) Cartesian position vector of the base.

v_b
double, vector – (6×1) generalized base velocity vector (Cartesian and rotational velocity of the base).

Note: The data type is also useful for interfaces to get quickly the current state of the floating base.

See also:

`Interfaces.iCubWBM` and `WBMBase.getFloatingBaseState()`.

See also:

`Interfaces.iCubWBM` and `WBMBase.getFloatingBaseState()`.

class `+WBM.`**`wbmBody`**(*chn_names*, *chn_idx*, *jnt_names*, *jnt_idx*)
Bases: `handle`

`wbmBody` is a *data type* (class) to define and configure the body components of a given floating-base robot.

chains
cell, matrix – $(n_{chns} \times 3)$ list of *chain names* (effectors) with their corresponding *joint index pair numbers*, specified in the form $\{chain\_name, idx_{start}, idx_{end}\}$. The joint index pairs are defining the start and end positions of each chain in the joint name array.

Note: A "chain" represents a subset of joints of a robot that are linked together. The joint names of the robot must be named as defined in the iCub Model Naming Conventions and must be set in the same order as specified in the configuration file yarpWholeBodyInterface.ini.

nChns
uint8, scalar – Number of chains that are defining the full body of the robot (default 0, if the robot is undefined).

joints
cell, matrix – $(n_{jnts} \times 2)$ list of *joint names* with the corresponding *joint index number*, specified in the form $\{jnt\_name, jnt_idx\}$.

Note: The order of the joint names must be set as defined in the list `ROBOT_MEX_WBI_TOOLBOX` of the configuration file yarpWholeBodyInterface.ini.

nJnts
uint8, scalar – Number of joints of the given robot (default 0, if the robot is undefined).

wbmBody (*chn_names*, *chn_idx*, *jnt_names*, *jnt_idx*)
Constructor.

Parameters

- **chn_names** (*cellstr, vector*) – Column-array with the *chain names* of each chain element of the robot model.
- **chn_idx** (*int, matrix*) – $(n_{chns} \times 2)$ matrix of the form $[idx_{start}, idx_{end}]$, with start and end positions of each chain of the robot.
- **jnt_names** (*cellstr, vector*) – Column-array with the *joint names* of the robot model as specified in the iCub Model Naming Conventions.
- **jnt_idx** (*int, vector*) – $(nJnts \times 1)$ vector with ascending index numbers for the joints of the robot.

Returns *obj* – An instance of the `wbmBody` data type of the given robot.

getChainIndices (*chn_name*)
Returns the index numbers of all joint members of a given chain.

Parameters chn_name (*char, vector*) – String matching name of the chain element of the robot.

Returns *jnt_idx (int, vector)* – Row vector with the joint index numbers.

getJointIndex (*jnt_name*)
Returns the index number of a specific joint.

Parameters jnt_name (*char, vector*) – String matching name of a specific joint of the robot.

Returns *jnt_idx (int, scalar)* – Index number of the given joint.

getJointNames (*jnt_idx*)
Returns the joint names of a given list of joint index numbers.

Parameters chn_idx (*int, vector*) – Row-vector with the index numbers of specific joints in ascending or descending order.

Returns *jnt_names (cellstr, vector)* – Column-vector with the joint names that are corresponding to the index list.

getChainTable ()
Returns the table of all chain elements of the robot body.

Returns *chn_tbl (table)* – Table object with the variables `chain_name`, `start_idx` and `end_idx` as column names.

getJointTable ()
Returns the table of all joints of the robot body.

Returns *jnt_tbl (table)* – Table object with the variables `joint_name` and `idx` as column names.

class +WBM.**genericSimConfig**(*wnd_title*, *rob_sim_body*, *varargin*)

Bases: `WBM.wbmSimConfig`

`genericSimConfig` is an *data type* (class) and represents the *generic configuration* for the visualizer of the robot simulation.

custom_view
double, vector – Custom viewpoint specification in terms of *azimuth* (az) and *elevation* (el) as a row-vector $[az, el]$.

axes_vwpts
cell, matrix – List of predefined viewpoint positions for the orientation of the axes (*readonly*). The list is a 2-dimensional array where in the first column are the names of the axes orientations and in the second column the corresponding viewpoint vectors.

wnd_visible
logical, scalar – Figure window visibility, specified as *true* or *false*. If the property is set to *false*, then the entire figure is disabled and invisible on the screen. This is important for cases where the visibility of the figure is temporarily undesired (e.g. for performance reasons).

DF_WND_POS
double, vector – Default window position vector, specified in *pixels* with the values $[50\text{ (left)}, 400\text{ (bottom)}]$, (*constant*).

DF_WND_SIZE
double, vector – Default window size of the figure, specified as a vector with the values $[600\text{ (width)}, 650\text{ (height)}]$, in *pixels* (*constant*).

DF_AXES_NBR
uint8, scalar – Default number of 4 axes graphics objects in the figure window, to divide the figure into a (2×2) grid (*constant*).

DF_AXES_POS
double, matrix – Default positions of each axes graphics object in the figure window, specified as a (4×4) matrix of the form $[x\text{ (left)}, y\text{ (bottom)}, wid\text{ (width)}, hgt\text{ (height)}]$, where each row represents an axes position vector (*constant*).

DF_AXES_COLORS
double, matrix – Default (4×3) color matrix for the axes back planes and for the axis lines, tick values and labels (*constant*).

Note: The default color for all axes components is defined as `gray80` (hex: #CCCCCC) of the color class `wbmColor`.

DF_AXES_VWPTS
cell, matrix – List of default viewpoint positions to determine the orientation of each axes (*constant*).

Following default axes viewpoints are defined:

Viewpoint Name:	View Angles $[az, el]$:
`'front'`	$[-90°, 1°]$
`'side_l'`	$[0°, 1°]$
`'side_r'`	$[180°, 1°]$
`'top'`	$[-90°, 90°]$
`'custom'`	$[-37.5°, 30°]$

DF_VWPTS_ANNOT
cellstr, vector – Default list with following annotation strings (titles) for the default axes viewpoints (*constant*):

```
{'front', 'l. side', 'r. side', 'top', 'perspective'}.
```

DF_AXES_VIEWS
cell, vector – Default viewpoint name list that specifies in the visualizer the view of each axes (*constant*):

{'custom', 'top', 'side_l', 'front'}.

DF_AXIS_LIMITS
double, vector – Default (6×1) vector of the form $[x_{min}, x_{max}, y_{min}, y_{max}, z_{min}, z_{max}]$, to specify the axis limits of every Cartesian axes in the figure window (*constant*).

DF_GROUND_SHAPE
double, matrix – Default (3×4) polygon matrix, consisting of four rectangular vertex coordinates, which form the shape of the ground floor plane (*constant*).

DF_GROUND_COLOR
double, vector – Default color of the ground, specified by the RGB-triplet $[153, 153, 204]$ (hex: #9999CC) (*constant*).

DF_GROUND_COLOR2
double, vector – Optional default color for the ground, specified by the RGB-triplet $[201, 220, 222]$ (hex: #C9DCDE) (*constant*).

DF_LIGHT_POS
double, vector – Default location of the point light, specified by the Cartesian vector $[0.25, 0, 1.25]$ (*constant*).

DF_VID_FILENAME
char, vector – Default filename string for creating a video file of the current robot simulation: `'robot-simulation.avi'` (*constant*).

DF_VID_FPS
double, scalar – Default rate of video playback in *frames per second* [fps]: 30 (*constant*).

robot_body
`wbmSimBody` – Data object to define the geometric shape for the body of the simulated robot.

environment
`wbmSimEnvironment` – Data object to define the environment settings for the robot simulation.

trajectories
`wbmLinkTrajectory`, *vector* – Array of trajectory objects to show the trajectory curves of specific links of the robot.

target_pts
`wbmTargetPoint`, *vector* – Array of target points that should reached in the simulation by specific links of the robot.

hWndFigure
`matlab.ui.Figure` – Figure window for the visualization of the robot simulation.

wnd_title
char, vector – Title (name) of the figure window.

wnd_pos
double, vector – Location of the left inner corner of the figure window with the origin at the left bottom corner of the primary display, specified as a row-vector of the form $[pos_{left}, pos_{bottom}]$ (default location: `DF_WND_POS`).

wnd_size
double, vector – Size of the figure window, specified as a row-vector of the form $[width, height]$ (default size: `DF_WND_SIZE`).

show_wnd
logical, scalar – Boolean flag to determine if the figure should be displayed on the screen. If the value is set to *false*, then the figure is after the simulation setup disabled and invisible on the screen. This can be helpful for performance reasons, e.g., to increase the speed of creating a video of the current simulation (default: *true*).

nAxes
uint8, scalar – Number of axes graphics objects in the figure window (default number: `DF_AXES_NBR`).

hAxes
double, vector – $(1 \times n_{axes})$ array with handles of the created axes objects.

axes_pos
double, matrix – Matrix with size and position of each axes within the figure window, specified as row-vectors of the form $[pos_{left}, pos_{bottom}, width, height]$ (default positions: `DF_AXES_POS`).

axes_colors
double, matrix – Colors for the axes back planes and for the axis lines, tick values and labels, specified by RGB-triplets as row-vectors (default colors: `DF_AXES_COLORS`).

axes_views
cell, vector – Row-array of different string matching viewpoint names to define the robot simulation at each axes in a distinctive view (default views: `DF_AXES_VIEWS`).

axis_limits
double, vector – (6×1) vector of the form $[x_{min}, x_{max}, y_{min}, y_{max}, z_{min}, z_{max}]$, to specify the axis limits of every Cartesian axes in the figure window (default limits: `DF_AXIS_LIMITS`).

vwpts_annot
cell, vector – Row-list with short annotation strings (or titles) for the given viewpoints in the figure window.

gfx_objects
cell, vector – $(1 \times n_{axes})$ array of *graphics object handles* as data container for each axes in the figure window to store, manipulate and visualize all graphics objects (environment, robot body, trajectories and target points) of the simulation.

show_light
logical, scalar – Boolean flag to enable a point light in the simulation environment that radiates from a specified location in all directions (default: *false*).

light_pos
double, vector – Location of the point light, specified by a Cartesian vector $[x, y, z]$.

show_titles
logical, scalar – Boolean flag to show the viewpoint titles in the left bottom corner of each axes (default: *true*).

tit_font_sz
double, scalar – Font size of the viewpoint titles, specified as a scalar numeric value.

tit_font_color
double/char, vector – Font color of the viewpoint titles, specified by a RGB-triplet or a color name.

show_legend
logical, scalar – Boolean flag to show in the figure window the legend for the link trajectories of the robot (default: *true*).

lgd_font_sz
double, scalar – Font size of the legend, specified as a scalar numeric value.

lgd_font_color
double/char, vector – Font color of the legend, specified by a RGB-triplet or a color name.

lgd_bkgrd_color
double/char, vector – Background color of the legend, specified by a RGB-triplet or a color name.

lgd_edge_color
double or char, vector – Edge color of the box outline of the legend, specified by a RGB-triplet or a color name.

lgd_location
char, vector – Location of the legend with respect to the axes. The location values are the same as listed in the table of the *legend properties* of Matlab.

lgd_orient
char, vector – Orientation of the legend, specified by one of these values: `'vertical'`, `'horizontal'`.

mkvideo
logical, scalar – Boolean flag to create a video of the current robot simulation (default: *false*).

Note: If the flag is enabled by setting the value to *true*, then the figure window will not shown on the screen, to speed up the creation of the video.

vid_axes_idx
uint8, scalar – Index of an axes graphics object in the figure window to capture an image or video only from a particular subplot frame. If the value is set to 0, then the entire figure window (all axes) will be captured for the image or video (default value: 0).

vid_filename
char, vector – The filename of the video with the `.avi` extension and the full path where the file will be written.

Note: Currently only the *uncompressed AVI* file format can be used for creating video files, because only this file format will be supported by Matlab on all platforms.

vid_fps
double, scalar – Frame rate of the video playback in frames per second [fps], specified by a positive number.

pl_stack
cell, matrix – Index stack (list) for the volume body objects that are defined as payloads for particular manipulators (hands) of the robot. Each payload object of the stack will be processed successively and is assigned to a manipulator of the robot, specified by one of these values:

- `'lh'`: Use the *left hand* as manipulator.
- `'rh'`: Use the *right hand* as manipulator.
- `'bh'`: Use *both hands* to manipulate the object.

Note: The payload stack is a 2-dimensional array. In the first column are listed the reference indices to the volume body objects of the environment and in the second column is each object linked to a particular manipulator (hand) for processing.

pl_time_idx
uint32, matrix – Utilization time matrix of the given payload objects in the stack to determine the points in time for the simulation when each object is grabbed and released by the specified manipulator.

Note: The rows of the index matrix representing time index vectors of the form, $[idx_{start}, idx_{end}]$, where idx_{start} denotes the time index when the object is grabbed and idx_{end} is the time index when the object is released by the manipulator.

zoom_axes
double, matrix – Zoom index matrix to specify which axes of the figure window should be zoomed by a particular zoom factor.

Note: Each row of the index matrix is of the form $[idx_{ax}, zfac]$, where idx_{ax} denotes the reference index of an axes graphics object in the figure and $zfac$ is the corresponding zoom factor. If $zfac > 1$, the scene appears larger (*zoom in*); if $zfac \geq 0$ and $zfac < 1$, the scene appears smaller (*zoom out*). If $zfac = 1$, then the scene keeps its original size. The row number of the matrix can grow at most n_{axes} long.

shift_axes
cell, matrix – Shift index array to specify which axes of the figure window should be shifted in a particular direction within the axis limits that are scaled to a given zoom factor. This is very useful if only a special area of the scene is of interest.

Note: The index array is a 2-element row-array of the form $\{idx_{ax}, v_{sh}\}$. The first element idx_{ax} of the array represents a column-vector with reference indices to the axes graphics objects in the figure window. The second element v_{sh} describes a $(n \times 3)$ matrix of Cartesian vectors of the form $[x, y, z]$.

Each row-vector of the matrix specifies the *shift direction* of the scene of the current axes. The row numbers of both array elements must be equal and can grow at most n_{axes} long.

genericSimConfig(*wnd_title*, *rob_sim_body*, *varargin*)

Constructor.

The constructor of the *generic simulation configuration* can be called in two different ways, where the *keyword arguments* in the square brackets are optional:

- **genericSimConfig**(*wnd_title*, *rob_sim_body*[, *nax*, *env_settings*])
- **genericSimConfig**(*wnd_title*, *rob_sim_body*[, *env_settings*])
- **genericSimConfig**(*wnd_title*, *rob_sim_body*[, *nax*])

Parameters

- **wnd_title** (*char, vector*) – String to specify the title (name) of the figure window.
- **rob_sim_body** (`wbmSimBody`) – Data object to define the geometric shape for the body of the simulated robot.
- **varargin** – Variable-length input argument list.

Keyword Arguments

- **nax** (*int, scalar*) – The number of axes graphics objects to be created in the figure window (*optional*).
- **env_settings** (`wbmSimEnvironment`) – Data object to define the environment settings for the robot simulation (*optional*).

Returns *obj* – An instance of the `genericSimConfig` data type.

zoomAxes(*axes_idx*, *zoom_fac*)

Specifies the axes in the figure window that should be zoomed in or out by a particular *zoom factor*.

Parameters

- **axes_idx** (*int, vector*) – Vector with the index numbers of the axes that should be zoomed in the figure window.
- **zoom_fac** (*double, vector*) – Vector with the corresponding *zoom factors* of the axes that should be zoomed in or out.

Note: Both vectors must have the same length and can not exceed the size of n_{axes}.

shiftAxes(*axes_idx*, *shift_pos*)

Specifies the axes in the figure window that should be shifted to a particular position of interest.

Parameters

- **axes_idx** (*int, vector*) – Vector with the index numbers of the axes that should be zoomed in the figure window.
- **shift_pos** (*double, matrix*) – $(n \times 3)$ matrix with the corresponding Cartesian *shift coordinates* of the axes that should be shifted.

Note: The index vector and the shift matrix must have the same length and can not exceed the size of n_{axes}.

addView(*vp_name*, *vp*)

Adds a new view to the viewpoint positions list that determines the axes orientations in the figure window.

Parameters

- **vp_name** (*char, vector*) – Variable name of the new viewpoint (axes orientation name).
- **vp** (*double, vector*) – (1×3) Cartesian viewpoint position vector.

Note: For flexibility reasons, the viewpoint list may contain more views than axes are indicated in the figure window.

---

createVideo(*varargin*)

Sets the settings for creating a video of the current robot simulation.

The method can be called in three different ways:

- **createVideo**(*filename*[, *rmode*, *fps*])
- **createVideo**(*filename*[, *rmode*])
- **createVideo**(*filename*[, *fps*])
- **createVideo**()

If the parameters are not completely given, then the method uses the default values for the video settings.

Parameters **varargin** – Variable-length input argument list.

Keyword Arguments

- **filename** (*char, vector*) – The filename of the video with the full path where the file will be written.

  Note: Currently only the *uncompressed AVI* file format can be used, since Matlab supports only this format on all platforms.

- **rmode** (*char, vector*) – Render mode to specify the rendering method for the graphics objects, specified by one of these values:
  - `'r_fast'`: Fast rendering mode (*childorder*), where the objects will be drawn in the order in which they are created by the graphics functions (*default*).
  - `'r_depth'`: Draw the graphics objects by a *depth-sort method* (back-to-front order) based on the current view.

    Note: Without further optimizations this method is mostly slow and can cause "animation flickering" during the simulation.

- **fps** (*int, scalar*) – Frame rate of the video playback in frames per second [fps] (default rate: `DF_VID_FPS`).

Note: The method enables in the configuration object the boolean property `mkvideo` with the effect, that the figure will not displayed on the screen during the creation of the video. This will increase the performance of creating the video.

---

renderMode(*rmode*, *vis*)

Specifies the *rendering method* to be used for the graphics objects during the animation of the robot.

Parameters

- **rmode** (*char, vector*) – Render mode, specified by one of these values:
  - `'r_fast'`: Fast rendering mode (*childorder*), where the objects will be drawn in the order in which they are created by the graphics functions (*default*).

  - `'r_depth'`: Draw the graphics objects by a *depth-sort method* (back-to-front order) based on the current view.

    Note: Without further optimizations this method is mostly slow and can cause "animation flickering" during the simulation.

- **vis** (*char, vector*) – Determines if the figure window will be *visible* on the screen, specified as `'on'` or `'off'` (default value: `'on'`).

Note: This method requires a Matlab version of at least R2014b (8.4.0) or higher!

See also:

`wbmSimConfig`, `wbmSimBody`, `wbmSimEnvironment`, `wbmLinkTrajectory` and `wbmTargetPoint`.

2.2 Default Error Messages

class `+WBM.`**`wbmErrorMsg`**

`wbmErrorMsg` is a *utility class* with a set of constant member variables of predefined error messages for the WBM-Library.

2.3 Tool and Payload Links

The tool and payload link classes are special classes to assign a tool or a payload object with a specific link of the robot.

class `+WBM.`**`wbmToolLink`**

`wbmToolLink` is a *data type* (class) to define a tool with its tool-tip frame (tt) at a specified link of an end-effector (ee).

`urdf_link_name`

char, vector – URDF-name of the end-effector link frame (default name: `'none'`).

`ee_vqT_tt`

double, vector – (7×1) VQ-transformation frame of the tool, relative from the tool-tip *tt* to the link frame of the end-effector *ee*. Default vector: $[0, 0, 0, 1, 0, 0, 0]^T$.

`pl_idx`

uint8, scalar – Index number of a specified *payload link object* for the case that a grabbed payload object is also a tool. If the index is set to 0, then the tool is not linked with a given payload object (default value: 0).

See also:

`wbmPayloadLink`.

class `+WBM.`**`wbmPayloadLink`**

`wbmPayloadLink` is a *data type* (class) to define a payload that is assigned to a specific link of a given floating-base robot.

`urdf_link_name`

char, vector – URDF-name of the specified *reference link frame* or *contact link frame*. The link frame can be an end-effector frame of a hand or a special frame on which a payload object is mounted (knapsack, battery pack, etc.). If the object is undefined, the default link name is `'none'`.

`lnk_p_cm`

double, vector – (3×1) Cartesian position vector relative from the center of mass *cm* of the payload object to the link frame *lnk*. Default position: $[0, 0, 0]^T$.

`t_idx`
: *uint8, scalar* – Index number of a specified *tool link object* to define that the grabbed payload object is also a tool. If the index value is 0, then the payload object is not linked with a tool (default value: 0).

`vb_idx`
: *uint8, scalar* – Index number of a *volume body object* that will be linked to the given link frame. If the index value is 0, then the payload is not assigned to any given geometric volume body of the simulation environment (default value: 0).

`vb_grabbed`
: *logical, scalar* – Boolean flag to indicate that the payload object is *grabbed* by the given reference link.

`vb_released`
: *logical, scalar* – Boolean flag to indicate that the payload object is *released* by the given reference link.

`m_rb`
: *double, scalar* – Mass of the rigid body (solid volume body) in $[\mathrm{kg}]$. The value is 0 by default, if the mass of the object is undefined.

`I_cm`
: *double, matrix* – (3×3) *inertia tensor* of a rigid body or geometric object with the origin of the coordinate system at the center of mass (CoM) of the object body. The matrix is *empty*, if the inertia is undefined.

See also:

`wbmToolLink`.

2.4 Graphics and Colors

class `+WBM.`**`wbmGObj`**
: `wbmGObj` is an *abstract data type* to define specific graphics objects for the robot simulation.

`description`
: *char, vector* – Short description string (annotation) about the graphics object.

`line_width`
: *double, scalar* – Line width of the graphics object, specified as a positive value in points.

`getGObj`(*obj*)
: *abstract* – Creates and draws the graphics object and returns a handle to it.

Note: The graphics object can consist of several sub-graphics objects.

`updGObj`(*obj, hgo*)
: *abstract* – Updates the data parameters of the graphics object.

See also:

`wbmTargetPoint` and `wbmLinkTrajectory`.

`getGObj`(*obj*)
: Creates a graphics object and returns a handle of it (*abstract*).

`updGObj`(*obj, hgo*)
: Updates the data parameters of the graphics object (*abstract*).

See also:

`wbmTargetPoint`, `wbmLinkTrajectory` and `vbObject`.

class `+WBM.`**`wbmColor`**
: `wbmColor` is a *utility class* with a list of constant member variables of predefined RGB-triplets to colorize the simulation of the robot with some nice colors.

The color list is based on the color map of *R for Statistical Computing* (www.r-project.org).

Further details about the R colors are available in the technical note of the *Stowers Institute for Medical Research*: http://research.stowers.org/mcm/efg/R/Color/Chart/index.htm.

2.5 Target Points and Link Trajectories

Special classes to create *trajectory curves* and *target points* to be reached in the simulation by specific links of the robot.

class +WBM.**wbmTargetPoint**(*p*)

Bases: `WBM.wbmGObj`

`wbmTargetPoint` is a *data type* (class) that defines a target point in the simulation environment which should be reached by a link of the simulated robot.

pos

double, vector – (3×1) Cartesian position vector of the target point in the world frame (wf). Default position: $[0, 0, 0]^T$.

description

char, vector – Short description string (annotation) about the target point (default: *empty*).

line_width

double, scalar – Line width of the marker, specified as a positive value in points (default width: 0.5).

marker

char, vector – Marker symbol for the data point. The symbols for the marker are the same as specified in Matlab (default symbol: `'o'`).

mkr_size

double, scalar – Marker size, specified as a positive value in points (default size: 8).

mkr_color

double/char, vector – Marker outline color, specified by a RGB-triplet or a color name (default color: `'yellow'`).

Note: The specified target points will be shown in the robot simulation, if in the given *simulation configuration structure*, an array of initialized *target point objects* will be set with a size of at least 1.

wbmTargetPoint(*p*)

Constructor.

Parameters **p** (*double, vector*) – (3×1) Cartesian position for the target point in world frame (wf).

Returns *obj* – An instance of the target point data type.

getGObj()

Plots a 3D data point at the specified target position and returns the graphics object handle of the created data point.

Returns *hgo* – Handle of the created graphics object (chart line object).

updGObj(*hgo*)

Updates the position of the 3D data point.

Parameters **hgo** – Graphics object handle of the 3D data point.

Returns *hgo* – Graphics object handle (chart line object) with the new position.

Example:

```
% define the target points:
trg_pts = repmat(WBM.wbmTargetPoint, nTrg, 1);
for i = 1:nTrg
   trg_pts(i,1).pos       = trg_pos(1:3,i);
   trg_pts(i,1).marker    = '+';
   trg_pts(i,1).mkr_color = 'red';
end

% setup the window and draw parameters for the WBM-simulator:
sim_config = initSimConfigICub(robot_model.urdf_robot_name);
sim_config = wbm_icub.setupSimulation(sim_config);
sim_config.target_pts = trg_pts;
```

See also:

`wbmSimConfig` and `genericSimConfig`.

class `+WBM.`**`wbmLinkTrajectory`**(*lnk_name*, *lnk_pos*)

Bases: `WBM.wbmGObj`

`wbmLinkTrajectory` is a *data type* (class) that specifies the trajectory curve of a specific link of the simulated robot.

`urdf_link_name`

char, vector – String matching URDF-name of the link frame to be tracked of the robot model (default name: `'none'`).

`jnt_annot_pos`

cell, vector – Cell array of the form $\{\text{lnk\_group\_name}, idx\}$, to specify the *index position* of the annotation string for the parent joint of the given link (default: *empty*).

Note: In `utilities`, the joint annotation function `getJointAnnotationICub()`, is only defined for the iCub humanoid robot. For other YARP-based robots, a custom annotation function must be used.

`description`

char, vector – Short description about the trajectory curve of the link (default: *empty*).

`line_style`

char, vector – Line style of the link trajectory curve. The options for the line style are the same as specified in Matlab (default line style: `':'`)

`line_width`

double, scalar – Line width of the trajectory curve, specified as a positive value in points (default width: 0.5).

`line_color`

double/char, vector – Color of the trajectory curve, specified by a RGB-triplet or a color name (default color: `'green'`).

`ept_marker`

char, vector – Marker symbol for the end point of the trajectory curve. The symbols for the marker are the same as specified in Matlab (default symbol: `'o'`).

`ept_marker_sz`

double, scalar – Marker size of the trajectory end point, specified as a positive value in points (default size: 8).

`ept_line_wid`

double, scalar – Line width of the trajectory end point, specified as a positive value in points (default width: 0.5).

`ept_color`

double/char, vector – Color of the trajectory end point, specified by a RGB-triplet or a color name (default color: `'green'`).

lnk_pos

double, matrix – $(n \times 3)$ data matrix with Cartesian positions to specify in the simulation the trajectory vertices of the given link frame of the robot. The size n of the matrix represents the *number of time steps* of the simulation.

wbmLinkTrajectory(*lnk_name*, *lnk_pos*)

Constructor.

Parameters

- **lnk_name** (*char, vector*) – String matching URDF-name of the link frame to be tracked of the robot model.
- **lnk_pos** (*double, matrix*) – $(n \times 3)$ data matrix with Cartesian positions in each row, to specify the coordinates of the trajectory vertices of the given link frame of the robot. The size n of the matrix represents the *number of time steps* of the simulation.

Returns *obj* – An instance of the `wbmLinkTrajectory` data type.

getGObj()

Creates and draws the trajectory curve of the specified link of the simulated robot.

Returns *hgo (gobjects, vector)* – Column-array of graphics object handles of the created trajectory curve.

updGObj(*hgo*)

Updates the vertex coordinates of the link trajectory curve.

Parameters **hgo** (*gobjects, vector*) – Column-array of graphics object handles of the link trajectory curve.

Returns *hgo (gobjects, vector)* – Column-array with the updated vertex coordinates of the graphics object handles.

See also:

`wbmGObj`, `wbmTargetPoint`, `wbmSimConfig` and `genericSimConfig`.

2.6 Volume Bodies

The volume body classes enables to create and specify simple *geometric volume body objects* for the environment scenario of the robot simulation, such as a *box*, a *ball*, or a *cylinder*. These basic geometric forms are mostly sufficient in a research project to create a simple scenario for a given robot model. Additionally the defined volume bodies can be linked with certain links of the robot, such as the hands. This enables in Matlab to simulate the manipulation of objects and their effects in a given environment.

class +WBM.**vbObject**

Bases: `matlab.mixin.Heterogeneous`, `handle`

`vbObject` is an *abstract class* to specify *geometric volume body objects* for the environment scenario of the robot simulation.

origin

double, vector – (3×1) Cartesian position of the origin of the object. Default origin: $[0, 0, 0]^T$.

rotm

double, matrix – (3×3) rotation matrix of the object. If undefined, the default orientation is the *identity matrix*.

tform

double, matrix – (4×4) transformation matrix of the object. If `origin` and `rotm` are undefined, then by default, the transformation matrix is an *identity matrix*.

`frame`
: *double, vector* – (7×1) VQ-transformation vector (position and orientation) of the object. If *`origin`* and *`rotm`* are undefined, then the frame vector uses the default transformation vector *`DF_FRAME`*.

`DF_FRAME`
: *double, vector* – Default frame vector (VQ-transformation) of the form $[0, 0, 0, 1, 0, 0, 0]^T$ (*constant*).

`description`
: *char, vector* – Short description string about the volume body object (default: *empty*).

`ismovable`
: *logical, scalar* – Boolean flag to indicate if the volume body object is movable (default: *false*).

`line_width`
: *double, scalar* – Line width of the edges of the (geometric) volume body, specified as a positive value in points.

`edge_color`
: *double/char, vector* – Edge color of the (geometric) volume body, specified by a RGB-triplet or a color name.

`face_color`
: *double/char, vector* – Face color of the volume body, specified by a RGB-triplet or a color name.

`face_alpha`
: *double, scalar* – Face transparency of the volume body, specified by a scalar in range $[0, 1]$.

`obj_type`
: *char, vector* – Type of the object (*read only*), specified by one of these values:

- `'obs'`: The object defines an *obstacle* (for the robot).
- `'bdy'`: The object is only an arbitrary *volume body* in the environment and not a specific obstacle.

The default object type is `'bdy'`.

`isobstacle`
: *logical, scalar* – Boolean flag to indicate if the volume body object is also defined as an obstacle (*read only*).

Note: The value will be set to *true*, if and only if *`obj_type`* is defined as an obstacle (*'obs'*), else *false* (default).

`issolid`
: *logical, scalar* – Boolean flag to indicate if the volume body is a *solid* object (*read only*). Default: *false*.

Note: If a volumetric mass density is given, then the object is automatically defined as a *solid volume body*, i.e. the variable `issolid` will be set to *true*.

`istool`
: *logical, scalar* – Boolean flag to indicate if the volume body defines also a tool to be used by the robot (*read only*). Default: *false*.

`init_frame`
: *double, vector* – (7×1) initial frame vector (VQ-transformation) of the object, in dependency of the given origin position and orientation (*read only*).

Note: If the attributes *`origin`* and *`rotm`* are not defined, then the initial frame uses the default transformation vector *`DF_FRAME`*.

`dimension`
: *double, vector* – Row-vector to define the dimensions of the geometric volume body (*read only*).

Note: The form of the dimension vector depends strongly on the given shape of the volume body and can be different to each object.

com
: *double, vector* – (3×1) Cartesian position of the center of mass (CoM) of the volume body object relative to the given origin (*read only*).

  Note: By default, the CoM of the object is placed at the origin.

rho
: *double, scalar* – Volumetric mass density of the object, specified as a positive value in $[\mathrm{kg}/\mathrm{m}^3]$ (*read only*).

  Note: The object is defined as a *solid volume body* if and only if the density value $\rho > 0$. If the density of the object is undefined, then the default value is 0.

m_rb
: *double, scalar* – Mass of the solid volume body object (rigid body *rb*) in $[\mathrm{kg}]$ (*read only*).

  Note: If *rho* is not defined, the by default the mass value is 0.

I_cm
: *double, matrix* – (3×3) inertia matrix of the solid volume body object at the center of mass *cm* (*read only*).

  Note: If *rho* is not defined, then the inertia of the object is by default the *identity matrix*.

mgrid
: *struct* – Data structure for the *grid point coordinates* of the internal 3D-meshgrid of the object (*read only*).

  The 3D-meshgrid will be used in the robot simulation to simulate the volume body as a solid obstacle.

  Note: The data structure consists of the fields `X`, `Y` and `Z` that are representing the x, y and z-coordinates over a grid as 3D-arrays with three inputs. The meshgrid for the object can be created only if the object is defined as an *obstacle*.

setInitFrame(*obj*, *varargin*)
: *abstract* – Sets the object to the given initial position and orientation.

getGObj(*obj*)
: *abstract* – Creates and draws the volume body object and returns a handle to the created graphics objects.

updGObj(*obj*, *hgo*)
: *abstract* – Updates the data parameters, i.e. the vertex coordinates, of the volume body object.

drawMGrid(*obj*, *pt_color*)
: *abstract* – Draws the meshgrid of the object and returns a graphics object handle to it.

ptInObj(*obj*, *pt_pos*)
: *abstract* – Determines if some specified points are below the surface (i.e. inside) of the volume body object.

See also:

`vbCuboid`, `vbCylinder` and `vbSphere`.

static getDefaultScalarElement()
: Returns the default volume body object for heterogeneous array operations.

  Returns *default_object* – An instance of `vbCuboid` with default values to preallocate the memory.

class +WBM.**vbSphere**(*varargin*)
: Bases: `WBM.vbObject`

  `vbSphere` is a class to specify *sphere objects* for the environment scenario of the robot simulation.

  origin
  : *double, vector* – (3×1) Cartesian position of the origin of the sphere. Default origin: $[0, 0, 0]^T$.

`rotm`
double, matrix – (3×3) rotation matrix of the sphere. If undefined, the default orientation is the *identity matrix*.

`tform`
double, matrix – (4×4) transformation matrix of the sphere. If *`origin`* and *`rotm`* are undefined, then by default, the transformation matrix is an *identity matrix*.

`frame`
double, vector – (7×1) VQ-transformation vector (position and orientation) of the sphere. If *`origin`* and *`rotm`* are undefined, then the frame vector uses the default transformation vector `DF_FRAME`.

`description`
char, vector – Short description string about the sphere object (default: *empty*).

`ismovable`
logical, scalar – Boolean flag to indicate if the sphere object is movable (default: *false*).

`line_width`
double, scalar – Line width of the edges of the sphere, specified as a positive value in points (default width: 0.4).

`edge_color`
double/char, vector – Edge color of the sphere, specified by a RGB-triplet or a color name (default color: *'black'*).

`face_color`
double/char, vector – Face color of the sphere, specified by a RGB-triplet or a color name (default color: `wbmColor.lightsteelblue`).

`face_alpha`
double, scalar – Face transparency of the sphere, specified by a scalar in range $[0, 1]$ (default alpha: 0.2).

`obj_type`
char, vector – Type of the object (*read only*), specified by one of these values:

- `'obs'`: The sphere defines an *obstacle* (for the robot).
- `'bdy'`: The sphere is only an arbitrary *volume body* in the environment and not a specific obstacle.

The default object type is `'bdy'`.

`isobstacle`
logical, scalar – Boolean flag to indicate if the sphere object is also defined as an obstacle (*read only*).

Note: The value will be set to *true*, if and only if *`obj_type`* is defined as an obstacle (*'obs'*), *false* otherwise (default).

`issolid`
logical, scalar – Boolean flag to indicate if the sphere is a *solid* object (*read only*). Default: *false*.

Note: If a volumetric mass density is given, then the sphere object is automatically defined as a *solid volume body*, i.e. the variable `issolid` will be set to *true*.

`istool`
logical, scalar – Boolean flag to indicate if the sphere defines also a tool to be used by the robot (*read only*). Default: *false*.

`init_frame`
double, vector – (7×1) initial frame vector (VQ-transformation) of the sphere, in dependency of the given origin position and orientation (*read only*).

Note: If the attributes *`origin`* and *`rotm`* are not defined, then the initial frame uses the default transformation vector `DF_FRAME`.

dimension

double, vector – (1×2) vector of the form $[radius_{inner}, radius_{outer}]$, which defines the dimensions of the sphere (*read only*).

Note: If the dimensions of the sphere are undefined, then the vector is by default a *0-vector*.

vertices

struct – Data structure for the *vertex coordinates* of the sphere at the given origin frame (*read only*).

Note: Since the sphere object is *approximated* by a $(n \times n)$ *spherical polyhedron*, the data structure consists of the fields `X`, `Y` and `Z` which representing the x, y and z-coordinate vectors of each vertex of the given spherical polyhedron.

com

double, vector – (3×1) Cartesian position of the center of mass (CoM) of the sphere relative to the given origin (*read only*).

Note: By default, the CoM of the sphere is placed at the origin.

rho

double, scalar – Volumetric mass density of the sphere, specified as a positive value in $[\mathrm{kg/m^3}]$ (*read only*).

Note: The sphere is defined as a *solid volume body* if and only if the density value $\rho > 0$. If the density of the sphere is undefined, then the default value is 0.

m_rb

double, scalar – Mass of the sphere object (rigid body *rb*) in $[\mathrm{kg}]$ (*read only*).

Note: If *rho* is not defined, then by default the mass value is 0.

I_cm

double, matrix – (3×3) inertia matrix of the sphere at the center of mass *cm* (*read only*).

Note: If *rho* is not defined, then the inertia of the sphere is by default the *identity matrix*.

mgrid

struct – Data structure for the *grid point coordinates* of the internal 3D-meshgrid of the sphere (*read only*).

The 3D-meshgrid will be used in the robot simulation to simulate the sphere object as a solid obstacle.

Note: The data structure consists of the fields `X`, `Y` and `Z` that are representing the x, y and z-coordinates over a grid, specified as 3D-arrays (with three inputs each). The meshgrid for the sphere object can be created only if the object is defined as an *obstacle*.

vbSphere(*varargin*)

Constructor.

The constructor of the `vbSphere` class can be called in two different ways, where the *keyword arguments* in the square brackets are optional:

- **vbSphere**(*ri*, *ro*[, *orig*, *rotm*[, *obj_prop*]])
- **vbSphere**(*r*[, *orig*, *rotm*[, *obj_prop*]])

The first option specifies a *hollow sphere* (sphere shell) as volume body object and the second one a *solid sphere* (ball).

Parameters **varargin** – Variable-length input argument list.

Keyword Arguments

- **ri** (*double, scalar*) – Inner radius of the sphere shell.
- **ro** (*double, scalar*) – Outer radius of the sphere shell.
- **r** (*double, scalar*) – Radius of the solid sphere.
- **orig** (*double, vector*) – (3×1) origin position of the sphere.

- **rotm** (*double, matrix*) – (3×3) orientation matrix of the sphere.
- **obj_prop** (*struct*) – Data structure for the *object properties*, specified by following fields:
  - `line_width`: Line width of the edges of the sphere.
  - `edge_color`: Edge color of the sphere (RGB-triplet or color name).
  - `face_color`: Face color of the sphere (RGB-triplet or color name).
  - `face_alpha`: Face transparency of the sphere in range $[0, 1]$.

  optional fields:
  - `description`: Annotation string of the sphere object.
  - `ismovable`: Boolean flag to indicate if the sphere object is *movable* or *fixed*.
  - `istool`: Boolean flag to indicate if the sphere object is also a *tool* or not.
  - `obj_type`: Object type, specified by the value *'obs'* or *'bdy'* (see `obj_type`).
  - `rho`: Volumetric mass density of the sphere object in $[\mathrm{kg/m}^3]$.

Returns *obj* – An instance of the `vbSphere` class.

`setInitFrame`(*varargin*)

Sets the origin frame of the sphere to the given initial position and orientation.

The method can be called in three different ways:

- **`setInitFrame`**(*p*, *R*)
- **`setInitFrame`**(*vqT*)
- **`setInitFrame`**()

If no parameters are given, then `setInitFrame()` uses for the initial origin frame of the sphere the current frame vector of the property `init_frame`.

Parameters **`varargin`** – Variable-length input argument list.

Keyword Arguments

- **p** (*double, vector*) – (3×1) Cartesian position to specify the *origin* of the sphere.
- **R** (*double, matrix*) – (3×3) rotation matrix to specify the *orientation* of the sphere.
- **vqT** (*double, vector*) – (7×1) VQ-transformation (position and orientation in quaternions) to specify the *origin frame* of the sphere.

`getGObj`()

Creates and draws the sphere object and returns a handle to the created graphics object.

Returns *hgo* – Handle to the chart surface graphics object of the created sphere.

`updGObj`(*hgo*)

Updates the vertex coordinates of the sphere.

Parameters **`hgo`** – Handle to the chart surface graphics object with the x, y and z-coordinate data of the sphere.

Returns *hgo* – Handle to the chart surface graphics object with the changed vertex coordinates of the sphere.

`drawMGrid`(*pt_color*)

Draws the meshgrid of the sphere and returns a graphics object handle to it.

Parameters **`pt_color`** (`double/char, vector`) – Color of the grid points, specified by a RGB-triplet or a color name (default color: *'green'*).

Returns *hmg* – Handle to the scatter series object of the generated meshgrid.

ptInObj(*pt_pos*)

Determines if some specified points are below the surface, i.e. inside, of the sphere.

Parameters **pt_pos** (*double, vector/matrix*) – A single position vector or a $(n \times 3)$ matrix with positions of some specified points.

Returns *result* – $(n \times 1)$ vector with the logical results if the given points are below the surface of the sphere. Each value is *true* if the given point is below the surface, *false* otherwise.

See also:

`vbObject`, `vbCuboid` and `vbCylinder`.

class +WBM.**vbCylinder**(*varargin*)

Bases: `WBM.vbObject`

`vbCylinder` is a class to specify *circular cylinder objects* for the environment scenario of the robot simulation.

origin

double, vector – (3×1) Cartesian position of the origin of the cylinder. Default origin: $[0, 0, 0]^T$.

rotm

double, matrix – (3×3) rotation matrix of the cylinder. If undefined, the default orientation is the *identity matrix*.

tform

double, matrix – (4×4) transformation matrix of the cylinder. If *origin* and *rotm* are undefined, then by default, the transformation matrix is an *identity matrix*.

frame

double, vector – (7×1) VQ-transformation vector (position and orientation) of the cylinder. If *origin* and *rotm* are undefined, then the frame vector uses the default transformation vector `DF_FRAME`.

description

char, vector – Short description string about the cylinder object (default: *empty*).

ismovable

logical, scalar – Boolean flag to indicate if the cylinder object is movable (default: *false*).

line_width

double, scalar – Line width of the edges of the cylinder, specified as a positive value in points (default width: 0.4).

edge_color

double/char, vector – Edge color of the cylinder, specified by a RGB-triplet or a color name (default color: *'black'*).

face_color

double/char, vector – Face color of the cylinder, specified by a RGB-triplet or a color name (default color: `wbmColor.lightsteelblue`).

face_alpha

double, scalar – Face transparency of the cylinder, specified by a scalar in range $[0, 1]$ (default alpha: 0.2).

obj_type

char, vector – Type of the object (*read only*), specified by one of these values:

- `'obs'`: The cylinder defines an *obstacle* (for the robot).
- `'bdy'`: The cylinder is only an arbitrary *volume body* in the environment and not a specific obstacle.

The default object type is `'bdy'`.

isobstacle
logical, scalar – Boolean flag to indicate if the cylinder object is also defined as an obstacle (*read only*).

Note: The value will be set to *true*, if and only if `obj_type` is defined as an obstacle (*'obs'*), *false* otherwise (default).

issolid
logical, scalar – Boolean flag to indicate if the cylinder is a *solid* object (*read only*). Default: *false*.

Note: If a volumetric mass density is given, then the cylinder object is automatically defined as a *solid volume body*, i.e. the variable `issolid` will be set to *true*.

istool
logical, scalar – Boolean flag to indicate if the cylinder defines also a tool to be used by the robot (*read only*). Default: *false*.

init_frame
double, vector – (7×1) initial frame vector (VQ-transformation) of the cylinder, in dependency of the given origin position and orientation (*read only*).

Note: If the attributes `origin` and `rotm` are not defined, then the initial frame uses the default transformation vector `DF_FRAME`.

dimension
double, vector – (1×3) vector of the form $[radius_{inner}, radius_{outer}, height]$, which defines the dimensions of the cylinder (*read only*).

Note: If the dimensions of the cylinder are undefined, then the vector is by default a *0-vector*.

vertices
struct – Data structure for the *vertex coordinates* of the cylinder at the given origin frame (*read only*).

Note: Since the cylinder object is *linearized* by a *n*-gonal prism (polyhedron), the data structure consists of the fields `X`, `Y` and `Z` which representing the x, y and z-coordinate vectors of each vertex of the given polyhedron.

com
double, vector – (3×1) Cartesian position of the center of mass (CoM) of the cylinder relative to the given origin (*read only*).

Note: By default, the CoM of the cylinder is placed at the origin.

rho
double, scalar – Volumetric mass density of the cylinder, specified as a positive value in $[\mathrm{kg/m^3}]$ (*read only*).

Note: The cylinder is defined as a *solid volume body* if and only if the density value $\rho > 0$. If the density of the cylinder is undefined, then the default value is 0.

m_rb
double, scalar – Mass of the cylinder object (rigid body *rb*) in $[\mathrm{kg}]$ (*read only*).

Note: If `rho` is not defined, then by default the mass value is 0.

I_cm
double, matrix – (3×3) inertia matrix of the cylinder at the center of mass *cm* (*read only*).

Note: If `rho` is not defined, then the inertia of the cylinder is by default the *identity matrix*.

mgrid
struct – Data structure for the *grid point coordinates* of the internal 3D-meshgrid of the cylinder (*read only*).

The 3D-meshgrid will be used in the robot simulation to simulate the cylinder object as a solid obstacle.

Note: The data structure consists of the fields `X`, `Y` and `Z` that are representing the `x`, `y` and z-coordinates over a grid, specified as 3D-arrays (with three inputs each). The meshgrid for the cylinder object can be created only if the object is defined as an *obstacle*.

vbCylinder(*varargin*)
Constructor.

The constructor of the `vbCylinder` class can be called in two different ways, where the *keyword arguments* in the square brackets are optional:

- **vbCylinder**(*ri*, *ro*, *h*[, *orig*, *rotm*[, *obj_prop*]])
- **vbCylinder**(*r*, *h*[, *orig*, *rotm*[, *obj_prop*]])

The first option specifies a *circular hollow cylinder* (cylindrical shell) as volume body object and the second one a *right circular cylinder*.

Parameters **varargin** – Variable-length input argument list.

Keyword Arguments

- **ri** (*double, scalar*) – Inner radius of the cylindrical shell.
- **ro** (*double, scalar*) – Outer radius of the cylindrical shell.
- **r** (*double, scalar*) – Radius of the right circular cylinder.
- **h** (*double, scalar*) – Height of the cylinder along the z-direction.
- **orig** (*double, vector*) – (3×1) origin position of the cylinder.
- **rotm** (*double, matrix*) – (3×3) orientation matrix of the cylinder.
- **obj_prop** (*struct*) – Data structure for the *object properties*, specified by following fields:
  - `line_width`: Line width of the edges of the cylinder.
  - `edge_color`: Edge color of the cylinder (RGB-triplet or color name).
  - `face_color`: Face color of the cylinder (RGB-triplet or color name).
  - `face_alpha`: Face transparency of the cylinder in range $[0, 1]$.

  optional fields:
  - `description`: Annotation string of the cylinder object.
  - `ismovable`: Boolean flag to indicate if the cylinder object is *movable* or *fixed*.
  - `istool`: Boolean flag to indicate if the cylinder object is also a *tool* or not.
  - `obj_type`: Object type, specified by the value *'obs'* or *'bdy'* (see *`obj_type`*).
  - `rho`: Volumetric mass density of the cylinder object in $[\mathrm{kg/m^3}]$.

Returns *obj* – An instance of the `vbCylinder` class.

setInitFrame(*varargin*)
Sets the origin frame of the cylinder to the given initial position and orientation.

The method can be called in three different ways:

- **setInitFrame**(*p*, *R*)
- **setInitFrame**(*vqT*)
- **setInitFrame**()

If no parameters are given, then `setInitFrame()` uses for the initial origin frame of the cylinder the current frame vector of the property *`init_frame`*.

Parameters **varargin** – Variable-length input argument list.

Keyword Arguments

- **p** (*double, vector*) – (3×1) Cartesian position to specify the *origin* of the cylinder.
- **R** (*double, matrix*) – (3×3) rotation matrix to specify the *orientation* of the cylinder.

- **vqT** (*double, vector*) – (7×1) VQ-transformation (position and orientation in quaternions) to specify the *origin frame* of the cylinder.

getGObj()

Creates and draws the cylinder object and returns a handle to the created graphics object.

Returns *hgo* – Handle to the chart surface graphics object of the created cylinder.

updGObj(*hgo*)

Updates the vertex coordinates of the cylinder.

Parameters **hgo** – Handle to the chart surface graphics object with the x, y and z-coordinate data of the cylinder.

Returns *hgo* – Handle to the chart surface graphics object with the changed vertex coordinates of the cylinder.

drawMGrid(*pt_color*)

Draws the meshgrid of the cylinder and returns a graphics object handle to it.

Parameters **pt_color** (*double/char, vector*) – Color of the grid points, specified by a RGB-triplet or a color name (default color: *'green'*).

Returns *hmg* – Handle to the scatter series object of the generated meshgrid.

ptInObj(*pt_pos*)

Determines if some specified points are below the surface, i.e. inside, of the cylinder[6].

Parameters **pt_pos** (*double, vector/matrix*) – A single position vector or a $(n \times 3)$ matrix with positions of some specified points.

Returns *result* – $(n \times 1)$ vector with the logical results if the given points are below the surface of the cylinder. Each value is *true* if the given point is below the surface, *false* otherwise.

See also:

`vbObject`, `vbCuboid` and `vbSphere`.

class +WBM.**vbCuboid**(*varargin*)

Bases: `WBM.vbObject`

`vbCuboid` is a class to create *rectangular cuboid objects* for the environment scenario of the robot simulation.

origin

double, vector – (3×1) Cartesian position of the origin of the cuboid. Default origin: $[0, 0, 0]^T$.

rotm

double, matrix – (3×3) rotation matrix of the cuboid. If undefined, the default orientation is the *identity matrix*.

tform

double, matrix – (4×4) transformation matrix of the cuboid. If *`origin`* and *`rotm`* are undefined, then by default, the transformation matrix is an *identity matrix*.

frame

double, vector – (7×1) VQ-transformation vector (position and orientation) of the cuboid. If *`origin`* and *`rotm`* are undefined, then the frame vector uses the default transformation vector `DF_FRAME`.

description

char, vector – Short description string about the cuboid object (default: *empty*).

ismovable

logical, scalar – Boolean flag to indicate if the cuboid object is movable (default: *false*).

[6] The calculation approach to determine if a point is inside of a cylinder is taken and adapted from:
- https://stackoverflow.com/questions/19899612/cylinder-with-filled-top-and-bottom-in-matlab and
- https://de.mathworks.com/matlabcentral/answers/43534-how-to-speed-up-the-process-of-determining-if-a-point-is-inside-a-cylinder.

line_width
double, scalar – Line width of the edges of the cuboid, specified as a positive value in points (default width: 0.4).

edge_color
double/char, vector – Edge color of the cuboid, specified by a RGB-triplet or a color name (default color: `'black'`).

face_color
double/char, vector – Face color of the cuboid, specified by a RGB-triplet or a color name (default color: `wbmColor.lightsteelblue`).

face_alpha
double, scalar – Face transparency of the cuboid, specified by a scalar in range $[0, 1]$ (default alpha: 0.2).

obj_type
char, vector – Type of the object (*read only*), specified by one of these values:

- `'obs'`: The cuboid defines an *obstacle* (for the robot).
- `'bdy'`: The cuboid is only an arbitrary *volume body* in the environment and not a specific obstacle.

The default object type is `'bdy'`.

isobstacle
logical, scalar – Boolean flag to indicate if the cuboid object is also defined as an obstacle (*read only*).

Note: The value will be set to *true*, if and only if *`obj_type`* is defined as an obstacle (`'obs'`), *false* otherwise (default).

issolid
logical, scalar – Boolean flag to indicate if the cuboid is a *solid* object (*read only*). Default: *false*.

Note: If a volumetric mass density is given, then the cuboid object is automatically defined as a *solid volume body*, i.e. the variable `issolid` will be set to *true*.

istool
logical, scalar – Boolean flag to indicate if the cuboid defines also a tool to be used by the robot (*read only*). Default: *false*.

init_frame
double, vector – (7×1) initial frame vector (VQ-transformation) of the cuboid, in dependency of the given origin position and orientation (*read only*).

Note: If the attributes *`origin`* and *`rotm`* are not defined, then the initial frame uses the default transformation vector `DF_FRAME`.

dimension
double, vector – (1×3) vector of the form [width, length, height], which defines the dimensions of the cuboid (*read only*).

Note: If the dimensions of the cuboid are undefined, then the vector is by default a *0-vector*.

vertices
double, matrix – (8×3) *vertex positions matrix* of the cuboid at the given origin frame (*read only*).

Note: If the dimensions of cuboid are not defined, then the matrix is by default a *0-matrix*.

com
double, vector – (3×1) Cartesian position of the center of mass (CoM) of the cuboid relative to the given origin (*read only*).

Note: By default, the CoM of the cuboid is placed at the origin.

rho
double, scalar – Volumetric mass density of the cuboid, specified as a positive value in $[\mathrm{kg/m^3}]$ (*read only*).

Note: The cuboid is defined as a *solid volume body* if and only if the density value $\rho > 0$. If the density of the cuboid is undefined, then the default value is 0.

m_rb
double, scalar – Mass of the cuboid object (rigid body *rb*) in [kg] (*read only*).

Note: If `rho` is not defined, then by default the mass value is 0.

I_cm
double, matrix – (3×3) inertia matrix of the cuboid at the center of mass *cm* (*read only*).

Note: If `rho` is not defined, then the inertia of the cuboid is by default the *identity matrix*.

mgrid
struct – Data structure for the *grid point coordinates* of the internal 3D-meshgrid of the cuboid (*read only*).

The 3D-meshgrid will be used in the robot simulation to simulate the cuboid object as a solid obstacle.

Note: The data structure consists of the fields `X`, `Y` and `Z` that are representing the x, y and z-coordinates over a grid, specified as 3D-arrays (with three inputs each). The meshgrid for the cuboid object can be created only if the object is defined as an *obstacle*.

vbCuboid(*varargin*)
Constructor.

The constructor of the `vbCuboid` class can be called in three different ways, where the *keyword arguments* in the square brackets are optional:

- **vbCuboid**(*l_x*, *l_y*, *l_z*[, *orig*, *rotm*[, *obj_prop*]])
- **vbCuboid**(*l*[, *orig*, *rotm*[, *obj_prop*]])
- **vbCuboid**()

The first two options specifying as volume body object either a *rectangular cuboid* or a *cube*. The third option without parameters will be used for creating the *default object* of a heterogeneous hierarchy in `vbObject` (see `getDefaultScalarElement()`).

Parameters **varargin** – Variable-length input argument list.

Keyword Arguments

- **l_x** (*double, scalar*) – Length along the x-direction (width).
- **l_y** (*double, scalar*) – Length along the y-direction (length).
- **l_z** (*double, scalar*) – Length along the z-direction (height).
- **l** (*double, scalar*) – Length of all sides of the cube.
- **orig** (*double, vector*) – (3×1) origin position of the cuboid.
- **rotm** (*double, matrix*) – (3×3) orientation matrix of the cuboid.
- **obj_prop** (*struct*) – Data structure for the *object properties*, specified by following fields:
  - `line_width`: Line width of the edges of the cuboid.
  - `edge_color`: Edge color of the cuboid (RGB-triplet or color name).
  - `face_color`: Face color of the cuboid (RGB-triplet or color name).
  - `face_alpha`: Face transparency of the cuboid in range $[0, 1]$.

  optional fields:

  - `description`: Annotation string of the cuboid object.
  - `ismovable`: Boolean flag to indicate if the cuboid object is *movable* or *fixed*.
  - `istool`: Boolean flag to indicate if the cuboid object is also a *tool* or not.

- `obj_type`: Object type, specified by the value `'obs'` or `'bdy'` (see *obj_type*).
- `rho`: Volumetric mass density of the cuboid object in $[\mathrm{kg/m^3}]$.

Returns *obj* – An instance of the `vbCuboid` class.

setInitFrame(*varargin*)

Sets the origin frame of the cuboid to the given initial position and orientation.

The method can be called in three different ways:

- **setInitFrame**(*p*, *R*)
- **setInitFrame**(*vqT*)
- **setInitFrame**()

If no parameters are given, then `setInitFrame()` uses for the initial origin frame of the cuboid the current frame vector of the property *init_frame*.

Parameters **varargin** – Variable-length input argument list.

Keyword Arguments

- **p** (*double, vector*) – (3×1) Cartesian position to specify the *origin* of the cuboid.
- **R** (*double, matrix*) – (3×3) rotation matrix to specify the *orientation* of the cuboid.
- **vqT** (*double, vector*) – (7×1) VQ-transformation (position and orientation in quaternions) to specify the *origin frame* of the cuboid.

getGObj()

Creates and draws the cuboid object and returns a handle to the created graphics object.

Returns *hgo* – Handle to the patch graphics object that contains the data for all the polygons.

updGObj(*hgo*)

Updates the vertex coordinates of the cuboid.

Parameters **hgo** – Handle to the patch object that contains the data of all polygons of the cuboid.

Returns *hgo* – Handle to the patch object with the changed vertex coordinates of the polygons.

drawMGrid(*pt_color*)

Draws the meshgrid of the cuboid and returns a graphics object handle to it.

Parameters **pt_color** (*double/char, vector*) – Color of the grid points, specified by a RGB-triplet or a color name (default color: `'green'`).

Returns *hmg* – Handle to the scatter series object of the generated meshgrid.

ptInObj(*pt_pos*)

Determines if some specified points are below the surface, i.e. inside, of the cuboid.

Parameters **pt_pos** (*double, vector/matrix*) – A single position vector or a $(n \times 3)$ matrix with positions of some specified points.

Returns *result* – $(n \times 1)$ vector with the logical results if the given points are below the surface of the cuboid. Each value is *true* if the given point is below the surface, *false* otherwise.

See also:

`vbObject`, `vbCylinder` and `vbSphere`.

2.7 Basic Whole-Body Model Class

class `+WBM.@WBMBase.`**`WBMBase`**(*robot_model*)

Bases: `handle`

The class `WBMBase` is the *base class* of the *whole-body model* (WBM) for YARP-based floating-base robots, such as the iCub humanoid robot. It is a *wrapper class* for the fast *C++ MEX Whole-Body Model Interface* for Matlab (mex-wholeBodyModel) and provides all necessary basic methods of the yarpWholeBodyInterface for state estimations, kinematic/dynamic models and actuators.

`fixed_link`

char, vector – String matching URDF name of the default *fixed reference link frame* (floating-base link).

Note: The default fixed link (floating-base link) may be different for each robot and the selection of the floating base link depends also from the given situation. Furthermore, the specified fixed link (reference link) can also be a *contact constraint link*.

`init_wf_R_b`

double, matrix – (3×3) rotation matrix as *initial orientation* from base frame *b* to the world frame *wf* (default orientation: *identity matrix*).

`init_wf_p_b`

double, vector – (3×1) vector to specify the *initial position* from the base frame *b* to the world frame *wf* (default position: $[0, 0, 0]^T$).

`g_wf`

double, vector – (3×1) Cartesian gravity vector in the world frame *wf*. If the gravity is not defined, then the gravity vector is by default a 0-vector.

`ndof`

uint16, scalar – Number of degrees of freedom of the given robot.

`frict_coeff`

struct – Data structure for the friction coefficients of the joints of the given robot model (*read only*).

The structure contains following field variables:

- `v`: $(n_{dof} \times 1)$ coefficient vector for the *viscous friction*.
- `c`: $(n_{dof} \times 1)$ coefficient vector for the *Coulomb friction*.

`joint_limits`

struct – Data structure for the *joint positions limits* of the given robot model (*read only*).

The structure consists of two limits fields:

- `lwr`: $(n_{dof} \times 1)$ vector for the *lower* joint limits.
- `upr`: $(n_{dof} \times 1)$ vector for the *upper* joint limits.

`robot_model`

`wbmRobotModel` – Model object with the model parameters of the given floating-base robot (*read only*).

`DF_ROBOT_MODEL`

char, vector – Default *URDF robot model* for the whole-body interface: `'icubGazeboSim'` (*constant*).

Note: The URDF-file of robot model *icubGazeboSim* is located in the directory of the yarpWholeBodyInterface library.

`MAX_NUM_JOINTS`

int, scalar – Maximum number of joints that a robot model can have: 35 (*constant*).

`WBMBase`(*robot_model*)

Constructor.

The constructor initializes the given robot model and sets the world frame (wf) to the initial position and orientation with a specified gravitation.

Parameters **`robot_model`** (`wbmRobotModel`) – Model object with the model parameters of the given floating-base robot.

Returns *obj* – An instance of the `WBMBase` class.

`copy`()

Clones the current object.

This copy method replaces the `matlab.mixin.Copyable.copy()` method and creates a *deep copy* of the current object by using directly the memory.

Returns *newObj* – An exact copy (clone) of the current `WBMBase` object.

`delete`()

Destructor.

Removes all class properties from the workspace (free-up memory).

`initModel`(*urdf_model_name*)

Initializes the whole-body model of the floating-base robot.

The method can be called in two different modes:

Normal mode

Parameters **`urdf_model_name`** (*`char, vector`*) – String matching *name* of an existing robot model[3].

Optimized mode – *No arguments*.

The MEX whole-body interface loads the configuration files of the *default robot model* that is specified in the system environment variable `YARP_ROBOT_NAME`[3].

`initModelURDF`(*urdf_file_name*)

Initializes the whole-body model of a YARP-based robot with a specific URDF-file.

Parameters **`urdf_model_name`** (*`char, vector`*) – String with the full path to the URDF-file of the user defined robot model to be read.

`setWorldFrame`(*wf_R_b*, *wf_p_b*, *g_wf*)

Sets the world frame (wf) at a given *position* and *orientation* relative from a specified *fixed link frame* (base reference frame).

Parameters

- **`wf_R_b`** (*`double, matrix`*) – (3×3) rotation matrix (orientation) from the base frame *b* to world frame *wf*.
- **`wf_p_b`** (*`double, vector`*) – (3×1) position vector from the base frame *b* to the world frame *wf*.
- **`g_wf`** (*`double, vector`*) – (3×1) Cartesian gravity vector in the world frame *wf* (*optional*).

  If the gravity is not given, then the default gravity vector $[0, 0, \text{-}9.81]^T$ will be used.

Note: The *base reference frame*, also called *floating-base frame*, is attached to the *fixed reference link* of the robot. The specified fixed link (base reference link) of the robot can also be a *contact constraint link*.

`setInitWorldFrame`(*wf_R_b*, *wf_p_b*, *g_wf*)

Setup the world frame (wf) with the given *initial parameters* or update it with new parameters that are previously changed.

The method can be called in two ways:

- **setInitWorldFrame** (*wf_R_b*, *wf_p_b*[, *g_wf*])
- **setInitWorldFrame** ()

Parameters

- **wf_R_b** (*double, matrix*) – (3×3) rotation matrix (orientation) from the base frame *b* to world frame *wf*.
- **wf_p_b** (*double, vector*) – (3×1) position vector from the base frame *b* to the world frame *wf*.
- **g_wf** (*double, vector*) – (3×1) Cartesian gravity vector in the world frame *wf* (*optional*).

  If the gravity is not given, then the default gravity vector $[0, 0, \text{-}9.81]^T$ will be used.

Note: If no parameters are given, then `setInitWorldFrame` updates the world frame *wf* with the new initial parameters from the model configuration that were changed individually from outside.

See also:

`init_wf_R_b`, `init_wf_p_b` and `g_wf`.

getWorldFrameFromFixLnk (*urdf_fixed_link*, *q_j*)

Computes the *position* and *orientation* of the floating base w.r.t. a world frame (wf) that is intentionally set at a specified *fixed link frame* (base reference frame).

The *base reference frame*, also called *floating-base frame*, is attached to the *fixed reference link* of the robot. Since the most humanoid robots and other legged robots are not physically attached to the world, the *floating-base framework* provides a more general representation for the robot control. The returned position and orientation of the floating base is obtained from the *forward kinematics* w.r.t. the specified fixed link frame.

The method can be called by one of the given modes:

Normal mode – Computes the base frame for a specific joint configuration:

getWorldFrameFromFixLnk (*urdf_fixed_link*, *q_j*)

Optimized mode – Computes the base frame with the current joint configuration:

getWorldFrameFromFixLnk (*urdf_fixed_link*)

Parameters

- **urdf_fixed_link** (*char, vector*) – String matching *URDF name* of the fixed link as reference link.
- **q_j** (*double, vector*) – (n_{dof} x 1) joint positions vector in [rad] (*optional*).

Returns

[wf_p_b, wf_R_b] –

2-element tuple containing:

- **wf_p_b** (*double, vector*) – (3×1) position vector from the base frame *b* to the world frame *wf*.
- **wf_R_b** (*double, matrix*) – (3×3) rotation matrix (orientation) from the base frame *b* to the world frame *wf*.

Note: The specified fixed link (base reference link) can also be a *contact constraint link*.

getWorldFrameFromDfltFixLnk(*q_j*)

Computes the *position* and *orientation* of the floating base w.r.t. a world frame (wf) that is set to the default *fixed reference link frame*.

The method can be called by one of the given modes:

Normal mode – Computes the base frame for a specific joint configuration:

getWorldFrameFromFixLnk(*q_j*)

Optimized mode – Computes the base frame with the current joint configuration:

getWorldFrameFromFixLnk()

Parameters **q_j** (*double, vector*) – $(n_{dof} \times 1)$ joint positions vector in $[\text{rad}]$ (*optional*).

Returns

[wf_p_b, wf_R_b] –

2-element tuple containing:

- **wf_p_b** (*double, vector*) – (3×1) position vector from the base frame *b* to the world frame *wf*.
- **wf_R_b** (*double, matrix*) – (3×3) rotation matrix (orientation) from the base frame *b* to the world frame *wf*.

setState(*q_j*, *dq_j*, *v_b*)

Updates or sets the *state* of the robot model, i.e. the *joint angles*, *joint velocities* and the *generalized base velocity* of the floating base.

Parameters

- **q_j** (*double, vector*) – $(n_{dof} \times 1)$ joint positions vector in $[\text{rad}]$.
- **dq_j** (*double, vector*) – $(n_{dof} \times 1)$ joint velocities vector in $[\text{rad/s}]$.
- **v_b** (*double, vector*) – (6×1) generalized base velocity vector (Cartesian and rotational velocity of the base).

getState()

Obtains the currently stored *state* of the robot system, in particular

- the *base VQ-transformation*,
- the *joint angles* and *velocities* and
- the *generalized base velocity*.

Returns

[vqT_b, q_j, v_b, dq_j] –

4-element tuple containing:

- **vqT_b** (*double, vector*) – (7×1) VQ-transformation frame relative from the base *b* to the world frame *wf*[7].
- **q_j** (*double, vector*) – $(n_{dof} \times 1)$ joint angle vector in $[\text{rad}]$.
- **v_b** (*double, vector*) – (6×1) generalized base velocity vector.
- **dq_j** (*double, vector*) – $(n_{dof} \times 1)$ joint angle velocity vector in $[\text{rad/s}]$.

getFloatingBaseState()

Returns the current *state*, i.e. the position, orientation and velocity of the robot's floating base *b*.

[7] The first 3 elements of the vector-quaternion transformation represent the *position* and the last 4 elements of the vector define the *orientation* in *quaternions*.

Returns

stFltb (`wbmFltgBaseState`) –

Floating-base state object with the current state parameters of the robot's floating base *b* related to the world frame *wf*.

Note: The values of the floating-base state are derived from the currently stored state of the robot system.

See also:

`WBMBase.getState()`.

`transformationMatrix`(*varargin*)

Computes the *homogeneous transformation matrix* H of a specified link of the robot w.r.t. the current joint configuration q_j.

The method can be called by one of the given modes:

Normal mode – Computes the transformation matrix of the given link, specified by the base orientation and the positions:

`transformationMatrix`(*wf_R_b*, *wf_p_b*, *q_j*[, *urdf_link_name*])

Optimized mode – Computes the transformation matrix of the given link at the current state of the robot system:

`transformationMatrix`([*urdf_link_name*])

Parameters

- **`wf_R_b`** (*`double, matrix`*) – (3×3) rotation matrix (orientation) from the base frame *b* to world frame *wf*.
- **`wf_p_b`** (*`double, vector`*) – (3×1) position vector from the the base frame *b* to the world frame *wf*.
- **`q_j`** (*`double, vector`*) – $(n_{dof} \times 1)$ joint positions vector in [rad].
- **`urdf_link_name`** (*`char, vector`*) – String matching *URDF name* of the link (*optional*).

  Note: If the link is not given, then the method uses as default the *fixed link* of the robot.

Returns *wf_H_lnk (double, matrix)* – (4×4) homogeneous transformation matrix relative from the link frame *lnk* to the world frame *wf*.

`massMatrix`(*wf_R_b*, *wf_p_b*, *q_j*)

Computes the *generalized mass matrix* M of the floating-base robot w.r.t. the given joint configuration q_j.

The method can be called in two different modes:

Normal mode – Computes the mass matrix, specified by the base orientation and the positions:

Parameters

- **`wf_R_b`** (*`double, matrix`*) – (3×3) rotation matrix (orientation) from the base frame *b* to world frame *wf*.
- **`wf_p_b`** (*`double, vector`*) – (3×1) position vector from the base frame *b* to the world frame *wf*.
- **`q_j`** (*`double, vector`*) – $(n_{dof} \times 1)$ joint positions vector in [rad].

Optimized mode – *No arguments* (use the current state of the robot system).

Returns *M (double, matrix)* – $(n \times n)$ generalized mass matrix of the robot, where $n = n_{dof} + 6$.

getJointLimits()

Obtains the lower and upper *joint position limits* of the given robot model.

Returns

[jlmts_lwr, jlmts_upr] –

2-element tuple containing:

- **jlmts_lwr** (*double, vector*) – $(n_{dof} \times 1)$ lower joint limits vector.
- **jlmts_upr** (*double, vector*) – $(n_{dof} \times 1)$ upper joint limits vector.

isJointLimit(*q_j*)

Verifies, if the given joint configuration exceeds the upper or lower joint limits of the given robot.

Returns *resv (logical, vector)* – $(n_{dof} \times 1)$ vector with the logical results to determine if the current joint configuration is within the given limits of the robot. Each value is set to *true* if the corresponding joint has exceeded its limit, *false* otherwise.

generalizedBaseAcc(*M*, *c_qv*, *ddq_j*)

Computes the *generalized floating-base acceleration* for a *hybrid-dynamic system*.

This method may useful for example in inverse dynamics, when the base acceleration is unknown.

Parameters

- **M** (*double, matrix*) – $(n \times n)$ generalized mass matrix with $n = n_{dof} + 6$.
- **c_qv** (*double, vector*) – $(n \times 1)$ generalized bias force vector with $n = n_{dof} + 6$.
- **ddq_j** (*double, vector*) – $(n_{dof} \times 1)$ joint angle acceleration vector in $[\mathrm{rad/s}^2]$.

Returns *dv_b (double, vector)* – (6×1) generalized base acceleration vector of the floating base *b*.

See also:

`WBM.utilities.mbd.generalizedBaseAcc()`.

inverseDynamics(*varargin*)

Computes the inverse dynamics of a rigid-body system. It depends on the *state parameter triplet* $[q_j, \dot{q}_j, v_b]$, the *joint angle accelerations* $\ddot{q}_j$ and the *generalized base accelerations* $\dot{v}_b$.

If the *generalized floating-base acceleration* $\dot{v}_b$ is unknown, then the inverse dynamics problem changes to a *hybrid-dynamics problem* ([Fea08], p. 183). In general the equation of motion is given as follows:

$$\tau_j = M \cdot \ddot{q}_j + C(q_j, \dot{q}_j) + \tau_{fr} \tag{2.1}$$

where $C(q_j, \dot{q}_j)$ denotes the generalized bias force.

In contrast to the fixed-based system, the *equation of motion* for a *floating-base system* is defined as follows:

$$\begin{bmatrix} 0 \\ \tau_j \end{bmatrix} = \begin{bmatrix} M_{00} & M_{01} \\ M_{01}^T & M_{11} \end{bmatrix} \cdot \begin{bmatrix} \dot{v}_b \\ \ddot{q}_j \end{bmatrix} + \begin{bmatrix} h_0 \\ h_1 \end{bmatrix} + \begin{bmatrix} 0 \\ \tau_{fr} \end{bmatrix} \tag{2.2}$$

The matrix on the left side describes the $(n \times n)$ *mass matrix* $\mathbf{M}$ of the robot and the vector $h = [h_0, h_1]^T$ is a $(n \times 1)$ vector representing the *generalized bias forces* (Coriolis, centrifugal and gravity forces), where $n = n_{dof} + 6$. The inverse dynamics model will be solved by using the first row of the above equation (2.2) to obtain the *generalized base acceleration*:

$$\dot{v}_b = \text{-}M_{00}^{\text{-}1} \cdot (M_{01} \cdot \ddot{q}_j + h_0) \tag{2.3}$$

The method can be called by one of the given modes:

Normal mode – Computes the inverse dynamics of the robot, specified by the base orientation, the positions, the velocities and accelerations:

inverseDynamics (*wf_R_b*, *wf_p_b*, *q_j*, *dq_j*, *v_b*, *ddq_j*[, *dv_b*])

Optimized mode – Computes the inverse dynamics of the robot at the current state of the robot system:

inverseDynamics (*dq_j*, *ddq_j*[, *dv_b*])

Parameters

- **wf_R_b** (*double, matrix*) – (3×3) rotation matrix (orientation) from the base frame *b* to world frame *wf*.
- **wf_p_b** (*double, vector*) – (3×1) position vector from the base frame *b* to the world frame *wf*.
- **q_j** (*double, vector*) – $(n_{dof} \times 1)$ joint positions vector in $[\mathrm{rad}]$.
- **dq_j** (*double, vector*) – $(n_{dof} \times 1)$ joint angle velocity vector in $[\mathrm{rad/s}]$.
- **v_b** (*double, vector*) – (6×1) generalized base velocity vector (Cartesian and rotational velocity of the base).
- **ddq_j** (*double, vector*) – $(n_{dof} \times 1)$ joint angle acceleration vector in $[\mathrm{rad/s^2}]$.
- **dv_b** (*double, vector*) – (6×1) generalized base acceleration vector (*optional*).

Returns *tau_j (double, vector)* – $(n \times 1)$ generalized force vector of the joints and the base of the robot with $n = n_{dof} + 6$.

References

- *[Fea08]*, chapter 9.3-9.5, pp. 180-184, eq. (9.13) and (9.24).
- *[Kha11]*, p. 14, eq. (36) and (37).
- *[SSD12]*, p. 119, eq. (7.1) and (7.2).

jacobian (*varargin*)

Computes the *Jacobian* of a specified link of the robot w.r.t. the current joint configuration q_j.

The method can be called by one of the given modes:

Normal mode – Computes the Jacobian matrix of the given link, specified by the base orientation and the positions:

jacobian (*wf_R_b*, *wf_p_b*, *q_j*[, *urdf_link_name*])

Optimized mode – Computes the Jacobian matrix of the given link at the current state of the robot system:

jacobian ([*urdf_link_name*])

Parameters

- **wf_R_b** (*double, matrix*) – (3×3) rotation matrix (orientation) from the base frame *b* to world frame *wf*.
- **wf_p_b** (*double, vector*) – (3×1) position vector from the the base frame *b* to the world frame *wf*.

- **q_j** (*double, vector*) – $(n_{dof} \times 1)$ joint positions vector in [rad].
- **urdf_link_name** (*char, vector*) – String matching *URDF name* of the link (*optional*).

  Note: If the link is not given, then the method uses as default the *fixed link* of the robot.

Returns *wf_J_lnk (double, matrix)* – $(6 \times n)$ Jacobian matrix relative from the link frame *lnk* to the world frame *wf* with $n = n_{dof} + 6$.

dJdq (*varargin*)

Computes the *bias acceleration*, i.e. the *product* of the time derivative of the Jacobian and the joint velocities w.r.t. the current state and the specified link of the robot.

Note: The bias acceleration $\dot{J}\dot{q}_j$ is an acceleration that is not due to a robot acceleration.

This method is useful for the *Operational Space Control* and for the calculation of the *external contact constraints*. It can be called by one of the given modes:

Normal mode – Computes the bias acceleration of the given link, specified by the base orientation, the positions and the velocities:

dJdq (*wf_R_b*, *wf_p_b*, *q_j*, *dq_j*, *v_b*[, *urdf_link_name*])

Optimized mode – Computes the bias acceleration of the given link at the current state of the robot system:

dJdq ([*urdf_link_name*])

Parameters

- **wf_R_b** (*double, matrix*) – (3×3) rotation matrix (orientation) from the base frame *b* to world frame *wf*.
- **wf_p_b** (*double, vector*) – (3×1) position vector from the the base frame *b* to the world frame *wf*.
- **q_j** (*double, vector*) – $(n_{dof} \times 1)$ joint positions vector in [rad].
- **dq_j** (*double, vector*) – $(n_{dof} \times 1)$ joint angle velocity vector in [rad/s].
- **v_b** (*double, vector*) – (6×1) generalized base velocity vector (Cartesian and rotational velocity of the base).
- **urdf_link_name** (*char, vector*) – String matching *URDF name* of the link (*optional*).

  Note: If the link is not given, then the method uses as default the *fixed link* of the robot.

Returns *djdq_lnk (double, vector)* – (6×1) bias acceleration vector relative from the link frame *lnk* to the world frame *wf* in operational space.

centroidalMomentum (*wf_R_b*, *wf_p_b*, *q_j*, *dq_j*, *v_b*)

Computes the *centroidal momentum* h_c of a robot system.

The centroidal momentum of a humanoid robot is the sum of all link moments projected to the center of mass (CoM) of the robot.

The method is a function of the state variables $q_j, \dot{q}_j$ and the generalized base velocity v_b and can be called by one of the given modes:

Normal mode – Computes the centroidal momentum specified by the base orientation, the positions and the velocities:

Parameters

- **wf_R_b** (*double, matrix*) – (3×3) rotation matrix (orientation) from the base frame *b* to world frame *wf*.
- **wf_p_b** (*double, vector*) – (3×1) position vector from the base frame *b* to the world frame *wf*.
- **q_j** (*double, vector*) – $(n_{dof} \times 1)$ joint positions vector in $[\mathrm{rad}]$.
- **dq_j** (*double, vector*) – $(n_{dof} \times 1)$ joint angle velocity vector in $[\mathrm{rad/s}]$.
- **v_b** (*double, vector*) – (6×1) generalized base velocity vector (Cartesian and rotational velocity of the base).

Optimized mode – *No arguments* (use the current state of the robot system).

Returns *h_c (double, vector)* – (6×1) centroidal moment vector of the robot.

forwardKinematics (*varargin*)

Computes the *forward kinematic transformation* of a specific link w.r.t. the current joint configuration q_j.

The method can be called by one of the given modes:

Normal mode – Computes the forward kinematic transformation of the given link, specified by the base orientation and the positions:

forwardKinematics (*wf_R_b*, *wf_p_b*, *q_j*[, *urdf_link_name*])

Optimized mode – Computes the forward kinematic transformation of the given link at the current state of the robot system:

forwardKinematics ([*urdf_link_name*])

Parameters

- **wf_R_b** (*double, matrix*) – (3×3) rotation matrix (orientation) from the base frame *b* to world frame *wf*.
- **wf_p_b** (*double, vector*) – (3×1) position vector from the the base frame *b* to the world frame *wf*.
- **q_j** (*double, vector*) – $(n_{dof} \times 1)$ joint positions vector in $[\mathrm{rad}]$.
- **urdf_link_name** (*char, vector*) – String matching *URDF name* of the link (*optional*).

Returns *vqT_lnk (double, vector)* – (7×1) VQ-transformation frame relative from the link frame *lnk* to the world frame *wf*[7].

generalizedBiasForces (*wf_R_b*, *wf_p_b*, *q_j*, *dq_j*, *v_b*)

Computes the *generalized bias forces* $C(q_j, \dot{q}_j)$ in the dynamics of rigid-body systems.

It accounts the *Coriolis*, *centrifugal* and *gravity forces* that are depending on the joint angles q_j, the joint velocities $\dot{q}_j$ and the generalized base velocity v_b (*[Fea08]*, p. 40).

The method can be called in two different modes:

Normal mode – Computes the generalized bias forces, specified by the base orientation, the positions and the velocities:

Parameters

- **wf_R_b** (*double, matrix*) – (3×3) rotation matrix (orientation) from the base frame *b* to world frame *wf*.
- **wf_p_b** (*double, vector*) – (3×1) position vector from the base frame *b* to the world frame *wf*.
- **q_j** (*double, vector*) – $(n_{dof} \times 1)$ joint positions vector in $[\mathrm{rad}]$.
- **dq_j** (*double, vector*) – $(n_{dof} \times 1)$ joint angle velocity vector in $[\mathrm{rad/s}]$.

- **v_b** (*double, vector*) – (6×1) generalized base velocity vector (Cartesian and rotational velocity of the base).

Optimized mode – *No arguments* (use the current state of the robot system).

Returns *c_qv (double, vector)* – $(n \times 1)$ generalized bias force vector of the robot with $n = n_{dof} + 6$.

generalizedForces (*varargin*)

Computes the *generalized forces* τ_{gen} in dependency of some given external forces that are acting on the robot system, such that

$$\tau_{gen} = C(q_j, \dot{q}_j) - J_e^T f_e \;, \tag{2.4}$$

where $C(q_j, \dot{q}_j)$ denotes the *generalized bias forces* and $J_e^T f_e$ the *external forces* in joint space (*[Fea08]*, p. 40).

The method can be called in two different modes:

Normal mode – Computes the generalized forces, specified by the base orientation, the positions, velocities and the *external forces* in joint space:

generalizedForces (*wf_R_b, wf_p_b, q_j, dq_j, v_b, Je_t, f_e*)

Optimized mode – Computes the generalized forces at the current state of the robot system, in dependency of the given external force vector f_e and the Jacobian J_e^T:

generalizedForces (*Je_t, f_e*)

Parameters

- **wf_R_b** (*double, matrix*) – (3×3) rotation matrix (orientation) from the base frame *b* to world frame *wf*.
- **wf_p_b** (*double, vector*) – (3×1) position vector from the base frame *b* to the world frame *wf*.
- **q_j** (*double, vector*) – $(n_{dof} \times 1)$ joint positions vector in $[\text{rad}]$.
- **dq_j** (*double, vector*) – $(n_{dof} \times 1)$ joint angle velocity vector in $[\text{rad/s}]$.
- **v_b** (*double, vector*) – (6×1) generalized base velocity vector.
- **Je_t** (*double, matrix*) – $(n \times 6)$ transposed Jacobian to transform the external forces into the joint space with $n = n_{dof} + 6$.
- **f_e** (*double, vector*) – (6×1) external force vector that is acting on the robot system.

Returns *tau_gen (double, vector)* – $(n \times 1)$ generalized force vector of the robot with $n = n_{dof} + 6$.

See also:

WBMBase.generalizedBiasForces().

coriolisBiasForces (*wf_R_b, wf_p_b, q_j, dq_j, v_b*)

Computes the *bias Coriolis* and *centrifugal forces* $C(q_j, \dot{q}_j)$ in the dynamics of rigid-body systems.

The method can be called in two different modes:

Normal mode – Computes the bias Coriolis and centrifugal forces, specified by the base orientation, the positions and the velocities:

Parameters

- **wf_R_b** (*double, matrix*) – (3×3) rotation matrix (orientation) from the base frame *b* to world frame *wf*.
- **wf_p_b** (*double, vector*) – (3×1) position vector from the base frame *b* to the world frame *wf*.
- **q_j** (*double, vector*) – $(n_{dof} \times 1)$ joint positions vector in $[\text{rad}]$.
- **dq_j** (*double, vector*) – $(n_{dof} \times 1)$ joint angle velocity vector in $[\text{rad/s}]$.
- **v_b** (*double, vector*) – (6×1) generalized base velocity vector (Cartesian and rotational velocity of the base).

Optimized mode – *No arguments* (use the current state of the robot system).

Returns *c_qv (double, vector)* – $(n \times 1)$ bias Coriolis and centrifugal force vector of the robot with $n = n_{dof} + 6$.

Note: This method is similar to the generalized bias force method and depends on the same state parameter triplet $[q_j, \dot{q}_j, v_b]$.

See also:

WBMBase.generalizedBiasForces().

gravityBiasForces (*wf_R_b*, *wf_p_b*, *q_j*)

Computes the *gravity bias forces* $G(q_j)$ in the dynamics of rigid-body systems.

The method can be called in two different modes:

Normal mode – Computes the gravity bias forces, specified by the base orientation and the positions:

Parameters

- **wf_R_b** (*double, matrix*) – (3×3) rotation matrix (orientation) from the base frame *b* to world frame *wf*.
- **wf_p_b** (*double, vector*) – (3×1) position vector from the base frame *b* to the world frame *wf*.
- **q_j** (*double, vector*) – $(n_{dof} \times 1)$ joint positions vector in $[\text{rad}]$.

Optimized mode – *No arguments* (use the current state of the robot system).

Returns *g_q (double, vector)* – $(n \times 1)$ gravity bias force vector of the robot with $n = n_{dof} + 6$.

Note: This method is similar to the generalized bias force method, but depends only on the given joint configuration q_j.

See also:

WBMBase.generalizedBiasForces().

frictionForces (*dq_j*)

Computes the *frictional torque-forces* $F(\dot{q}_j)$ by using a simplified model.

The method depends on the current *joint angle acceleration* $\dot{q}_j$ and on the given *friction coefficients* of the joints. It can be called in two different modes:

Normal mode – Computes the frictional torque-forces in dependency of the current joint velocities:

Parameters **dq_j** (*double, vector*) – $(n_{dof} \times 1)$ joint angle velocity vector in $[\text{rad/s}]$.

Optimized mode – *No arguments* (use the current state of the robot system).

Returns *tau_fr (double, vector)* – $(n_{dof} \times 1)$ frictional torque-force vector with negated values.

Note: The friction coefficients of the joints must be predefined in `wbmRobotModel.frict_coeff`, otherwise the whole-body model assumes that the robot system is *frictionless*.

References

- *[SS00]*, p. 133 and p. 141.
- *[Cra05]*, pp. 188-189, eq. (6.110)-(6.112).
- *[Cor17]*, p. 252, eq. (9.1) and (9.2).

`wholeBodyDynamics`(*wf_R_b*, *wf_p_b*, *q_j*, *dq_j*, *v_b*)

Obtains the main components for the whole-body dynamics of a robot system, in particular it computes w.r.t. the state parameter triplet $[q_j, \dot{q}_j, v_b]$,

- the *generalized mass matrix*,
- the *generalized bias forces* and
- the *centroidal momentum*

of the given floating-base robot.

The method can be called in two different modes:

Normal mode – Computes the whole-body dynamics components, specified by the base orientation, the positions and the velocities:

Parameters

- **wf_R_b** (*double, matrix*) – (3×3) rotation matrix (orientation) from the base frame *b* to world frame *wf*.
- **wf_p_b** (*double, vector*) – (3×1) position vector from the base frame *b* to the world frame *wf*.
- **q_j** (*double, vector*) – $(n_{dof} \times 1)$ joint positions vector in $[\text{rad}]$.
- **dq_j** (*double, vector*) – $(n_{dof} \times 1)$ joint angle velocity vector in $[\text{rad/s}]$.
- **v_b** (*double, vector*) – (6×1) generalized base velocity vector (Cartesian and rotational velocity of the base).

Optimized mode – *No arguments* (use the current state of the robot system).

Returns

[M, c_qv, h_c] –

3-element tuple containing:

- **M** (*double, matrix*) – $(n \times n)$ generalized mass matrix of the robot,
- **c_qv** (*double, vector*) – $(n \times 1)$ generalized bias force vector,
- **h_c** (*double, vector*) – (6×1) centroidal moment vector of the robot,

with $n = n_{dof} + 6$.

`dispModel`(*prec*)

Displays the current model settings of the given floating-base robot.

Parameters **prec** (*int, scalar*) – Precision number to specify for the values the number of digits after the decimal point (default precision: 2) – *optional*.

2.8 Extended Whole-Body Model Class

class `+WBM.@WBM.`**`WBM`**(*robot_model*, *robot_config*, *wf2fixlnk*)
Bases: `WBM.WBMBase`

The class `WBM` is the *main class* of the *whole-body model* (WBM) for YARP-based floating-base robots. It is a derived class from the class `WBMBase` and offers additionally *higher-level methods* for the robot as extension, such as *visualization, forward dynamics, contact forces, payloads, tools*, etc.

`stvLen`
uint16, scalar – Length of the state parameter vector, i.e. the length of all state parameters concatenated in in one vector. The length can be either $2n_{dof} + 13$ (default), $2n_{dof} + 6$ (without `x_b` and `qt_b`) or *zero*, if *`init_state`* is empty[5] (*read only*).

`vqT_base`
double, vector – Base VQ-transformation frame (of the floating base) at the current state of the robot system (*read only*).

`init_vqT_base`
double, vector – Initial VQ-transformation frame of the robot's floating base (*read only*).

`init_stvChi`
double, vector – Initial state vector χ of the robot for the initial integration of the forward dynamics in state-space form[8] (*read only*).

Note: The state variable χ is a vector to express the forward dynamics in *state-space form* (*[Fea08]*, p. 42, eq. (3.8)) and is defined as, $\chi = [x_b, qt_b, q_j, \dot{x}_b, \omega_b, \dot{q}_j]^T$ with:

- x_b – Cartesian position of the floating base *b* in Euclidean space $\mathbb{R}^3$.
- qt_b – Orientation of the base *b* in *quaternions* (global parametrization of $\mathbf{SO}^3$).
- q_j – Joint positions (angles) of dimension $\mathbb{R}^{n_{dof}}$ in $[\mathrm{rad}]$.
- $\dot{x}_b$ – Cartesian velocity of the base *b* in Euclidean space $\mathbb{R}^3$.
- ω_b – Angular velocity describing the *rotational velocity* of the base *b* in $\mathbf{SO}^3$.

`init_state`
`wbmStateParams` – Data object to define the initial state parameters of the given floating-base robot.

`robot_body`
`wbmBody` – Data object that specifies the body components of the given floating-base robot (*read only*).

`robot_config`
`wbmRobotConfig` – Configuration object with the configuration settings of the given floating-base robot (*read only*).

`robot_params`
`wbmRobotParams` – Data object with the current settings of the model and configuration parameters of the given floating-base robot (*read only*).

Note: This property is useful to exchange the parameter settings of the robot between interfaces.

`DF_STIFFNESS`
int, scalar – Default stiffness control gain for the position correction: 2.5 (*constant*).

`MAX_NUM_TOOLS`
int, scalar – Maximum number of tools that a humanoid robot can use simultaneously: 2 (*constant*).

`MAX_JNT_SPEED`
int, scalar – Maximum joint speed for execution in $[\mathrm{ksps}]$ (kilosample(s) per second)[9]: 250 (*constant*).

[8] The state-space form reduces (through variable substitution) the inhomogeneous second-order ODE to a first-order ODE (*[Fea08]*, chapter 3, pp. 40-42, eq. (3.8)).
[9] Source from the *velocity control thread*: http://wiki.icub.org/brain/velControlThread_8cpp.html

MAX_JNT_ACC
int, scalar – Maximum joint acceleration with $1\,\mathrm{Ms/s}^2$ (Megasample(s) per second squared) that any joint is allowed to attempt[9] (*constant*).

MAX_JNT_TRQ
int, scalar – Maximum joint torque in [ksps] that any joint is allowed to attempt: 1×10^5 (*constant*).

ZERO_CVEC_12
int, scalar – (12×1) column-vector of zeros (for velocities, accelerations, forces, etc.) – *constant*.

ZERO_CVEC_6
int, scalar – (6×1) column-vector of zeros (for velocities, accelerations, etc.) – *constant*.

WBM(*robot_model*, *robot_config*, *wf2fixlnk*)
Constructor.

The constructor initializes the given model and configuration settings of the floating-base robot and sets the world frame (wf) to the initial position and orientation with a specified gravitation.

Parameters

- **robot_model** (`wbmRobotModel`) – Model object with the model parameters of the given floating-base robot.
- **robot_config** (`wbmRobotConfig`) – Configuration object with the configuration settings of the given floating-base robot.
- **wf2fixlnk** (*`logical, scalar`*) – Boolean flag to indicate if the world frame (wf) will be set to a fixed reference link frame (*optional*). Default: *false*.

  Note: If the fixed reference link is not specified in the given robot model, then the constructor uses as *default fixed link* the *first entry* of the given *contact constraint list*.

Returns *obj* – An instance of the `WBM` class.

copy()
Clones the current object.

This copy method replaces the `matlab.mixin.Copyable.copy()` method and creates a *deep copy* of the current object by using directly the memory.

Returns *newObj* – An exact copy (clone) of the current `WBM` object.

delete()
Destructor.

Removes all class properties from the workspace (free-up memory).

setWorldFrameAtFixLnk(*urdf_fixed_link*, *q_j*, *dq_j*, *v_b*, *g_wf*)
Sets the world frame (wf) at the computed *position* and *orientation* relative from a specified *fixed reference link* (base reference frame).

The *base reference frame*, also called *floating-base frame*, is attached to the *fixed reference link* of the robot. Since the most humanoid robots and other legged robots are not physically attached to the world, the *floating-base framework* provides a more general representation for the robot control. The position and orientation of the world frame, relative from the specified fixed link, is obtained from the *forward kinematics* w.r.t. the frame of the given fixed link. The specified fixed link (*floating-base link* or *base reference link*) can also be a *contact constraint link*.

The method can be called in two ways:

- **setWorldFrameAtFixLnk**(*urdf_fixed_link*, *q_j*, *dq_j*, *v_b*[, *g_wf*])
- **setWorldFrameAtFixLnk**(*urdf_fixed_link*)

Parameters

- **urdf_fixed_link** (*`char, vector`*) – String matching *URDF name* of the fixed link as reference link (frame).

- **q_j** (*double, vector*) – $(n_{dof} \times 1)$ joint positions vector in $[\mathrm{rad}]$ (*optional*).
- **dq_j** (*double, vector*) – $(n_{dof} \times 1)$ joint velocities vector in $[\mathrm{rad/s}]$ (*optional*).
- **v_b** (*double, vector*) – (6×1) generalized base velocity vector (*optional*).
- **g_wf** (*double, vector*) – (3×1) Cartesian gravity vector in the world frame *wf* (*optional*).

Note: If only the fixed link is specified, then by default the method uses for the position and orientation calculation (from the floating base to world frame) the predefined *initial state parameters* of the robot configuration.

See also:

`WBMBase.getWorldFrameFromFixLnk()` and `WBMBase.setWorldFrame()`.

`updateWorldFrameFromFixLnk`(*q_j*, *dq_j*, *v_b*, *g_wf*)

Updates the *position* and *orientation* of the world frame (wf) relative from the defined *fixed reference link* (base reference frame).

The position and orientation of the world frame (relative from the specified fixed link) is obtained from the *forward kinematics* w.r.t. the frame of the predefined fixed link.

The method can be called in two ways:

- **`updateWorldFrameFromFixLnk`**(*q_j*, *dq_j*, *v_b*[, *g_wf*])
- **`updateWorldFrameFromFixLnk`**()

Parameters

- **q_j** (*double, vector*) – $(n_{dof} \times 1)$ joint positions vector in $[\mathrm{rad}]$ (*optional*).
- **dq_j** (*double, vector*) – $(n_{dof} \times 1)$ joint velocities vector in $[\mathrm{rad/s}]$ (*optional*).
- **v_b** (*double, vector*) – (6×1) generalized base velocity vector (*optional*).
- **g_wf** (*double, vector*) – (3×1) Cartesian gravity vector in the world frame *wf* (*optional*).

Note: If no arguments are given, then by default the method uses for the position and orientation calculation (from the floating base to the world frame) the predefined *initial state parameters* of the robot configuration.

See also:

`WBM.setWorldFrameAtFixLnk()`, `WBMBase.getWorldFrameFromDfltFixLnk()` and `WBMBase.setWorldFrame()`.

`updateInitVQTransformation`()

Updates the initial state parameters of the *position* and *orientation* of the *floating base* with the base VQ-transformation values of the current state of the robot system.

`fkinVQTransformation`(*urdf_link_name*, *q_j*, *vqT_b*, *g_wf*)

Computes the forward kinematic *VQ-transformation frame* of a specific link of the floating-base robot.

The method can be called as follows:

`fkinVQTransformation`(*urdf_link_name*, *q_j*, *vqT_b*[, *g_wf*])

Parameters

- **urdf_link_name** (*char, vector*) – String matching *URDF name* of the link.

- **q_j** (`double, vector`) – $(n_{dof} \times 1)$ joint positions vector in $[\mathrm{rad}]$.
- **vqT_b** (`double, vector`) – (7×1) VQ-transformation frame relative from the base *b* to the world frame *wf*[7].
- **g_wf** (`double, vector`) – (3×1) Cartesian gravity vector in the world frame *wf* (*optional*).

Returns *vqT_lnk (double, vector)* – (7×1) VQ-transformation frame relative from the given link frame *lnk* to the world frame *wf*.

See also:

`WBMBase.forwardKinematics().`

`contactJacobians` (*varargin*)

Computes the *contact constraint Jacobian* and the *bias acceleration*, i.e. the *product* of the contact Jacobian derivative with the joint velocities of the floating-base robot.

The method can be called in two different modes:

Normal mode – Computes the contact Jacobian and the bias acceleration of the given contact links configuration, specified by the base orientation, the positions and the velocities:

`contactJacobians` (*wf_R_b_arr*, *wf_p_b*, *q_j*, *dq_j*, *v_b*[, *idx_list*])

Optimized mode – Computes the contact Jacobian and the bias acceleration of the given contact links configuration at the current state of the robot system:

`contactJacobians` (*clnk_conf*)

Parameters

- **wf_R_b_arr** (`double, vector`) – (9×1) rotation matrix reshaped in vector form from the base frame *b* to the world frame *wf* (*optional*).
- **wf_p_b** (`double, vector`) – (3×1) position vector from the base frame *b* to the world frame *wf* (*optional*).
- **q_j** (`double, vector`) – $(n_{dof} \times 1)$ joint positions vector in $[\mathrm{rad}]$ (*optional*).
- **dq_j** (`double, vector`) – $(n_{dof} \times 1)$ joint angle velocity vector in $[\mathrm{rad/s}]$ (*optional*).
- **v_b** (`double, vector`) – (6×1) generalized base velocity vector (*optional*).
- **idx_list** (`int, vector`) – $(1 \times k)$ vector with the index numbers of specific contact constraints in ascending order with $1 \leq k \leq n_{cstrs}$ (*optional*).

  Note: If the index list for the contact constraints is undefined, then the method uses automatically all defined contact constraints of the robot.

Returns

[Jc, djcdq] –

2-element tuple containing:

- **Jc** (*double, vector*) – $(6m \times n)$ Jacobian of the *contact constraints* in contact space $\mathrm{C} = \{\mathrm{C_1}, \ldots, \mathrm{C_k}\}$.
- **djcdq** (*double, vector*) – $(6m \times 1)$ bias acceleration vector $\dot{J}_c \dot{q}_j$ in contact space C.

The variable m denotes the *number of contact constraints* of the robot and $n = n_{dof} + 6$.

Note: If both contact links that are defined in the given configuration structure are (currently) not in contact with the ground or an object, then the arrays of the contact Jacobian J_c and of the product $\dot{J}_c \dot{q}_j$

will be set to zero. Furthermore, the bias acceleration $\dot{J}_c \dot{q}_j$ is an acceleration that is not due to a robot acceleration.

contactForces (*tau*, *Jc*, *djcdq*, *M*, *c_qv*, *dq_j*)

Computes the *contact forces* f_c of the given *contact constraints* of the floating-base robot, that are generated by the environment.

The formula for calculating the *contact constraint forces* is derived from the dynamic equations of motion of the robot, such that:

$$f_c = \Lambda_c \cdot (J_c M^{\text{-1}} \cdot (C(q_j, \dot{q}_j) - \tau_{gen}) - \dot{J}_c \dot{q}_j) \,, \tag{2.5}$$

where $\Lambda_c = \Upsilon_c^{\text{-1}} = (J_c M^{\text{-1}} J_c^T)^{\text{-1}}$ denotes the *inertia matrix* (or *pseudo-kinetic energy matrix* [Kha87]) in contact space $\mathrm{C} = \{\mathrm{C}_1, \ldots, \mathrm{C_k}\}$, $\tau_{gen} = S_j \cdot (\tau - \tau_{fr})$ represents the *generalized forces* with the *joint selection matrix* $S_j = [\mathbf{0}_{(6\times n)}\ \mathbb{I}_n]^T$ and the *friction forces* τ_{fr}.

The method can be called as follows:

contactForces (*tau*, *Jc*, *djcdq*, *M*, *c_qv*[, *dq_j*])

Parameters

- **tau** (*double, vector*) – $(n \times 1)$ torque force vector of the joints and the base of the robot.
- **Jc** (*double, matrix*) – $(6m \times n)$ Jacobian of the *contact constraints* in contact space $\mathrm{C} = \{\mathrm{C}_1, \ldots, \mathrm{C_k}\}$.
- **djcdq** (*double, vector*) – $(6m \times 1)$ product vector of the time derivative of the contact Jacobian $\dot{J}_c$ and the joint velocity $\dot{q}_j$.
- **M** (*double, matrix*) – $(n \times n)$ generalized mass matrix of the robot.
- **c_qv** (*double, vector*) – $(n \times 1)$ generalized bias force vector of the robot.
- **dq_j** (*double, vector*) – $(n_{dof} \times 1)$ joint angle velocity vector in $[\mathrm{rad/s}]$ (*optional*).

  Note: If the joint velocity is not given, then the method assumes that the robot system is *frictionless*.

The variable m denotes the *number of contact constraints* of the robot and $n = n_{dof} + 6$.

Returns

[f_c, tau_gen] –

2-element tuple containing:

- **f_c** (*double, vector*) – $(n \times 1)$ contact constraint force vector,
- **tau_gen** (*double, vector*) – $(n \times 1)$ generalized force vector of the robot,

with $n = n_{dof} + 6$.

See also:

`WBMBase.frictionForces()`.

References

- *[Par06]*, chapter 5, pp. 106-110, eq. (5.5)-(5.14).
- *[MLS94]*, pp. 269-270, eq. (6.5) and (6.6).

- *[Kha87]*, p. 49-50, eq. (51).

contactForcesEF (*tau*, *fe_c*, *ac*, *Jc*, *djcdq*, *M*, *c_qv*, *dq_j*)

Computes the *contact forces* f_c of the given *contact constraints* with *external forces* (*EF*) that are acting on the contact links of the robot.

The formula for calculating the *contact constraint forces* is derived from the dynamic equations of motion of the robot, such that:

$$f_c = \Lambda_c \cdot (a_c + J_c M^{-1} \cdot (C(q_j, \dot{q}_j) - \tau_{gen}) - \dot{J}_c \dot{q}_j) - f_e^C \tag{2.6}$$

The variable $\Lambda_c = \Upsilon_c^{-1} = (J_c M^{-1} J_c^T)^{-1}$ denotes the *inertia matrix* (or *pseudo-kinetic energy matrix [Kha87]*) in contact space $\mathrm{C} = \{\mathrm{C}_1, \ldots, \mathrm{C}_k\}$, $\tau_{gen} = S_j \cdot (\tau - \tau_{fr})$ represents the *generalized forces* with the *joint selection matrix* $S_j = [\mathbf{0}_{(6\times n)}\ \mathbb{I}_n]^T$ and the *friction forces* τ_{fr}. The variable a_c denotes the *mixed contact accelerations* and f_e^C represents the *external forces* that are acting on the *contact links* in contact space C.

The method can be called as follows:

contactForcesEF (*tau*, *fe_c*, *ac*, *Jc*, *djcdq*, *M*, *c_qv*[, *dq_j*])

Parameters

- **tau** (*double, vector*) – $(n \times 1)$ torque force vector for the joints and the base of the robot.
- **fe_c** (*double, vector*) – $(k \times 1)$ vector of external forces (in contact space) that are acting on the specified contact links with size of $k = 6$ (one link) or $k = 12$ (both links).
- **ac** (*double, vector*) – $(k \times 1)$ mixed acceleration vector for the *contact points* of the specified contact links with the size of $k = 6$ (one link) or $k = 12$ (both links).

  Note: If the given accelerations are very small, then the vector can also be *constant* or *zero*.
- **Jc** (*double, matrix*) – $(6 \cdot m \times n)$ Jacobian of the *contact constraints* in contact space $\mathrm{C} = \{\mathrm{C}_1, \ldots, \mathrm{C}_k\}$.
- **djcdq** (*double, vector*) – $(6m \times 1)$ product vector of the time derivative of the contact Jacobian $\dot{J}_c$ and the joint velocity $\dot{q}_j$.
- **M** (*double, matrix*) – $(n \times n)$ generalized mass matrix of the robot.
- **c_qv** (*double, vector*) – $(n \times 1)$ generalized bias force vector of the robot.
- **dq_j** (*double, vector*) – $(n_{dof} \times 1)$ joint angle velocity vector in $[\mathrm{rad/s}]$ (*optional*).

  Note: If the joint velocity is not given, then the method assumes that the robot system is *frictionless*.

The variable m denotes the *number of contact constraints* of the robot and $n = n_{dof} + 6$.

Returns

[f_c, tau_gen] –

2-element tuple containing:

- **f_c** (*double, vector*) – $(n \times 1)$ contact constraint force vector,
- **tau_gen** (*double, vector*) – $(n \times 1)$ generalized force vector of the robot,

with $n = n_{dof} + 6$.

See also:

WBM.contactForces(), *contactForcesCLPCEF()* and `WBMBase.frictionForces()`.

References

- *[Par06]*, chapter 5, pp. 106-110, eq. (5.5)-(5.14).
- *[MLS94]*, pp. 269-270, eq. (6.5) and (6.6).
- *[Kha87]*, p. 49-50, eq. (51).

contactForcesCLPCEF (*clnk_conf*, *tau*, *fe_c*, *ac*, *Jc*, *djcdq*, *M*, *c_qv*, *varargin*)

Computes the *contact forces* f_c of the given *contact constraints* with *contact link pose corrections* and additionally *external forces* (*CLPCEF*) that are acting on the contact links of the robot.

The formula for calculating the *contact constraint forces* is derived from the dynamic equations of motion of the robot, such that:

$$f_c = \Lambda_c \cdot (a_c + J_c M^{\text{-}1} \cdot (C(q_j, \dot{q}_j) - \tau_{gen}) - \dot{J}_c \dot{q}_j - k_v \dot{\varepsilon}_c - k_p \varepsilon_c) - f_e^C \tag{2.7}$$

The variable $\Lambda_c = \Upsilon_c^{\text{-}1} = (J_c M^{\text{-}1} J_c^T)^{\text{-}1}$ denotes the *inertia matrix* (or *pseudo-kinetic energy matrix* *[Kha87]*) in contact space $\mathrm{C} = \{\mathrm{C}_1, \ldots, \mathrm{C}_\mathrm{k}\}$, $\tau_{gen} = S_j \cdot (\tau - \tau_{fr})$ represents the *generalized forces* with the *joint selection matrix* $S_j = [\mathbf{0}_{(6\times n)}\ \mathbb{I}_n]^T$ and the *friction forces* τ_{fr}. The variable a_c in the formula denotes the *mixed contact accelerations* and the variables k_p and k_v are *stiffness* and *damping control gains* for the closed-loop system. The variables ε_c and $\dot{\varepsilon}_c = J_c\nu$ denoting the *position* and *velocity errors* of the position-regulation system with the *mixed generalized velocities* ν. The force vector f_e^C represents the *external forces* that are acting on the *contact links* in contact space C.

The method can be called as follows:

- **contactForcesCLPCEF** (*clnk_conf*, *tau*, *fe_c*, *ac*, *Jc*, *djcdq*, *M*, *c_qv*, *wf_R_b_arr*, *wf_p_b*, *q_j*[, *dq_j*], *nu*)
- **contactForcesCLPCEF** (*clnk_conf*, *tau*, *fe_c*, *ac*, *Jc*, *djcdq*, *M*, *c_qv*[, *dq_j*], *nu*)

Parameters

- **clnk_conf** (*struct*) – Configuration structure to specify the *qualitative state* of at most two *contact links*.

  The data structure specifies which link is currently in contact with the ground or an object. It specifies also the *desired poses*, *angular velocities* and *control gains* for the position-regulation system of the links.

- **tau** (*double, vector*) – $(n \times 1)$ torque force vector for the joints and the base of the robot.

- **fe_c** (*double, vector*) – $(k \times 1)$ vector of external forces (in contact space) that are affecting at the specified contact links with size of $k = 6$ (one link) or $k = 12$ (both links).

- **ac** (*double, vector*) – $(k \times 1)$ mixed acceleration vector for the *contact points* of the specified contact links with the size of $k = 6$ (one link) or $k = 12$ (both links).

  Note: If the given accelerations are very small, then the vector can also be *constant* or *zero*.

- **Jc** (*double, matrix*) – $(6m \times n)$ Jacobian of the *contact constraints* in contact space $\mathrm{C} = \{\mathrm{C}_1, \ldots, \mathrm{C}_\mathrm{k}\}$.

- **djcdq** (*double, vector*) – $(6m \times 1)$ product vector of the time derivative of the contact Jacobian $\dot{J}_c$ and the joint velocity $\dot{q}_j$.
- **M** (*double, matrix*) – $(n \times n)$ generalized mass matrix of the robot.
- **c_qv** (*double, vector*) – $(n \times 1)$ generalized bias force vector of the robot.
- **nu** (*double, vector*) – $((6+n_{dof}) \times 1)$ mixed generalized velocity vector (generalized base velocity and joint velocity).

The variable m denotes the *number of contact constraints* of the robot and $n = n_{dof} + 6$.

Other Parameters

- **wf_R_b_arr** (*double, vector*) – (9×1) rotation matrix reshaped in vector form from the base frame *b* to the world frame *wf* (*optional*).
- **wf_p_b** (*double, vector*) – (3×1) position vector from the base frame *b* to the world frame *wf* (*optional*).
- **q_j** (*double, vector*) – $(n_{dof} \times 1)$ joint positions vector in $[\text{rad}]$ (*optional*).
- **dq_j** (*double, vector*) – $(n_{dof} \times 1)$ joint angle velocity vector in $[\text{rad/s}]$ (*optional*).

  Note: If the joint velocity is not given, then the method assumes that the robot system is *frictionless*.

Returns

[f_c, tau_gen] –

2-element tuple containing:

- **f_c** (*double, vector*) – $(n \times 1)$ contact constraint force vector,
- **tau_gen** (*double, vector*) – $(n \times 1)$ generalized force vector of the robot,

with $n = n_{dof} + 6$.

See also:

WBM.contactForces(), *contactForcesEF()* and `WBMBase.frictionForces()`.

References

- *[Par06]*, chapter 5, pp. 106-110, eq. (5.5)-(5.14).
- *[MLS94]*, pp. 269-270, eq. (6.5) and (6.6).
- *[Kha87]*, p. 49-50, eq. (51).

wholeBodyDynamicsCS(*clnk_conf, varargin*)

Computes the *whole-body dynamics* of a floating-base robot in dependency of the current contact state (CS).

The method can be called in two different modes:

Normal mode – Computes the whole-body dynamics of the given contact links configuration, specified by the base orientation, the positions and the velocities:

wholeBodyDynamicsCS(*clnk_conf, wf_R_b_arr, wf_p_b, q_j, dq_j, v_b*)

Optimized mode – Computes the whole-body dynamics of the given contact links configuration at the current state of the robot system:

wholeBodyDynamicsCS(*clnk_conf*)

Parameters

- **`clnk_conf`** (`struct`) – Configuration structure to specify the *qualitative state* of at most two *contact links*.

  The data structure specifies which link is currently in contact with the ground or an object. It specifies also the *desired poses*, *angular velocities* and *control gains* for the position-regulation system of the links.

- **`wf_R_b_arr`** (`double, vector`) – (9×1) rotation matrix reshaped in vector form from the base frame *b* to the world frame *wf* (*optional*).
- **`wf_p_b`** (`double, vector`) – (3×1) position vector from the base frame *b* to the world frame *wf* (*optional*).
- **`q_j`** (`double, vector`) – $(n_{dof} \times 1)$ joint positions vector in $[\mathrm{rad}]$ (*optional*).
- **`dq_j`** (`double, vector`) – $(n_{dof} \times 1)$ joint angle velocity vector in $[\mathrm{rad/s}]$ (*optional*).
- **`v_b`** (`double, vector`) – (6×1) generalized base velocity vector (Cartesian and rotational velocity of the base) – *optional*.

Returns

[M, c_qv, Jc, djcdq] –

4-element tuple containing:

- **M** (*double, vector*) – $(n \times n)$ generalized mass matrix of the robot.
- **c_qv** (*double, vector*) – $(n \times 1)$ generalized bias force vector of the robot.
- **Jc** (*double, vector*) – $(6 \cdot m \times n)$ Jacobian of the *contact constraints* in contact space $\mathrm{C} = \{\mathrm{C}_1, \ldots, \mathrm{C}_\mathrm{k}\}$.
- **djcdq** (*double, vector*) – $(6 \cdot m \times 1)$ vector of the product of the contact Jacobian derivative $\dot{J}_c$ with the joint velocity $\dot{q}_j$.

The variable m denotes the *number of contact constraints* of the robot and $n = n_{dof} + 6$.

Note: If both contact links that are defined in the given configuration structure are (currently) not in contact with the ground or an object, then the arrays of the contact Jacobian J_c and of the product $\dot{J}_c \dot{q}_j$ will be set to zero.

See also:

`WBM.contactJacobiansCS()`.

`contactJacobiansCS` (*clnk_conf*, *varargin*)

Computes the *contact constraint Jacobian* and the *bias acceleration*, i.e. the product of the contact Jacobian derivative with the joint velocities of the floating-base robot, in dependency of the current contact state (CS).

The method can be called in two different modes:

Normal mode – Computes the contact Jacobian and the bias acceleration of the given contact links configuration, specified by the base orientation, the positions and the velocities:

`contactJacobiansCS` (*clnk_conf*, *wf_R_b_arr*, *wf_p_b*, *q_j*, *dq_j*, *v_b*)

Optimized mode – Computes the contact Jacobian and the bias acceleration of the given contact links configuration at the current state of the robot system:

`contactJacobiansCS` (*clnk_conf*)

Parameters

- **`clnk_conf`** (*`struct`*) – Configuration structure to specify the *qualitative state* of at most two *contact links*.

  The data structure specifies which link is currently in contact with the ground or an object. It specifies also the *desired poses*, *angular velocities* and *control gains* for the position-regulation system of the links.
- **`wf_R_b_arr`** (*`double, vector`*) – (9×1) rotation matrix reshaped in vector form from the base frame *b* to the world frame *wf* (*optional*).
- **`wf_p_b`** (*`double, vector`*) – (3×1) position vector from the base frame *b* to the world frame *wf* (*optional*).
- **`q_j`** (*`double, vector`*) – $(n_{dof} \times 1)$ joint positions vector in [rad] (*optional*).
- **`dq_j`** (*`double, vector`*) – $(n_{dof} \times 1)$ joint angle velocity vector in [rad/s] (*optional*).
- **`v_b`** (*`double, vector`*) – (6×1) generalized base velocity vector (*optional*).

Returns

[Jc, djcdq] –

2-element tuple containing:

- **Jc** (*double, vector*) – $(6m \times n)$ Jacobian of the *contact constraints* in contact space $\mathrm{C} = \{\mathrm{C}_1, \ldots, \mathrm{C}_\mathrm{k}\}$.
- **djcdq** (*double, vector*) – $(6m \times 1)$ bias acceleration vector $\dot{J}_c\dot{q}_j$ in contact space C.

The variable m denotes the *number of contact constraints* of the robot and $n = n_{dof} + 6$.

Note: If both contact links that are defined in the given configuration structure are (currently) not in contact with the ground or an object, then the arrays of the contact Jacobian J_c and of the product $\dot{J}_c\dot{q}_j$ will be set to zero. Furthermore, the bias acceleration $\dot{J}_c\dot{q}_j$ is an acceleration that is not due to a robot acceleration.

`jointAccelerations`(*tau*, *varargin*)

Calculates the *joint angle accelerations* $\ddot{q}_j$ of the given floating-base robot.

Using the dynamic equations of motion, the calculation of the *joint acceleration vector* $\ddot{q}_j$ for the closed chain is solved as follows:

$$\ddot{q}_j = M^{\text{-1}} \cdot (\tau_{gen} - C(q_j, \dot{q}_j) - (J_c^T \cdot \text{-}f_c)) \tag{2.8}$$

The method can be called in two different modes:

Normal mode – Computes the joint accelerations, specified by the base orientation, the positions and the velocities:

- **`jointAccelerations`**(*tau*, *wf_R_b*, *wf_p_b*, *q_j*, *dq_j*, *v_b*)
- **`jointAccelerations`**(*tau*, *wf_R_b*, *wf_p_b*, *q_j*, *nu*)

Optimized mode – Computes the joint accelerations at the current state of the robot system, in dependency of the given joint velocities:

`jointAccelerations`(*tau*[, *dq_j*])

Parameters

- **`tau`** (*`double, vector`*) – $(n \times 1)$ torque force vector for the joints and the base of the robot with $n = n_{dof} + 6$.

- **wf_R_b** (`double, matrix`) – (3×3) rotation matrix (orientation) from the base frame *b* to world frame *wf* (*optional*).
- **wf_p_b** (`double, vector`) – (3×1) position vector from the base frame *b* to the world frame *wf* (*optional*).
- **q_j** (`double, vector`) – $(n_{dof} \times 1)$ joint positions vector in $[\text{rad}]$ (*optional*).
- **dq_j** (`double, vector`) – $(n_{dof} \times 1)$ joint angle velocity vector in $[\text{rad/s}]$ (*optional*).
- **v_b** (`double, vector`) – (6×1) generalized base velocity vector (*optional*).
- **nu** (`double, vector`) – $((6+n_{dof}) \times 1)$ mixed generalized velocity vector (generalized base velocity and joint velocity) – *optional*.

Returns

[ddq_j[, fd_prms]] –

2-element tuple containing:

- **ddq_j** (*double, vector*) – $(n_{dof} \times 1)$ joint angle acceleration vector in $[\text{rad/s}^2]$.
- **fd_prms** (*struct*) – Data structure for the parameter values of the forward dynamics calculation with the fields `tau_gen` and `f_c` (*optional*).

  Note: The second output argument can be used for further calculations or for data logging.

See also:

`WBM.jointAccelerationsEF()`, *`WBM.jointAccelerationsPL()`* and *`WBM.contactForces()`*.

References

[Lil92], p. 82, eq. (5.2).

`jointAccelerationsEF` (*foot_conf*, *clnk_conf*, *tau*, *fe_c*, *ac*, *varargin*)

Calculates the *joint angle accelerations* $\ddot{q}_j$ with additional *external forces* (*EF*) that are acting on the specified contact links of the given floating-base robot – *experimental*.

The joint accelerations $\ddot{q}_j$ will be calculated as defined in equation (2.8). In order to obtain the joint accelerations from the robot system, the method computes at first the *contact forces* f_c of the feet and of the contact links, in dependency of the *external forces* that are acting on the contact links of the robot.

The method assumes that *only the feet* of the robot may have contact with the *ground* and can be called in two different modes:

Normal mode – Computes the joint accelerations, specified by the base orientation, the positions and the velocities:

`jointAccelerationsEF` (*foot_conf*, *clnk_conf*, *tau*, *fe_c*, *ac*[, *ac_f*], *wf_R_b*, *wf_p_b*, *q_j*, *dq_j*, *v_b*)

Optimized mode – Computes the joint accelerations at the current state of the robot system, in dependency of the given joint velocities:

`jointAccelerationsEF` (*foot_conf*, *clnk_conf*, *tau*, *fe_c*, *ac*[, *ac_f*], *dq_j*)

Note: This method is still untested and should be considered as *experimental*.

Parameters

- **foot_conf** (*struct*) – Configuration structure to specify the *qualitative state* of the feet.

  The data structure specifies which foot is currently in contact with the ground.
- **clnk_conf** (*struct*) – Configuration structure to specify the *qualitative state* of at most two *contact links*.

  The data structure specifies which link is currently in contact with the ground or an object.
- **tau** (*double, vector*) – $(n \times 1)$ torque force vector for the joints and the base of the robot with $n = n_{dof} + 6$.
- **fe_c** (*double, vector*) – $(k \times 1)$ vector of external forces (in contact space C) that are acting on the specified contact links.
- **ac** (*double, vector*) – $(k \times 1)$ mixed acceleration vector for the *contact points* of the contact links.

  Note: If the accelerations are very small, then the vector can also be *constant* or *zero*.
- **ac_f** (*double, vector*) – $(k \times 1)$ mixed acceleration vector for the specified *foot contact points* (*optional*).
- **wf_R_b** (*double, matrix*) – (3×3) rotation matrix (orientation) from the base frame *b* to world frame *wf* (*optional*).
- **wf_p_b** (*double, vector*) – (3×1) position vector from the base frame *b* to the world frame *wf* (*optional*).
- **q_j** (*double, vector*) – $(n_{dof} \times 1)$ joint positions vector in $[\mathrm{rad}]$ (*optional*).
- **dq_j** (*double, vector*) – $(n_{dof} \times 1)$ joint angle velocity vector in $[\mathrm{rad/s}]$.
- **v_b** (*double, vector*) – (6×1) generalized base velocity vector (*optional*).

The variable k indicates the *size* of the given *force* and *acceleration vectors* in dependency of the specified contact links and feet:

- $k = 6$ – only one link/foot is defined.
- $k = 12$ – both links/feet are defined.

The given *external forces* and *foot accelerations* are either *constant* or *zero*.

Returns

[ddq_j[, fd_prms]] –

2-element tuple containing:

- **ddq_j** (*double, vector*) – $(n_{dof} \times 1)$ joint angle acceleration vector in $[\mathrm{rad/s}^2]$.
- **fd_prms** (*struct*) – Data structure for the parameter values of the forward dynamics calculation with the fields `tau_gen`, `f_c`, `a_c` and `f_e` (*optional*).

  Note: The second output argument can be used for further calculations or for data logging.

See also:

WBM.jointAccelerations(), *WBM.jointAccelerationsPL()* and *WBM.contactForcesEF()*.

jointAccelerationsPL (*foot_conf, hand_conf, tau, fhTotCWrench, f_cp, varargin*)

Calculates the *joint angle accelerations* $\ddot{q}_j$ of a humanoid robot with *payload forces* (*PL*) that are acting on the hands of the robot.

Since the given robot is defined without a floating base, there are no contact forces f_c to the ground, i.e. $f_c = 0$. The behavior of the model is the same as a robot with a fixed base.

The joint accelerations $\ddot{q}_j$ will be calculated as defined in equation (2.8). In order to obtain the joint accelerations from the robot system, the method computes at first the *contact forces* f_c of the feet and of the contact links, in dependency of the *external forces* that are acting on the contact links of the robot.

The method assumes that *only the feet* of the robot may have contact with the *ground* and can be called in two different modes:

Normal mode – Computes the joint accelerations, specified by the base orientation, the positions and the velocities:

jointAccelerationsPL (*foot_conf*, *hand_conf*, *tau*, *fhTotCWrench*, *f_cp*[, *ac_f*], *wf_R_b*, *wf_p_b*, *q_j*, *dq_j*, *v_b*)

Optimized mode – Computes the joint accelerations at the current state of the robot system, in dependency of the given joint velocities:

jointAccelerationsPL (*foot_conf*, *hand_conf*, *tau*, *fhTotCWrench*, *f_cp*[, *ac_f*], *dq_j*)

Parameters

- **foot_conf** (*struct*) – Configuration structure to specify the *qualitative state* of the feet.

  The data structure specifies which foot is currently in contact with the ground.
- **hand_conf** (*struct*) – Configuration structure to specify the *qualitative state* of the hands.

  The data structure specifies which hand is currently in contact with the payload object.
- **tau** (*double, vector*) – $(n \times 1)$ torque force vector for the joints and the base of the robot with $n = n_{dof} + 6$.
- **fhTotCWrench** (*function_handle*) – Function handle to a specific *total contact wrench function* in *contact space* $\mathrm{C_h} = \{\mathrm{C_1}, \ldots, \mathrm{C_n}\}$ of the hands that will be applied by the robot model.
- **f_cp** (*double, vector*) – Force vector or scalar applied to a grasped object at the *contact points* ${}^{O}p_{C_i}$ from the contact frames $\{\mathrm{C_1}, \ldots, \mathrm{C_n}\}$ of the hands to the origin frame O at the CoM of the object.

  The vector length l of the applied forces depends on the chosen *contact model* and if only one hand or both hands are involved in grasping an object, such that $l = h \cdot s$ with size $s \in \{1, 3, 4\}$ and the number of hands $h \in \{1, 2\}$.

  Note: The z-axis of a contact frame $\mathrm{C_{i \in \{1,\ldots,n\}}}$ points in the direction of the inward surface normal at the point of contact ${}^{O}p_{C_i}$. If the chosen contact model is *frictionless*, then each applied force to the object is a scalar, otherwise a vector.
- **ac_f** (*double, vector*) – $(k \times 1)$ mixed acceleration vector for the specified *foot contact points* with the size of $k = 6$ (one foot) or $k = 12$ (both feet) – *optional*.

  Note: The given *foot accelerations* are either *constant* or *zero*.
- **wf_R_b** (*double, matrix*) – (3×3) rotation matrix (orientation) from the base frame *b* to world frame *wf* (*optional*).
- **wf_p_b** (*double, vector*) – (3×1) position vector from the base frame *b* to the world frame *wf* (*optional*).
- **q_j** (*double, vector*) – $(n_{dof} \times 1)$ joint positions vector in $[\mathrm{rad}]$ (*optional*).
- **dq_j** (*double, vector*) – $(n_{dof} \times 1)$ joint angle velocity vector in $[\mathrm{rad/s}]$.
- **v_b** (*double, vector*) – (6×1) generalized base velocity vector (*optional*).

Returns

[ddq_j[, fd_prms]] –

2-element tuple containing:

- **ddq_j** (*double, vector*) – $(n_{dof} \times 1)$ joint angle acceleration vector in $[\mathrm{rad/s^2}]$.
- **fd_prms** (*struct*) – Data structure for the parameter values of the forward dynamics calculation with the fields `tau_gen`, `f_c`, `a_c` and `f_pl` (*optional*).

  Note: The second output argument can be used for further calculations or for data logging.

See also:

`WBM.jointAccelerations()`, *`WBM.jointAccelerationsEF()`*, *`WBM.handAccelerations()`*, `WBM.handPayloadForces()` and *`WBM.contactForcesEF()`*.

`jointAccelerationsNFB`(*tau*, *varargin*)

Calculates the *joint angle accelerations* $\ddot{q}_j$ of a humanoid robot without a *floating base* (*NFB*) or by exclusion of it.

Since the given robot is defined without a floating base, there are no contact forces f_c to the ground, i.e. $f_c = 0$. The behavior of the model is the same as a robot with a fixed base.

The joint accelerations will be calculated as defined in equation (2.8) and in *[Lil92]*, but without the contact forces f_c, such that:

$$\ddot{q}_j = M^{\text{-1}} \cdot (\tau_{gen} - C(q_j, \dot{q}_j)) \tag{2.9}$$

The method can be called in two different modes:

Normal mode – Computes the joint accelerations, specified by the base orientation, the positions and the velocities:

- **`jointAccelerationsNFB`**(*tau*, *wf_R_b*, *wf_p_b*, *q_j*, *dq_j*, *v_b*)
- **`jointAccelerationsNFB`**(*tau*, *wf_R_b*, *wf_p_b*, *q_j*, *nu*)

Optimized mode – Computes the joint accelerations at the current state of the robot system, in dependency of the given whole-body dynamics and the joint velocities:

- **`jointAccelerationsNFB`**(*tau*, *M*, *c_qv*[, *dq_j*])
- **`jointAccelerationsNFB`**(*tau*[, *dq_j*])

Parameters

- **tau** (*double, vector*) – $(n \times 1)$ torque force vector for the joints and the base of the robot with $n = n_{dof} + 6$.
- **wf_R_b** (*double, matrix*) – (3×3) rotation matrix (orientation) from the base frame *b* to world frame *wf* (*optional*).
- **wf_p_b** (*double, vector*) – (3×1) position vector from the base frame *b* to the world frame *wf* (*optional*).
- **q_j** (*double, vector*) – $(n_{dof} \times 1)$ joint positions vector in $[\mathrm{rad}]$ (*optional*).
- **dq_j** (*double, vector*) – $(n_{dof} \times 1)$ joint angle velocity vector in $[\mathrm{rad/s}]$ (*optional*).
- **v_b** (*double, vector*) – (6×1) generalized base velocity vector (*optional*).
- **nu** (*double, vector*) – $((6 + n_{dof}) \times 1)$ mixed generalized velocity vector (generalized base velocity and joint velocity) – *optional*.

Other Parameters

- **M** (*double, matrix*) – $(n \times n)$ generalized mass matrix of the robot (*optional*).
- **c_qv** (*double, vector*) – $(n \times 1)$ generalized bias force vector of the robot (*optional*).

Returns

[ddq_j[, fd_prms]] –

2-element tuple containing:

- **ddq_j** (*double, vector*) – $(n_{dof} \times 1)$ joint angle acceleration vector in $[\mathrm{rad/s}^2]$.
- **fd_prms** (*struct*) – Data structure for the parameter values of the forward dynamics calculation with the field `tau_gen` (*optional*).

  Note: The second output argument can be used for further calculations or for data logging.

See also:

WBM.jointAccelerationsNFBEF() and *WBM.jointAccelerationsNFBPL()*.

References

[Lil92], p. 82, eq. (5.2).

jointAccelerationsNFBEF (*clnk_conf, tau, fe_c, ac, varargin*)

Calculates the *joint angle accelerations* $\ddot{q}_j$ of a humanoid robot with *external forces* that are acting on the contact links of the robot, but without a *floating base* (*NFBEF*) or by exclusion of it – *experimental*.

Since the given robot is defined without a floating base, there are no contact forces f_c to the ground, i.e. $f_c = 0$. The behavior of the model is the same as a robot with a fixed base.

The joint accelerations $\ddot{q}_j$ will be calculated as defined in equation (2.8), but without the contact forces of the feet. In order to obtain the joint accelerations from the robot system, the method computes at first the *contact forces* f_c of the contact links, in dependency of the *external forces* that are acting on them.

The method can be called in two different modes:

Normal mode – Computes the joint accelerations, specified by the base orientation, the positions and the velocities:

jointAccelerationsNFBEF (*clnk_conf, tau, fe_c, ac, wf_R_b, wf_p_b, q_j, dq_j, v_b*)

Optimized mode – Computes the joint accelerations at the current state of the robot system, in dependency of the given whole-body dynamics and the joint velocities:

jointAccelerationsNFBEF (*clnk_conf, tau, fe_c, ac*[, *M, c_qv*], *dq_j*)

Note: This method is still untested and should be considered as *experimental*.

Parameters

- **clnk_conf** (*struct*) – Configuration structure to specify the *qualitative state* of at most two *contact links*.

  The data structure specifies which link is currently in contact with an object or a wall.
- **tau** (*double, vector*) – $(n \times 1)$ torque force vector for the joints and the base of the robot with $n = n_{dof} + 6$.

- **`fe_c`** (*`double, vector`*) – $(k \times 1)$ vector of external forces (in contact space C) that are acting on the specified contact links.

  Note: The *external forces* are either *constant* or *zero*.

- **`ac`** (*`double, vector`*) – $(k \times 1)$ mixed acceleration vector for the *contact points* of the contact links.

  Note: If the accelerations are very small, then the vector can also be *constant* or *zero*.

- **`wf_R_b`** (*`double, matrix`*) – (3×3) rotation matrix (orientation) from the base frame *b* to world frame *wf* (*optional*).
- **`wf_p_b`** (*`double, vector`*) – (3×1) position vector from the base frame *b* to the world frame *wf* (*optional*).
- **`q_j`** (*`double, vector`*) – $(n_{dof} \times 1)$ joint positions vector in $[\mathrm{rad}]$ (*optional*).
- **`dq_j`** (*`double, vector`*) – $(n_{dof} \times 1)$ joint angle velocity vector in $[\mathrm{rad/s}]$.
- **`v_b`** (*`double, vector`*) – (6×1) generalized base velocity vector (*optional*).

The variable k indicates the *size* of the given *force* and *acceleration vectors* in dependency of the specified contact links:

- $k = 6$ – only one link is defined.
- $k = 12$ – both links are defined.

Other Parameters

- **M** (*double, matrix*) – $(n \times n)$ generalized mass matrix of the robot (*optional*).
- **c_qv** (*double, vector*) – $(n \times 1)$ generalized bias force vector of the robot (*optional*).

Returns

[ddq_j[, fd_prms]] –

2-element tuple containing:

- **ddq_j** (*double, vector*) – $(n_{dof} \times 1)$ joint angle acceleration vector in $[\mathrm{rad/s}^2]$.
- **fd_prms** (*struct*) – Data structure for the parameter values of the forward dynamics calculation with the fields `tau_gen`, `f_c` and `f_e` (*optional*).

  Note: The second output argument can be used for further calculations or for data logging.

See also:

`WBM.jointAccelerationsNFB()`, *`WBM.jointAccelerationsNFBPL()`* and *`WBM.contactForcesEF()`*.

`jointAccelerationsNFBPL` (*hand_conf*, *tau*, *fhTotCWrench*, *f_cp*, *varargin*)

Calculates the *joint angle accelerations* $\ddot{q}_j$ of a humanoid robot with *payload forces* that are acting on the hands of the robot, but without a *floating base* (*NFBPL*) or by exclusion of it.

Since the given robot is defined without a floating base, there are no contact forces f_c to the ground, i.e. $f_c = 0$. The behavior of the model is the same as a robot with a fixed base.

The joint accelerations $\ddot{q}_j$ will be calculated as defined in equation (2.8), but without the contact forces of the feet. In order to obtain the joint accelerations from the robot system, the method computes at first the *contact forces* f_c of the hands, in dependency of the *payload forces* that are acting on them.

The method can be called in two different modes:

Normal mode – Computes the joint accelerations, specified by the base orientation, the positions and the velocities:

jointAccelerationsNFBPL(*hand_conf*, *tau*, *fhTotCWrench*, *f_cp*, *wf_R_b*, *wf_p_b*, *q_j*, *dq_j*, *v_b*)

Optimized mode – Computes the joint accelerations at the current state of the robot system, in dependency of the given whole-body dynamics and the joint velocities:

jointAccelerationsNFBPL(*clnk_conf*, *tau*, *fhTotCWrench*, *f_cp*[, *M*, *c_qv*], *dq_j*)

Parameters

- **hand_conf** (*struct*) – Configuration structure to specify the *qualitative state* of the hands.

  The data structure specifies which hand is currently in contact with the payload object.
- **tau** (*double, vector*) – $(n \times 1)$ torque force vector for the joints and the base of the robot with $n = n_{dof} + 6$.
- **fhTotCWrench** (*function_handle*) – Function handle to a specific *total contact wrench function* in *contact space* $\mathrm{C_h} = \{\mathrm{C_1}, \ldots, \mathrm{C_n}\}$ of the hands that will be applied by the robot model.
- **f_cp** (*double, vector*) – Force vector or scalar applied to a grasped object at the *contact points* ${}^{O}p_{C_i}$ from the contact frames $\{\mathrm{C_1}, \ldots, \mathrm{C_n}\}$ of the hands to the origin frame O at the CoM of the object.

  The vector length l of the applied forces depends on the chosen *contact model* and if only one hand or both hands are involved in grasping an object, such that $l = h \cdot s$ with size $s \in \{1, 3, 4\}$ and the number of hands $h \in \{1, 2\}$.

  Note: The z-axis of a contact frame $\mathrm{C_{i \in \{1,\ldots,n\}}}$ points in the direction of the inward surface normal at the point of contact ${}^{O}p_{C_i}$. If the chosen contact model is *frictionless*, then each applied force to the object is a scalar, otherwise a vector.
- **wf_R_b** (*double, matrix*) – (3×3) rotation matrix (orientation) from the base frame *b* to world frame *wf* (*optional*).
- **wf_p_b** (*double, vector*) – (3×1) position vector from the base frame *b* to the world frame *wf* (*optional*).
- **q_j** (*double, vector*) – $(n_{dof} \times 1)$ joint positions vector in [rad] (*optional*).
- **dq_j** (*double, vector*) – $(n_{dof} \times 1)$ joint angle velocity vector in [rad/s].
- **v_b** (*double, vector*) – (6×1) generalized base velocity vector (*optional*).

Other Parameters

- **M** (*double, matrix*) – $(n \times n)$ generalized mass matrix of the robot (*optional*).
- **c_qv** (*double, vector*) – $(n \times 1)$ generalized bias force vector of the robot (*optional*).

Returns

[ddq_j[, fd_prms]] –

2-element tuple containing:

- **ddq_j** (*double, vector*) – $(n_{dof} \times 1)$ joint angle acceleration vector in $[\mathrm{rad/s^2}]$.
- **fd_prms** (*struct*) – Data structure for the parameter values of the forward dynamics calculation with the fields `tau_gen`, `f_c`, `a_c` and `f_pl` (*optional*).

  Note: The second output argument can be used for further calculations or for data logging.

See also:

WBM.jointAccelerationsNFB(), *WBM.jointAccelerationsNFBEF()*, WBM.handPayloadForces() and *WBM.contactForcesEF()*.

jointAccelerationsCLPCEF(*clnk_conf*, *tau*, *fe_c*, *ac*, *varargin*)

Calculates the *joint angle accelerations* $\ddot{q}_j$ with *contact link pose corrections* and additionally *external forces* (*CLPCEF*) that are acting on the specified contact links of the floating-base robot.

The joint accelerations $\ddot{q}_j$ will be calculated as defined in equation (2.8). For a *closed-loop control system* with *velocity* and *position regulation*, the calculated joint acceleration vector depends on the given *contact constraints* and on the additional *external forces* that are acting on the contact links of the floating-base robot.

The method assumes that *only the contact links* of the robot may have contact with the *ground* or an *object* and can be called in three different modes:

Normal mode – Computes the joint accelerations, specified by by the base orientation, the positions and the velocities:

jointAccelerationsCLPCEF(*clnk_conf*, *tau*, *fe_c*, *ac*, *wf_R_b*, *wf_p_b*, *q_j*, *dq_j*, *v_b*, *nu*)

Semi-optimized mode:

jointAccelerationsCLPCEF(*clnk_conf*, *tau*, *fe_c*, *ac*, *wf_R_b*, *wf_p_b*, *q_j*, *nu*)

Optimized mode – Computes the joint accelerations at the current state of the robot system, in dependency of the given whole-body dynamics and velocities:

- **jointAccelerationsCLPCEF**(*clnk_conf*, *tau*, *fe_c*, *ac*, *Jc*, *djcdq*, *M*, *c_qv*[, *dq_j*], *nu*)
- **jointAccelerationsCLPCEF**(*clnk_conf*, *tau*, *fe_c*, *ac*, *nu*)

Parameters

- **clnk_conf** (*struct*) – Configuration structure to specify the *qualitative state* of at most two *contact links*.

  The data structure specifies which link is currently in contact with the ground or an object. It specifies also the *desired poses*, *angular velocities* and *control gains* for the position-regulation system of the links.

- **tau** (*double, vector*) – $(n \times 1)$ torque force vector for the joints and the base of the robot with $n = n_{dof} + 6$.
- **fe_c** (*double, vector*) – $(k \times 1)$ vector of external forces (in contact space C) that are acting on the specified contact links.

  Note: The *external forces* are either *constant* or *zero*.

- **ac** (*double, vector*) – $(k \times 1)$ mixed acceleration vector for the *contact points* of the contact links.

  Note: If the accelerations are very small, then the vector can also be *constant* or *zero*.

- **wf_R_b** (*double, matrix*) – (3×3) rotation matrix (orientation) from the base frame *b* to world frame *wf* (*optional*).
- **wf_p_b** (*double, vector*) – (3×1) position vector from the base frame *b* to the world frame *wf* (*optional*).
- **q_j** (*double, vector*) – $(n_{dof} \times 1)$ joint positions vector in $[\text{rad}]$ (*optional*).
- **dq_j** (*double, vector*) – $(n_{dof} \times 1)$ joint angle velocity vector in $[\text{rad/s}]$ (*optional*).
- **v_b** (*double, vector*) – (6×1) generalized base velocity vector (*optional*).
- **nu** (*double, vector*) – $((6 + n_{dof}) \times 1)$ mixed generalized velocity vector (generalized base velocity and joint velocity).

The variable k indicates the *size* of the given *force* and *acceleration vectors* in dependency of the specified contact links:

- $k = 6$ – only one link is defined.
- $k = 12$ – both links are defined.

If the joint velocity vector $\dot{q}_j$ (`dq_j`) is not given as an argument, then the method assumes that the robot system is *frictionless*.

Other Parameters

- **Jc** (*double, matrix*) – $(6m \times n)$ Jacobian of the *contact constraints* in contact space $\mathrm{C} = \{\mathrm{C_1}, \ldots, \mathrm{C_k}\}$ (*optional*).
- **djcdq** (*double, vector*) – $(6m \times 1)$ product vector of the time derivative of the contact Jacobian $\dot{J}_c$ and the joint velocity $\dot{q}_j$ (*optional*).
- **M** (*double, matrix*) – $(n \times n)$ generalized mass matrix of the robot (*optional*).
- **c_qv** (*double, vector*) – $(n \times 1)$ generalized bias force vector of the robot (*optional*).

The variable m denotes the *number of contact constraints* of the robot and $n = n_{dof} + 6$.

Returns

[ddq_j[, fd_prms]] –

2-element tuple containing:

- **ddq_j** (*double, vector*) – $(n_{dof} \times 1)$ joint angle acceleration vector in $[\mathrm{rad/s^2}]$.
- **fd_prms** (*struct*) – Data structure for the parameter values of the forward dynamics calculation with the fields `tau_gen`, `f_c`, `a_c` and `f_e` (*optional*).

  Note: The second output argument can be used for further calculations or for data logging.

See also:

WBM.jointAccelerationsFPC(), *WBM.jointAccelerationsHPCEF()* and *WBM.contactForcesCLPCEF()*.

jointAccelerationsFPC (*foot_conf, tau, ac_f, varargin*)

Calculates the *joint angle accelerations* $\ddot{q}_j$ with *foot pose corrections* (*FPC*), in dependency of the given contact constraints of the floating-base robot.

The equation for solving the joint accelerations $\ddot{q}_j$ of a closed chain is derived from the dynamic equations of motion (see (2.8)). In order to obtain the joint accelerations from the *closed-loop control system* with *position-regulation* (velocity and position correction) of the feet, the method computes at first the contact forces f_c of the feet and does not include any external forces that may affecting the system.

The method assumes that *only the feet* of the robot may have contact with the *ground* and can be called in three different modes:

Normal mode – Computes the joint accelerations, specified by the base orientation, the positions and the velocities:

jointAccelerationsFPC (*foot_conf, tau, ac_f, wf_R_b, wf_p_b, q_j, dq_j, v_b, nu*)

Semi-optimized mode:

jointAccelerationsFPC (*foot_conf, tau, ac_f, wf_R_b, wf_p_b, q_j, nu*)

Optimized mode – Computes the joint accelerations at the current state of the robot system, in dependency of the given whole-body dynamics and velocities:

- **jointAccelerationsFPC** (*foot_conf, tau, ac_f, Jc, djcdq, M, c_qv*[, *dq_j*], *nu*)
- **jointAccelerationsFPC** (*foot_conf, tau, ac_f, nu*)

Note: If the joint velocity vector $\dot{q}_j$ (`dq_j`) is not given as an argument, then the method assumes that the robot system is *frictionless*.

Parameters

- **`foot_conf`** (*`struct`*) – Configuration structure to specify the *qualitative state* of the feet.

  The data structure specifies which foot is currently in contact with the ground. It specifies also the *desired poses*, *angular velocities* and *control gains* for the position-regulation system of the feet.

- **`tau`** (*`double, vector`*) – $(n \times 1)$ torque force vector for the joints and the base of the robot with $n = n_{dof} + 6$.
- **`ac_f`** (*`double, vector`*) – $(k \times 1)$ mixed acceleration vector for the specified *foot contact points* with the size of $k = 6$ or $k = 12$.

  Note: The given *foot accelerations* are either *constant* or *zero*. If $k = 6$, then only one foot touches the ground, else both feet.

- **`wf_R_b`** (*`double, matrix`*) – (3×3) rotation matrix (orientation) from the base frame *b* to world frame *wf* (*optional*).
- **`wf_p_b`** (*`double, vector`*) – (3×1) position vector from the base frame *b* to the world frame *wf* (*optional*).
- **`q_j`** (*`double, vector`*) – $(n_{dof} \times 1)$ joint positions vector in $[\text{rad}]$ (*optional*).
- **`dq_j`** (*`double, vector`*) – $(n_{dof} \times 1)$ joint angle velocity vector in $[\text{rad/s}]$ (*optional*).
- **`v_b`** (*`double, vector`*) – (6×1) generalized base velocity vector (*optional*).
- **`nu`** (*`double, vector`*) – $((6 + n_{dof}) \times 1)$ mixed generalized velocity vector (generalized base velocity and joint velocity).

Other Parameters

- **Jc** (*double, matrix*) – $(6m \times n)$ Jacobian of the *contact constraints* in contact space $\mathrm{C} = \{\mathrm{C_1}, \ldots, \mathrm{C_k}\}$ (*optional*).
- **djcdq** (*double, vector*) – $(6m \times 1)$ product vector of the time derivative of the contact Jacobian $\dot{J}_c$ and the joint velocity $\dot{q}_j$ (*optional*).
- **M** (*double, matrix*) – $(n \times n)$ generalized mass matrix of the robot (*optional*).
- **c_qv** (*double, vector*) – $(n \times 1)$ generalized bias force vector of the robot (*optional*).

The variable m denotes the *number of contact constraints* of the robot and $n = n_{dof} + 6$.

Returns

[ddq_j[, fd_prms]] –

2-element tuple containing:

- **ddq_j** (*double, vector*) – $(n_{dof} \times 1)$ joint angle acceleration vector in $[\text{rad/s}^2]$.
- **fd_prms** (*struct*) – Data structure for the parameter values of the forward dynamics calculation with the fields `tau_gen`, `f_c`, `a_c` and `f_e` (*optional*).

  Note: The second output argument can be used for further calculations or for data logging.

See also:

WBM.jointAccelerationsFPCEF(), *WBM.jointAccelerationsFPCPL()* and *WBM.jointAccelerationsCLPCEF()*.

jointAccelerationsFPCEF (*foot_conf*, *clnk_conf*, *tau*, *fe_c*, *ac*, *varargin*)

Calculates the *joint angle accelerations* $\ddot{q}_j$ with *foot pose corrections* and and additionally *external forces* (*FPCEF*) that are acting on the specified contact links of the floating-base robot – *experimental*.

The equation for solving the joint accelerations $\ddot{q}_j$ of a closed chain is derived from the dynamic equations of motion (see (2.8)). In order to obtain the joint accelerations from the *closed-loop control system* with *position-regulation* (velocity and position correction) of the feet, the method computes at first the *contact forces* f_c of the feet and of the contact links, in dependency of the *external forces* that are acting on the contact links of the robot.

The method assumes that *only the feet* of the robot may have contact with the *ground* and can be called in two different modes:

Normal mode – Computes the joint accelerations, specified by the base orientation, the positions and the velocities:

jointAccelerationsFPCEF (*foot_conf*, *clnk_conf*, *tau*, *fe_c*, *ac*[, *ac_f*], *wf_R_b*, *wf_p_b*, *q_j*, *dq_j*, *v_b*, *nu*)

Optimized mode – Computes the joint accelerations at the current state of the robot system, in dependency of the given whole-body dynamics and velocities:

- **jointAccelerationsFPCEF** (*foot_conf*, *clnk_conf*, *tau*, *fe_c*, *ac*[, *ac_f*], *Jc_f*, *djcdq_f*, *M*, *c_qv*, *dq_j*, *nu*)
- **jointAccelerationsFPCEF** (*foot_conf*, *clnk_conf*, *tau*, *fe_c*, *ac*[, *ac_f*], *nu*)

Note: This method is still untested and should be considered as *experimental*.

Parameters

- **foot_conf** (`struct`) – Configuration structure to specify the *qualitative state* of the feet.

  The data structure specifies which foot is currently in contact with the ground. It specifies also the *desired poses*, *angular velocities* and *control gains* for the position-regulation system of the feet.

- **clnk_conf** (`struct`) – Configuration structure to specify the *qualitative state* of at most two *contact links*.

  The data structure specifies which link is currently in contact with the ground or an object.

- **tau** (`double, vector`) – $(n \times 1)$ torque force vector for the joints and the base of the robot with $n = n_{dof} + 6$.
- **fe_c** (`double, vector`) – $(k \times 1)$ vector of external forces (in contact space C) that are acting on the specified contact links.
- **ac** (`double, vector`) – $(k \times 1)$ mixed acceleration vector for the *contact points* of the contact links.

  Note: If the accelerations are very small, then the vector can also be *constant* or *zero*.

- **ac_f** (`double, vector`) – $(k \times 1)$ mixed acceleration vector for the specified *foot contact points* (*optional*).
- **wf_R_b** (`double, matrix`) – (3×3) rotation matrix (orientation) from the base frame *b* to world frame *wf* (*optional*).

- **wf_p_b** (*double, vector*) – (3×1) position vector from the base frame *b* to the world frame *wf* (*optional*).
- **q_j** (*double, vector*) – $(n_{dof} \times 1)$ joint positions vector in $[\mathrm{rad}]$ (*optional*).
- **dq_j** (*double, vector*) – $(n_{dof} \times 1)$ joint angle velocity vector in $[\mathrm{rad/s}]$ (*optional*).
- **v_b** (*double, vector*) – (6×1) generalized base velocity vector (*optional*).
- **nu** (*double, vector*) – $((6+n_{dof}) \times 1)$ mixed generalized velocity vector (generalized base velocity and joint velocity).

The variable k indicates the *size* of the given *force* and *acceleration vectors* in dependency of the specified contact links and feet:

- $k = 6$ – only one link/foot is defined.
- $k = 12$ – both links/feet are defined.

The given *external forces* and *foot accelerations* are either *constant* or *zero*.

Other Parameters

- **Jc** (*double, matrix*) – $(6m \times n)$ Jacobian of the *contact constraints* in contact space $\mathrm{C} = \{\mathrm{C}_1, \ldots, \mathrm{C_k}\}$ (*optional*).
- **djcdq** (*double, vector*) – $(6m \times 1)$ product vector of the time derivative of the contact Jacobian $\dot{J}_c$ and the joint velocity $\dot{q}_j$ (*optional*).
- **M** (*double, matrix*) – $(n \times n)$ generalized mass matrix of the robot (*optional*).
- **c_qv** (*double, vector*) – $(n \times 1)$ generalized bias force vector of the robot (*optional*).

The variable m denotes the *number of contact constraints* of the robot and $n = n_{dof} + 6$.

Returns

[ddq_j[, fd_prms]] –

2-element tuple containing:

- **ddq_j** (*double, vector*) – $(n_{dof} \times 1)$ joint angle acceleration vector in $[\mathrm{rad/s^2}]$.
- **fd_prms** (*struct*) – Data structure for the parameter values of the forward dynamics calculation with the fields `tau_gen`, `f_c`, `a_c` and `f_e` (*optional*).

  Note: The second output argument can be used for further calculations or for data logging.

See also:

WBM.jointAccelerationsFPC(), *WBM.jointAccelerationsFPCL()* and *WBM.contactForcesEF()*.

jointAccelerationsFPCPL (*foot_conf*, *hand_conf*, *tau*, *fhTotCWrench*, *f_cp*, *varargin*)

Calculates the *joint angle accelerations* $\ddot{q}_j$ with *foot pose corrections* and additionally *payload forces* (*FPCPL*) that are acting on the hands of the humanoid robot.

The equation for solving the joint accelerations $\ddot{q}_j$ of a closed chain is derived from the dynamic equations of motion (see (2.8)). In order to obtain the joint accelerations from the *closed-loop control system* with *position-regulation* (velocity and position correction) of the feet, the method computes at first the *contact forces* f_c of the feet and of the hands, in dependency of the *payload forces* that are acting on the hands of the robot.

The method assumes that *only the feet* of the robot may have contact with the *ground* and can be called in two different modes:

Normal mode – Computes the joint accelerations, specified by the base orientation, the positions and the velocities:

jointAccelerationsFPCPL(*foot_conf*, *hand_conf*, *tau*, *fhTotCWrench*, *f_cp*[, *ac_f*], *wf_R_b*, *wf_p_b*, *q_j*, *dq_j*, *v_b*, *nu*)

Optimized mode – Computes the joint accelerations at the current state of the robot system, in dependency of the given whole-body dynamics and velocities:

- **jointAccelerationsFPCPL**(*foot_conf*, *hand_conf*, *tau*, *fhTotCWrench*, *f_cp*[, *ac_f*], *Jc_f*, *djcdq_f*, *M*, *c_qv*, *dq_j*, *nu*)
- **jointAccelerationsFPCPL**(*foot_conf*, *hand_conf*, *tau*, *fhTotCWrench*, *f_cp*[, *ac_f*], *dq_j*, *v_b*, *nu*)

Parameters

- **foot_conf** (*struct*) – Configuration structure to specify the *qualitative state* of the feet.

  The data structure specifies which foot is currently in contact with the ground. It specifies also the *desired poses*, *angular velocities* and *control gains* for the position-regulation system of the feet.

- **hand_conf** (*struct*) – Configuration structure to specify the *qualitative state* of the hands.

  The data structure specifies which hand is currently in contact with the payload object.

- **tau** (*double, vector*) – $(n \times 1)$ torque force vector for the joints and the base of the robot with $n = n_{dof} + 6$.
- **fhTotCWrench** (*function_handle*) – Function handle to a specific *total contact wrench function* in *contact space* $\mathrm{C_h} = \{\mathrm{C_1}, \ldots, \mathrm{C_n}\}$ of the hands that will be applied by the robot model.
- **f_cp** (*double, vector*) – Force vector or scalar applied to a grasped object at the *contact points* ${}^{O}p_{C_i}$ from the contact frames $\{\mathrm{C_1}, \ldots, \mathrm{C_n}\}$ of the hands to the origin frame O at the CoM of the object.

  The vector length l of the applied forces depends on the chosen *contact model* and if only one hand or both hands are involved in grasping an object, such that $l = h \cdot s$ with size $s \in \{1, 3, 4\}$ and the number of hands $h \in \{1, 2\}$.

  Note: The z-axis of a contact frame $\mathrm{C_{i \in \{1,\ldots,n\}}}$ points in the direction of the inward surface normal at the point of contact ${}^{O}p_{C_i}$. If the chosen contact model is *frictionless*, then each applied force to the object is a scalar, otherwise a vector.

- **ac_f** (*double, vector*) – $(k \times 1)$ mixed acceleration vector for the specified *foot contact points* (*optional*).
- **wf_R_b** (*double, matrix*) – (3×3) rotation matrix (orientation) from the base frame *b* to world frame *wf* (*optional*).
- **wf_p_b** (*double, vector*) – (3×1) position vector from the base frame *b* to the world frame *wf* (*optional*).
- **q_j** (*double, vector*) – $(n_{dof} \times 1)$ joint positions vector in $[\mathrm{rad}]$ (*optional*).
- **dq_j** (*double, vector*) – $(n_{dof} \times 1)$ joint angle velocity vector in $[\mathrm{rad/s}]$ (*optional*).
- **v_b** (*double, vector*) – (6×1) generalized base velocity vector (*optional*).
- **nu** (*double, vector*) – $((6 + n_{dof}) \times 1)$ mixed generalized velocity vector (generalized base velocity and joint velocity).

The variable k indicates the *size* of the given *force* and *acceleration vectors* in dependency of the specified feet and hands:

- $k = 6$ – only one foot/hand is defined.

- $k = 12$ – both feet/hands are defined.

The given *external forces* and *foot accelerations* are either *constant* or *zero*.

Other Parameters

- **Jc_f** (*double, matrix*) – $(6m \times n)$ Jacobian of the *foot contact constraints* in contact space C_f (*optional*).
- **djcdq_f** (*double, vector*) – $(6m \times 1)$ product vector of the time derivative of the contact Jacobian $\dot{J}_c^f$ for the feet and the joint velocity $\dot{q}_j$ (*optional*).
- **M** (*double, matrix*) – $(n \times n)$ generalized mass matrix of the robot (*optional*).
- **c_qv** (*double, vector*) – $(n \times 1)$ generalized bias force vector of the robot (*optional*).

The variable m denotes the *number of contact constraints* of the robot and $n = n_{dof} + 6$.

Returns

[ddq_j[, fd_prms]] –

2-element tuple containing:

- **ddq_j** (*double, vector*) – $(n_{dof} \times 1)$ joint angle acceleration vector in $[\mathrm{rad/s}^2]$.
- **fd_prms** (*struct*) – Data structure for the parameter values of the forward dynamics calculation with the fields `tau_gen`, `f_c`, `a_c` and `f_pl` (*optional*).

  Note: The second output argument can be used for further calculations or for data logging.

See also:

WBM.jointAccelerationsFPC(), *WBM.jointAccelerationsFPCEF()*, `WBM.handPayloadForces()` and *WBM.contactForcesEF()*.

jointAccelerationsHPCEF(*hand_conf*, *tau*, *fe_h*, *ac_h*, *varargin*)

Calculates the *joint angle accelerations* $\ddot{q}_j$ with *hand pose corrections* and additionally *external forces* (*HPCEF*) that are acting on the hands of the humanoid robot.

The joint accelerations $\ddot{q}_j$ will be calculated as defined in equation (2.8). For a *closed-loop control system* with *velocity* and *position regulation*, the calculated joint acceleration vector depends on the given *contact constraints* and the additional *external forces* that are acting on the hands of the floating-base robot.

The method assumes that *only the hands* of the robot may have contact with the *ground/object/wall* and can be called in three different modes:

Normal mode – Computes the joint accelerations, specified by the base orientation, the positions and the velocities:

jointAccelerationsHPCEF(*hand_conf*, *tau*, *fe_h*, *ac_h*, *wf_R_b*, *wf_p_b*, *q_j*, *dq_j*, *v_b*, *nu*)

Semi-optimized mode:

jointAccelerationsHPCEF(*hand_conf*, *tau*, *fe_h*, *ac_h*, *wf_R_b*, *wf_p_b*, *q_j*, *nu*)

Optimized mode – Computes the joint accelerations at the current state of the robot system, in dependency of the given whole-body dynamics and velocities:

- **jointAccelerationsHPCEF**(*hand_conf*, *tau*, *fe_h*, *ac_h*, *Jc*, *djcdq*, *M*, *c_qv*[, *dq_j*], *nu*)
- **jointAccelerationsHPCEF**(*hand_conf*, *tau*, *fe_h*, *ac_h*, *nu*)

Parameters

- **hand_conf** (*struct*) – Configuration structure to specify the *qualitative state* of the hands.

  The data structure specifies which hand is currently in contact with the ground, or an object, or a wall. It specifies also the *desired poses*, *angular velocities* and *control gains* for the position-regulation system of the hands.

- **tau** (*double, vector*) – $(n \times 1)$ torque force vector for the joints and the base of the robot with $n = n_{dof} + 6$.
- **fe_h** (*double, vector*) – $(k \times 1)$ vector of external forces (in contact space) that are acting on the specified *contact points* of the hands.

  Note: The *external forces* are either *constant* or *zero*.

- **ac_h** (*double, vector*) – $(k \times 1)$ mixed acceleration vector for the specified *hand contact points*.

  Note: If the hand accelerations are very small, then the vector can also be *constant* or *zero*.

- **wf_R_b** (*double, matrix*) – (3×3) rotation matrix (orientation) from the base frame *b* to world frame *wf* (*optional*).
- **wf_p_b** (*double, vector*) – (3×1) position vector from the base frame *b* to the world frame *wf* (*optional*).
- **q_j** (*double, vector*) – $(n_{dof} \times 1)$ joint positions vector in $[\mathrm{rad}]$ (*optional*).
- **dq_j** (*double, vector*) – $(n_{dof} \times 1)$ joint angle velocity vector in $[\mathrm{rad/s}]$ (*optional*).
- **v_b** (*double, vector*) – (6×1) generalized base velocity vector (*optional*).
- **nu** (*double, vector*) – $((6 + n_{dof}) \times 1)$ mixed generalized velocity vector (generalized base velocity and joint velocity).

The variable k indicates the *size* of the given *force* and *acceleration vectors* in dependency of the specified hands:

- $k = 6$ – only one hand is defined.
- $k = 12$ – both hands are defined.

If the joint velocity vector $\dot{q}_j$ (dq_j) is not given as an argument, then the method assumes that the robot system is *frictionless*.

Other Parameters

- **Jc** (*double, matrix*) – $(6m \times n)$ Jacobian of the *contact constraints* in contact space $\mathrm{C} = \{\mathrm{C}_1, \ldots, \mathrm{C}_\mathrm{k}\}$ (*optional*).
- **djcdq** (*double, vector*) – $(6m \times 1)$ product vector of the time derivative of the contact Jacobian $\dot{J}_c$ and the joint velocity $\dot{q}_j$ (*optional*).
- **M** (*double, matrix*) – $(n \times n)$ generalized mass matrix of the robot (*optional*).
- **c_qv** (*double, vector*) – $(n \times 1)$ generalized bias force vector of the robot (*optional*).

The variable m denotes the *number of contact constraints* of the robot and $n = n_{dof} + 6$.

Returns

[ddq_j[, fd_prms]] –

2-element tuple containing:

- **ddq_j** (*double, vector*) – $(n_{dof} \times 1)$ joint angle acceleration vector in $[\mathrm{rad/s}^2]$.

- **fd_prms** (*struct*) – Data structure for the parameter values of the forward dynamics calculation with the fields `tau_gen`, `f_c`, `a_c` and `f_e` (*optional*).

  Note: The second output argument can be used for further calculations or for data logging.

See also:

WBM.jointAccelerationsCLPCEF().

jointAccelerationsFHPCEF (*foot_conf*, *hand_conf*, *tau*, *fe_h*, *ac_h*, *varargin*)

Calculates the *joint angle accelerations* $\ddot{q}_j$ with *foot* and *hand pose corrections* and additionally *external forces* (*FHPCEF*) that are acting on the hands of the humanoid robot – *experimental*.

The joint accelerations $\ddot{q}_j$ will be calculated as defined in equation (2.8). For a *closed-loop control system* with *velocity* and *position regulation*, the calculated joint acceleration vector depends on the given *contact constraints* and the additional *external forces* that are acting on the hands of the floating-base robot.

The method can be called in two different modes:

Normal mode – Computes the joint accelerations, specified by the base orientation, the positions and the velocities:

jointAccelerationsFHPCEF (*foot_conf*, *hand_conf*, *tau*, *fe_h*, *ac_h*[, *ac_f*], *wf_R_b*, *wf_p_b*, *q_j*, *dq_j*, *v_b*, *nu*)

Optimized mode – Computes the joint accelerations at the current state of the robot system, in dependency of the given whole-body dynamics and velocities:

- **jointAccelerationsFHPCEF** (*foot_conf*, *hand_conf*, *tau*, *fe_h*, *ac_h*[, *ac_f*], *Jc_f*, *djcdq_f*, *M*, *c_qv*, *dq_j*, *nu*)
- **jointAccelerationsFHPCEF** (*foot_conf*, *hand_conf*, *tau*, *fe_h*, *ac_h*[, *ac_f*], *dq_j*, *nu*)

Note: This method is still untested and should be considered as *experimental*.

Parameters

- **foot_conf** (*struct*) – Configuration structure to specify the *qualitative state* of the feet.

  The data structure specifies which foot is currently in contact with the ground.

- **hand_conf** (*struct*) – Configuration structure to specify the *qualitative state* of the hands.

  The data structure specifies which hand is currently in contact with the ground, or an object, or a wall.

- **tau** (*double, vector*) – $(n \times 1)$ torque force vector for the joints and the base of the robot with $n = n_{dof} + 6$.
- **fe_h** (*double, vector*) – $(k \times 1)$ vector of external forces (in contact space C_h) that are acting on the specified *contact points* of the hands.
- **ac_h** (*double, vector*) – $(k \times 1)$ mixed acceleration vector for the specified *hand contact points*.

  Note: If the hand accelerations are very small, then the vector can also be *constant* or *zero*.

- **ac_f** (*double, vector*) – $(k \times 1)$ mixed acceleration vector for the *foot contact points* (*optional*).
- **wf_R_b** (*double, matrix*) – (3×3) rotation matrix (orientation) from the base frame *b* to world frame *wf* (*optional*).

- **wf_p_b** (*double, vector*) – (3×1) position vector from the base frame *b* to the world frame *wf* (*optional*).
- **q_j** (*double, vector*) – $(n_{dof} \times 1)$ joint positions vector in $[\mathrm{rad}]$ (*optional*).
- **dq_j** (*double, vector*) – $(n_{dof} \times 1)$ joint angle velocity vector in $[\mathrm{rad/s}]$.
- **v_b** (*double, vector*) – (6×1) generalized base velocity vector (*optional*).
- **nu** (*double, vector*) – $((6+n_{dof}) \times 1)$ mixed generalized velocity vector (generalized base velocity and joint velocity).

The given configuration structures specifying also the *desired poses*, *angular velocities* and *control gains* for the position-regulation system of the feet and hands.

The variable k indicates the *size* of the given *force* and *acceleration vectors* in dependency of the specified feet and hands:

- $k = 6$ – only one foot/hand is defined.
- $k = 12$ – both feet/hands are defined.

The given *external forces* and *foot accelerations* are either *constant* or *zero*.

Other Parameters

- **Jc_f** (*double, matrix*) – $(6m \times n)$ Jacobian of the *foot contact constraints* in contact space C_f (*optional*).
- **djcdq_f** (*double, vector*) – $(6m \times 1)$ product vector of the time derivative of the contact Jacobian $\dot{J}_c^f$ for the feet and the joint velocity $\dot{q}_j$ (*optional*).
- **M** (*double, matrix*) – $(n \times n)$ generalized mass matrix of the robot (*optional*).
- **c_qv** (*double, vector*) – $(n \times 1)$ generalized bias force vector of the robot (*optional*).

The variable m denotes the *number of contact constraints* of the robot and $n = n_{dof} + 6$.

Returns

[ddq_j[, fd_prms]] –

2-element tuple containing:

- **ddq_j** (*double, vector*) – $(n_{dof} \times 1)$ joint angle acceleration vector in $[\mathrm{rad/s^2}]$.
- **fd_prms** (*struct*) – Data structure for the parameter values of the forward dynamics calculation with the fields `tau_gen`, `f_c`, `a_c` and `f_e` (*optional*).

  Note: The second output argument can be used for further calculations or for data logging.

See also:

WBM.jointAccelerationsFHPCPL() and *WBM.contactForcesCLPCEF()*.

jointAccelerationsFHPCPL (*foot_conf, hand_conf, tau, fhTotCWrench, f_cp, varargin*)

Calculates the *joint angle accelerations* $\ddot{q}_j$ with *foot* and *hand pose corrections* and additionally *payload forces* (*FHPCPL*) that are acting on the hands of the humanoid robot – *experimental*.

The joint accelerations $\ddot{q}_j$ will be calculated as defined in equation (2.8). For a *closed-loop control system* with *velocity* and *position regulation*, the calculated joint acceleration vector depends on the given *contact constraints* and the additional *payload forces* that are acting on the hands of the floating-base robot.

The method assumes that *only the feet* of the robot have contact with the *ground* and can be called in two different modes:

Normal mode – Computes the joint accelerations, specified by the base orientation, the positions and the velocities:

jointAccelerationsFHPCPL (*foot_conf*, *hand_conf*, *tau*, *fhTotCWrench*, *f_cp*[, *ac_f*], *wf_R_b*, *wf_p_b*, *q_j*, *dq_j*, *v_b*, *nu*)

Optimized mode – Computes the joint accelerations at the current state of the robot system, in dependency of the given whole-body dynamics and velocities:

- **jointAccelerationsFHPCPL** (*foot_conf*, *hand_conf*, *tau*, *fhTotCWrench*, *f_cp*[, *ac_f*], *Jc_f*, *djcdq_f*, *M*, *c_qv*, *dq_j*, *nu*)
- **jointAccelerationsFHPCPL** (*foot_conf*, *hand_conf*, *tau*, *fhTotCWrench*, *f_cp*[, *ac_f*], *dq_j*, *v_b*, *nu*)

Note: This method is still untested and should be considered as *experimental*.

Parameters

- **foot_conf** (*struct*) – Configuration structure to specify the *qualitative state* of the feet.

  The data structure specifies which foot is currently in contact with the ground.
- **hand_conf** (*struct*) – Configuration structure to specify the *qualitative state* of the hands.

  The data structure specifies which hand is currently in contact with the payload object.
- **tau** (*double, vector*) – $(n \times 1)$ torque force vector for the joints and the base of the robot with $n = n_{dof} + 6$.
- **fhTotCWrench** (*function_handle*) – Function handle to a specific *total contact wrench function* in *contact space* $\mathrm{C_h} = \{\mathrm{C_1}, \ldots, \mathrm{C_n}\}$ of the hands that will be applied by the robot model.
- **f_cp** (*double, vector*) – Force vector or scalar applied to a grasped object at the *contact points* ${}^{O}p_{C_i}$ from the contact frames $\{\mathrm{C_1}, \ldots, \mathrm{C_n}\}$ of the hands to the origin frame O at the CoM of the object.

  The vector length l of the applied forces depends on the chosen *contact model* and if only one hand or both hands are involved in grasping an object, such that $l = h \cdot s$ with size $s \in \{1, 3, 4\}$ and the number of hands $h \in \{1, 2\}$.

  Note: The z-axis of a contact frame $\mathrm{C_{i \in \{1,\ldots,n\}}}$ points in the direction of the inward surface normal at the point of contact ${}^{O}p_{C_i}$. If the chosen contact model is *frictionless*, then each applied force to the object is a scalar, otherwise a vector.
- **ac_f** (*double, vector*) – $(k \times 1)$ mixed acceleration vector for the specified *foot contact points* with the size of $k = 6$ (one foot) or $k = 12$ (both feet) – *optional*.

  Note: The given *foot accelerations* are either *constant* or *zero*.
- **wf_R_b** (*double, matrix*) – (3×3) rotation matrix (orientation) from the base frame *b* to world frame *wf* (*optional*).
- **wf_p_b** (*double, vector*) – (3×1) position vector from the base frame *b* to the world frame *wf* (*optional*).
- **q_j** (*double, vector*) – $(n_{dof} \times 1)$ joint positions vector in $[\mathrm{rad}]$ (*optional*).
- **dq_j** (*double, vector*) – $(n_{dof} \times 1)$ joint angle velocity vector in $[\mathrm{rad/s}]$.
- **v_b** (*double, vector*) – (6×1) generalized base velocity vector (*optional*).
- **nu** (*double, vector*) – $((6 + n_{dof}) \times 1)$ mixed generalized velocity vector (generalized base velocity and joint velocity).

The given configuration structures specifying also the *desired poses*, *angular velocities* and *control gains* for the position-regulation system of the feet and hands.

Other Parameters

- **Jc_f** (*double, matrix*) – $(6m \times n)$ Jacobian of the *foot contact constraints* in contact space C_f (*optional*).
- **djcdq_f** (*double, vector*) – $(6m \times 1)$ product vector of the time derivative of the contact Jacobian $\dot{J}_c^f$ for the feet and the joint velocity $\dot{q}_j$ (*optional*).
- **M** (*double, matrix*) – $(n \times n)$ generalized mass matrix of the robot (*optional*).
- **c_qv** (*double, vector*) – $(n \times 1)$ generalized bias force vector of the robot (*optional*).

The variable m denotes the *number of contact constraints* of the robot and $n = n_{dof} + 6$.

Returns

[ddq_j[, fd_prms]] –

2-element tuple containing:

- **ddq_j** (*double, vector*) – $(n_{dof} \times 1)$ joint angle acceleration vector in $[\mathrm{rad/s}^2]$.
- **fd_prms** (*struct*) – Data structure for the parameter values of the forward dynamics calculation with the fields `tau_gen`, `f_c`, `a_c` and `f_pl` (*optional*).

  Note: The second output argument can be used for further calculations or for data logging.

See also:

WBM.jointAccelerationsFHPCEF(), *WBM.contactForcesCLPCEF()*, *WBM.handAccelerations()* and `WBM.handPayloadForces()`.

handAccelerations (*foot_conf*, *hand_conf*, *tau*, *varargin*)

Calculates the *mixed hand accelerations* (end-effector accelerations) a_{c_h} of a humanoid robot in contact (operational) space C_h.

Using equation (2.8) for the joint accelerations $\ddot{q}_j$, the formula to calculate the *mixed accelerations* a_{c_h} at the specified *contact links* of the hands (end-effectors) can be expressed as follows:

$$a_{c_h} = \ddot{x}_h = J_{c_h} M^{\text{-}1} \cdot (\tau_{gen} - C(q_j, \dot{q}_j) - (J_{c_f}^T \cdot \text{-}f_{c_f})) + \dot{J}_{c_h} \dot{q}_j \;, \qquad (2.10)$$

where the matrix J_{c_h} denotes the contact Jacobian of the hands and $J_{c_f}^T \cdot \text{-}f_{c_f}$ represents the contact forces of the feet in joint space.

The method can be called in two different modes:

Normal mode – Computes the mixed accelerations, specified by the base orientation, the positions and the velocities:

handAccelerations (*foot_conf*, *hand_conf*, *tau*, *ac_f*, *wf_R_b*, *wf_p_b*, *q_j*, *dq_j*, *v_b*[, *nu*])

Optimized mode – Computes the mixed accelerations at the current state of the robot system, in dependency of the given whole-body dynamics and velocities:

- **handAccelerations** (*foot_conf*, *hand_conf*, *tau*, *ac_f*, *Jc_f*, *djcdq_f*, *M*, *c_qv*, *dq_j*, *nu*)
- **handAccelerations** (*foot_conf*, *hand_conf*, *tau*, *ac_f*, *dq_j*[, *nu*])

Parameters

- **foot_conf** (*struct*) – Configuration structure to specify the *qualitative state* of the feet.

  The data structure specifies which foot is currently in contact with the ground.

- **hand_conf** (`struct`) – Configuration structure to specify the *qualitative state* of the hands.

  The data structure specifies which hand is currently in contact with the ground, or an object, or a wall.

- **tau** (`double, vector`) – $(n \times 1)$ torque force vector for the joints and the base of the robot with $n = n_{dof} + 6$.
- **ac_f** (`double, vector`) – $(k \times 1)$ mixed acceleration vector for the specified *foot contact points* with the size of $k = 6$ (one foot) or $k = 12$ (both feet) – *optional*.

  Note: The given *foot accelerations* are either *constant* or *zero*.

- **wf_R_b** (`double, matrix`) – (3×3) rotation matrix (orientation) from the base frame *b* to world frame *wf* (*optional*).
- **wf_p_b** (`double, vector`) – (3×1) position vector from the base frame *b* to the world frame *wf* (*optional*).
- **q_j** (`double, vector`) – $(n_{dof} \times 1)$ joint positions vector in $[\mathrm{rad}]$ (*optional*).
- **dq_j** (`double, vector`) – $(n_{dof} \times 1)$ joint angle velocity vector in $[\mathrm{rad/s}]$.
- **v_b** (`double, vector`) – (6×1) generalized base velocity vector (*optional*).
- **nu** (`double, vector`) – $((6 + n_{dof}) \times 1)$ mixed generalized velocity vector (generalized base velocity and joint velocity).

In dependency of the situation, the given configuration structures can also specify the *desired poses*, *angular velocities* and *control gains* for the position-regulation system of the feet and hands.

Other Parameters

- **Jc_f** (*double, matrix*) – $(6m \times n)$ Jacobian of the *foot contact constraints* in contact space C_f (*optional*).
- **djcdq_f** (*double, vector*) – $(6m \times 1)$ product vector of the time derivative of the contact Jacobian $\dot{J}_c^f$ for the feet and the joint velocity $\dot{q}_j$ (*optional*).
- **M** (*double, matrix*) – $(n \times n)$ generalized mass matrix of the robot (*optional*).
- **c_qv** (*double, vector*) – $(n \times 1)$ generalized bias force vector of the robot (*optional*).

The variable m denotes the *number of contact constraints* of the robot and $n = n_{dof} + 6$.

Returns

[ac_h[, a_prms]] –

2-element tuple containing:

- **ac_h** (*double, vector*) – $(6m \times 1)$ mixed acceleration vector of the hands in $[\mathrm{rad/s}^2]$ with either $m = 1$ or $m = 2$.
- **a_prms** (*struct*) – Data structure of the calculated hand accelerations parameters with the fields:

  1. `M`, `c_qv`, `Jc_f`, `Jc_h`, `djcdq_f`, `djcdq_h`, `fc_f`, `tau_gen` or
  2. `Jc_h`, `djcdq_h`, `fc_f`, `tau_gen`

  Note: The second output argument can be used for further calculations or for data logging.

See also:

`WBM.jointAccelerations()` and *`WBM.handVelocities()`*.

References

[Lil92], p. 82, eq. (5.9).

handVelocities (*hand_conf*, *varargin*)

Calculates the *mixed hand velocities* (end-effector velocities) v_{c_h} of a given humanoid robot in contact (operational) space C_h.

The formula to calculate the *mixed velocities* c_{c_h} at the specified *contact links* of the hands (end-effectors) can be expressed as follows:

$$v_{c_h} = \dot{x}_h = J_{c_h} \cdot \dot{q}_j \ , \tag{2.11}$$

where the matrix J_{c_h} denotes the contact Jacobian of the hands.

The method can be called in two different modes:

Normal mode – Computes the hand accelerations, specified by the base orientation, the positions and the velocities:

handVelocities (*hand_conf*, *wf_R_b*, *wf_p_b*, *q_j*, *dq_j*, *v_b*)

Optimized mode – Computes the hand accelerations at the current state of the robot system, in dependency of the given contact Jacobian and the joint velocities:

handVelocities (*hand_conf*[, *Jc_h*], *dq_j*)

Parameters

- **hand_conf** (*struct*) – Configuration structure to specify the *qualitative state* of the hands.

  The data structure specifies which hand is currently in contact with the ground, or an object, or a wall.

  Note: In dependency of the situation, the configuration structure can also specify the *desired poses*, *angular velocities* and *control gains* for the position-regulation system of the hands.

- **wf_R_b** (*double, matrix*) – (3×3) rotation matrix (orientation) from the base frame *b* to world frame *wf* (*optional*).
- **wf_p_b** (*double, vector*) – (3×1) position vector from the base frame *b* to the world frame *wf* (*optional*).
- **q_j** (*double, vector*) – $(n_{dof} \times 1)$ joint positions vector in $[\mathrm{rad}]$ (*optional*).
- **dq_j** (*double, vector*) – $(n_{dof} \times 1)$ joint angle velocity vector in $[\mathrm{rad/s}]$.
- **v_b** (*double, vector*) – (6×1) generalized base velocity vector (*optional*).

Other Parameters **Jc_h** (*double, matrix*) – $(6m \times n)$ Jacobian of the *hand contact constraints* in contact space C_h with either $m = 1$ or $m = 2$ and $n = n_{dof} + 6$ (*optional*).

Returns

[vc_h[, v_prms]] –

2-element tuple containing:

- **vc_h** (*double, vector*) – $(6m \times 1)$ mixed velocity vector of the hands in $[\mathrm{rad/s}^2]$ with either $m = 1$ or $m = 2$.
- **v_prms** (*struct*) – Data structure of the calculated hand velocities parameters with the fields `Jc_h` and `djcdq_h`.

Note: The second output argument can be used for further calculations or for data logging.

See also:

WBM.jointAccelerations() and *WBM.handVelocities()*.

References

[Lil92], p. 82, eq. (5.7).

forwardDynamics (*t*, *stvChi*, *fhTrqControl*)

Computes the *forward dynamics* of the *whole body model* of the given floating-base robot.

The forward dynamics describes the calculation of the acceleration response $\ddot{q}$ of a rigid-body system to a given torque force τ (*[Fea08]*, p. 2). The dynamic model can be expressed as an *ordinary differential equation* (ODE) of the form:

$$\dot{\chi} = \mathrm{FD}(model, t, \chi, \tau)\ , \tag{2.12}$$

where FD denotes the function for the forward dynamics calculations in dependency of the given time step $t \in [t_0, t_f]$ of a given time interval and the vector variable χ to be integrated (compare also with *[Fea08]*, p. 41).

The method will be passed to a Matlab ODE solver *[SR97]* or other solvers for stiff differential equations and differential-algebraic systems (differential equations with constraints).

Parameters

- **t** (*double, scalar*) – Current time step of the given time interval of integration $[t_0, t_f]$ with $n > 1$ step elements in ascending order.
- **stvChi** (*double, vector*) – $((2n_{dof} + 13) \times 1)$ state vector χ of the robot model at the current time step t that will be integrated by ODE solver.
- **fhTrqControl** (*function_handle*) – Function handle to a specified time-dependent *torque control function* that controls the dynamics of the robot system.

Returns *dstvChi (double, vector)* – $((2n_{dof} + 13) \times 1)$ time-derivative vector (next state) of the state function $\chi(t)$ at the current time step t satisfying the above equation (2.12).

See also:

WBM.forwardDynamicsEF() and *WBM.forwardDynamicsPL()*.

References

- *[Fea08]*, p. 2 and p. 41, eq. (3.3).
- *[SR97]*

forwardDynamicsEF (*t*, *stvChi*, *fhTrqControl*, *foot_conf*, *clnk_conf*, *fe_c*, *ac*, *ac_f*)

Computes the *forward dynamics* of the *whole body model* of the given floating-base robot with *external forces* (*EF*) that are acting on the contact links of the robot – *experimental*.

The forward dynamics describes the calculation of the acceleration response $\ddot{q}$ of a rigid-body system to a given torque force τ (*[Fea08]*, p. 2). The dynamic model can be expressed as an *ordinary differential equation* (ODE) of the form:

$$\dot{\chi} = \mathrm{FD}(model, t, \chi, \tau, f_e^C)\ , \tag{2.13}$$

where FD denotes the function for the forward dynamics calculations in dependency of the given time step $t \in [t_0, t_f]$ of a given time interval, f_e^C are the external forces in contact space C and the vector variable χ to be integrated (compare also with *[Fea08]*, p. 41).

The method will be passed to a Matlab ODE solver *[SR97]* or other solvers for stiff differential equations and differential-algebraic systems (differential equations with constraints).

Note: This forward dynamics method is still untested and should be considered as *experimental*.

Parameters

- **`t`** (*`double, scalar`*) – Current time step of the given time interval of integration $[t_0, t_f]$ with $n > 1$ step elements in ascending order.
- **`stvChi`** (*`double, vector`*) – $((2n_{dof} + 13) \times 1)$ state vector χ of the robot model at the current time step t that will be integrated by ODE solver.
- **`fhTrqControl`** (*`function_handle`*) – Function handle to a specified time-dependent *torque control function* that controls the dynamics of the robot system.
- **`foot_conf`** (*`struct`*) – Configuration structure to specify the *qualitative state* of the feet.

  The data structure specifies which foot is currently in contact with the ground.
- **`clnk_conf`** (*`struct`*) – Configuration structure to specify the *qualitative state* of at most two *contact links*.

  The data structure specifies which link is currently in contact with the ground or an object.
- **`fe_c`** (*`double, vector`*) – $(k \times 1)$ vector of external forces (in contact space C) that are acting on the specified contact links.
- **`ac`** (*`double, vector`*) – $(k \times 1)$ mixed acceleration vector for the *contact points* of the contact links.
- **`ac_f`** (*`double, vector`*) – $(k \times 1)$ mixed acceleration vector for the *foot contact points*.

The variable k indicates the *size* of the given *force* and *acceleration vectors* in dependency of the specified contact links and feet:

- $k = 6$ – only one link/foot is defined.
- $k = 12$ – both links/feet are defined.

The given *external forces* and *accelerations* are either *constant* or *zero*.

Returns *dstvChi (double, vector)* – $((2n_{dof} + 13) \times 1)$ time-derivative vector (next state) of the state function $\chi(t)$ at the current time step t satisfying the above equation (2.13).

See also:

`WBM.forwardDynamics()` and *`WBM.forwardDynamicsPL()`*.

References

- *[Fea08]*, p. 2 and p. 41, eq. (3.3).
- *[SR97]*

forwardDynamicsPL (*t*, *stvChi*, *fhTrqControl*, *fhTotCWrench*, *foot_conf*, *hand_conf*, *f_cp*, *ac_f*)
Computes the *forward dynamics* of the *whole body model* of a humanoid robot with *payload forces* (*PL*) that are acting on the hands of the robot.

The forward dynamics describes the calculation of the acceleration response $\ddot{q}$ of a rigid-body system to a given torque force τ (*[Fea08]*, p. 2). The dynamic model can be expressed as an *ordinary differential equation* (ODE) of the form:

$$\dot{\chi} = \mathrm{FD}(model, t, \chi, \tau, f_{cp}^{C_h})\,, \tag{2.14}$$

where FD denotes the function for the forward dynamics calculations in dependency of the given time step $t \in [t_0, t_f]$ of a given time interval, $f_{cp}^{C_h}$ are the applied forces to a grasped object at the *contact points* in contact space C_h of the hands and the vector variable χ to be integrated (compare also with *[Fea08]*, p. 41).

The method will be passed to a Matlab ODE solver *[SR97]* or other solvers for stiff differential equations and differential-algebraic systems (differential equations with constraints).

Parameters

- **t** (*double, scalar*) – Current time step of the given time interval of integration $[t_0, t_f]$ with $n > 1$ step elements in ascending order.
- **stvChi** (*double, vector*) – $((2n_{dof}+13)\times 1)$ state vector χ of the robot model at the current time step t that will be integrated by ODE solver.
- **fhTrqControl** (*function_handle*) – Function handle to a specified time-dependent *torque control function* that controls the dynamics of the robot system.
- **fhTotCWrench** (*function_handle*) – Function handle to a specific *total contact wrench function* in *contact space* $\mathrm{C_h} = \{\mathrm{C_1}, \ldots, \mathrm{C_n}\}$ of the hands that will be applied by the robot model.
- **foot_conf** (*struct*) – Configuration structure to specify the *qualitative state* of the feet.

  The data structure specifies which foot is currently in contact with the ground.
- **hand_conf** (*struct*) – Configuration structure to specify the *qualitative state* of the hands.

  The data structure specifies which hand is currently in contact with the payload object.
- **f_cp** (*double, vector*) – Force vector or scalar applied to a grasped object at the *contact points* ${}^{O}p_{C_i}$ from the contact frames $\{\mathrm{C_1}, \ldots, \mathrm{C_n}\}$ of the hands to the origin frame O at the CoM of the object.

  The vector length l of the applied forces depends on the chosen *contact model* and if only one hand or both hands are involved in grasping an object, such that $l = h \cdot s$ with size $s \in \{1, 3, 4\}$ and the number of hands $h \in \{1, 2\}$.

  Note: The z-axis of a contact frame $\mathrm{C_{i\in\{1,\ldots,n\}}}$ points in the direction of the inward surface normal at the point of contact ${}^{O}p_{C_i}$. If the chosen contact model is *frictionless*, then each applied force to the object is a scalar, otherwise a vector.
- **ac_f** (*double, vector*) – $(k\times 1)$ mixed acceleration vector for the specified *foot contact points* with the size of $k = 6$ (one foot) or $k = 12$ (both feet).

  Note: The given *foot accelerations* are either *constant* or *zero*.

Returns *dstvChi (double, vector)* – $((2n_{dof}+13)\times 1)$ time-derivative vector (next state) of the state function $\chi(t)$ at the current time step t satisfying the above equation (2.14).

See also:

WBM.forwardDynamics() and *WBM.forwardDynamicsEF()*.

References

- *[Fea08]*, p. 2 and p. 41, eq. (3.3).
- *[SR97]*

forwardDynamicsNFB (*t*, *stvChi*, *fhTrqControl*)

Computes the *forward dynamics* of the *whole body model* of a humanoid robot without a *floating base* (*NFB*) or by exclusion of it.

Since the given robot is defined without a floating base, there are no contact forces f_c to the ground, i.e. $f_c = 0$. The behavior of the model is the same as a robot with a fixed base.

The forward dynamics describes the calculation of the acceleration response $\ddot{q}$ of a rigid-body system to a given torque force τ (*[Fea08]*, p. 2). The dynamic model will computed with an *ordinary differential equation* (ODE) as defined in (2.12).

The method will be passed to a Matlab ODE solver *[SR97]* or other solvers for stiff differential equations and differential-algebraic systems (differential equations with constraints).

Parameters

- **t** (*double, scalar*) – Current time step of the given time interval of integration $[t_0, t_f]$ with $n > 1$ step elements in ascending order.
- **stvChi** (*double, vector*) – $((2n_{dof}+13)\times 1)$ state vector χ of the robot model at the current time step t that will be integrated by ODE solver.
- **fhTrqControl** (*function_handle*) – Function handle to a specified time-dependent *torque control function* that controls the dynamics of the robot system.

Returns *dstvChi (double, vector)* – $((2n_{dof}+13)\times 1)$ time-derivative vector (next state) of the state function $\chi(t)$ at the current time step t satisfying the equation (2.12).

See also:

WBM.forwardDynamicsNFBEF() and *WBM.forwardDynamicsNFBPL()*.

References

- *[Fea08]*, p. 2 and p. 41, eq. (3.3).
- *[SR97]*

forwardDynamicsNFBEF (*t*, *stvChi*, *fhTrqControl*, *clnk_conf*, *fe_c*, *ac*)

Computes the *forward dynamics* of the *whole body model* of a humanoid robot with *external forces* that are acting on the contact links of the robot, but without a *floating base* (*NFBEF*) or by exclusion of it – *experimental*.

Since the given robot is defined without a floating base, there are no contact forces f_c to the ground, i.e. $f_c = 0$. The behavior of the model is the same as a robot with a fixed base.

The forward dynamics describes the calculation of the acceleration response $\ddot{q}$ of a rigid-body system to a given torque force τ (*[Fea08]*, p. 2). The dynamic model will computed with an *ordinary differential equation* (ODE) as defined in (2.13).

The method will be passed to a Matlab ODE solver *[SR97]* or other solvers for stiff differential equations and differential-algebraic systems (differential equations with constraints).

Note: This forward dynamics method is still untested and should be considered as *experimental*.

Parameters

- **t** (*double, scalar*) – Current time step of the given time interval of integration $[t_0, t_f]$ with $n > 1$ step elements in ascending order.
- **stvChi** (*double, vector*) – $((2n_{dof} + 13) \times 1)$ state vector χ of the robot model at the current time step t that will be integrated by ODE solver.
- **fhTrqControl** (*function_handle*) – Function handle to a specified time-dependent *torque control function* that controls the dynamics of the robot system.
- **clnk_conf** (*struct*) – Configuration structure to specify the *qualitative state* of at most two *contact links*.

  The data structure specifies which link is currently in contact with the ground or an object.
- **fe_c** (*double, vector*) – $(k \times 1)$ vector of external forces (in contact space C) that are acting on the specified contact links.
- **ac** (*double, vector*) – $(k \times 1)$ mixed acceleration vector for the *contact points* of the contact links.

The variable k indicates the *size* of the given *force* and *acceleration vectors* in dependency of the specified contact links:

- $k = 6$ – only one link is defined.
- $k = 12$ – both links are defined.

The given *external forces* and *accelerations* are either *constant* or *zero*.

Returns *dstvChi (double, vector)* – $((2n_{dof} + 13) \times 1)$ time-derivative vector (next state) of the state function $\chi(t)$ at the current time step t satisfying the equation (2.13).

See also:

WBM.forwardDynamicsNFB() and *WBM.forwardDynamicsNFBPL()*.

References

- *[Fea08]*, p. 2 and p. 41, eq. (3.3).
- *[SR97]*

forwardDynamicsNFBPL (*t, stvChi, fhTrqControl, fhTotCWrench, hand_conf, f_cp*)

Computes the *forward dynamics* of the *whole body model* of a humanoid robot with *payload forces* that are acting on the hands of the robot, but without a *floating base* (*NFBPL*) or by exclusion of it.

Since the given robot is defined without a floating base, there are no contact forces f_c to the ground, i.e. $f_c = 0$. The behavior of the model is the same as a robot with a fixed base.

The forward dynamics describes the calculation of the acceleration response $\ddot{q}$ of a rigid-body system to a given torque force τ (*[Fea08]*, p. 2). The dynamic model will computed with an *ordinary differential equation* (ODE) as defined in (2.14).

The method will be passed to a Matlab ODE solver *[SR97]* or other solvers for stiff differential equations and differential-algebraic systems (differential equations with constraints).

Parameters

- **t** (*double, scalar*) – Current time step of the given time interval of integration $[t_0, t_f]$ with $n > 1$ step elements in ascending order.
- **stvChi** (*double, vector*) – $((2n_{dof} + 13) \times 1)$ state vector χ of the robot model at the current time step t that will be integrated by ODE solver.
- **fhTrqControl** (*function_handle*) – Function handle to a specified time-dependent *torque control function* that controls the dynamics of the robot system.

- **fhTotCWrench** (*function_handle*) – Function handle to a specific *total contact wrench function* in *contact space* $\mathrm{C_h} = \{\mathrm{C_1}, \ldots, \mathrm{C_n}\}$ of the hands that will be applied by the robot model.
- **hand_conf** (*struct*) – Configuration structure to specify the *qualitative state* of the hands.

  The data structure specifies which hand is currently in contact with the payload object.
- **f_cp** (*double, vector*) – Force vector or scalar applied to a grasped object at the *contact points* ${}^{O}p_{C_i}$ from the contact frames $\{\mathrm{C_1}, \ldots, \mathrm{C_n}\}$ of the hands to the origin frame O at the CoM of the object.

  The vector length l of the applied forces depends on the chosen *contact model* and if only one hand or both hands are involved in grasping an object, such that $l = h \cdot s$ with size $s \in \{1, 3, 4\}$ and the number of hands $h \in \{1, 2\}$.

  Note: The z-axis of a contact frame $\mathrm{C_{i\in\{1,\ldots,n\}}}$ points in the direction of the inward surface normal at the point of contact ${}^{O}p_{C_i}$. If the chosen contact model is *frictionless*, then each applied force to the object is a scalar, otherwise a vector.

Returns *dstvChi (double, vector)* – $((2n_{dof} + 13) \times 1)$ time-derivative vector (next state) of the state function $\chi(t)$ at the current time step t satisfying the equation (2.14).

See also:

WBM.forwardDynamicsNFB() and *WBM.forwardDynamicsNFBEF()*.

References

- *[Fea08]*, p. 2 and p. 41, eq. (3.3).
- *[SR97]*

forwardDynamicsFPC (*t*, *stvChi*, *fhTrqControl*, *foot_conf*, *ac_f*)

Computes the *forward dynamics* of the *whole body model* of the given floating-base robot with *foot pose corrections* (*FPC*).

The forward dynamics describes the calculation of the acceleration response $\ddot{q}$ of a rigid-body system to a given torque force τ (*[Fea08]*, p. 2). The dynamic model will computed with an *ordinary differential equation* (ODE) as defined in (2.12).

The method will be passed to a Matlab ODE solver *[SR97]* or other solvers for stiff differential equations and differential-algebraic systems (differential equations with constraints).

Parameters

- **t** (*double, scalar*) – Current time step of the given time interval of integration $[t_0, t_f]$ with $n > 1$ step elements in ascending order.
- **stvChi** (*double, vector*) – $((2n_{dof}+13)\times 1)$ state vector χ of the robot model at the current time step t that will be integrated by ODE solver.
- **fhTrqControl** (*function_handle*) – Function handle to a specified time-dependent *torque control function* that controls the dynamics of the robot system.
- **foot_conf** (*struct*) – Configuration structure to specify the *qualitative state* of the feet.

  The data structure specifies which foot is currently in contact with the ground. It specifies also the *desired poses*, *angular velocities* and *control gains* for the position-regulation system of the feet.
- **ac_f** (*double, vector*) – $(k \times 1)$ mixed acceleration vector for the specified *foot contact points* with the size of $k = 6$ (one foot) or $k = 12$ (both feet).

  Note: The given *foot accelerations* are either *constant* or *zero*.

Returns *dstvChi (double, vector)* – $((2n_{dof} + 13) \times 1)$ time-derivative vector (next state) of the state function $\chi(t)$ at the current time step t satisfying the equation (2.12).

See also:

WBM.forwardDynamicsFPCEF() and *WBM.forwardDynamicsFPCPL()*.

References

- *[Fea08]*, p. 2 and p. 41, eq. (3.3).
- *[SR97]*

forwardDynamicsFPCEF (*t*, *stvChi*, *fhTrqControl*, *foot_conf*, *clnk_conf*, *fe_c*, *ac*, *ac_f*)

Computes the *forward dynamics* of the *whole body model* of the given floating-base robot with *foot pose corrections* with *external forces* (*FPCEF*) that are acting on the contact links of the robot – *experimental*.

The forward dynamics describes the calculation of the acceleration response $\ddot{q}$ of a rigid-body system to a given torque force τ (*[Fea08]*, p. 2). The dynamic model will computed with an *ordinary differential equation* (ODE) as defined in (2.13).

The method will be passed to a Matlab ODE solver *[SR97]* or other solvers for stiff differential equations and differential-algebraic systems (differential equations with constraints).

Note: This forward dynamics method is still untested and should be considered as *experimental*.

Parameters

- **t** (*double, scalar*) – Current time step of the given time interval of integration $[t_0, t_f]$ with $n > 1$ step elements in ascending order.
- **stvChi** (*double, vector*) – $((2n_{dof}+13) \times 1)$ state vector χ of the robot model at the current time step t that will be integrated by ODE solver.
- **fhTrqControl** (*function_handle*) – Function handle to a specified time-dependent *torque control function* that controls the dynamics of the robot system.
- **foot_conf** (*struct*) – Configuration structure to specify the *qualitative state* of the feet.

  The data structure specifies which foot is currently in contact with the ground. It specifies also the *desired poses*, *angular velocities* and *control gains* for the position-regulation system of the feet.
- **clnk_conf** (*struct*) – Configuration structure to specify the *qualitative state* of at most two *contact links*.

  The data structure specifies which link is currently in contact with the ground or an object.
- **fe_c** (*double, vector*) – $(k \times 1)$ vector of external forces (in contact space C) that are acting on the specified contact links.
- **ac** (*double, vector*) – $(k \times 1)$ mixed acceleration vector for the *contact points* of the contact links.
- **ac_f** (*double, vector*) – $(k \times 1)$ mixed acceleration vector for the *foot contact points*.

The variable k indicates the *size* of the given *force* and *acceleration vectors* in dependency of the specified contact links and feet:

- $k = 6$ – only one link/foot is defined.

- $k = 12$ – both links/feet are defined.

The given *external forces* and *accelerations* are either *constant* or *zero*.

Returns *dstvChi (double, vector)* – $((2n_{dof} + 13) \times 1)$ time-derivative vector (next state) of the state function $\chi(t)$ at the current time step t satisfying the equation (2.13).

See also:

`WBM.forwardDynamicsFPC()`, `WBM.forwardDynamicsFPCPL()` and `WBM.forwardDynamicsHPCEF()`.

References

- *[Fea08]*, p. 2 and p. 41, eq. (3.3).
- *[SR97]*

forwardDynamicsFPCPL (*t*, *stvChi*, *fhTrqControl*, *fhTotCWrench*, *foot_conf*, *hand_conf*, *f_cp*, *ac_f*)

Computes the *forward dynamics* of the *whole body model* of a humanoid robot with *foot pose corrections* and *payload forces* (*FPCPL*) that are acting on the hands of the robot.

The forward dynamics describes the calculation of the acceleration response $\ddot{q}$ of a rigid-body system to a given torque force τ (*[Fea08]*, p. 2). The dynamic model will computed with an *ordinary differential equation* (ODE) as defined in (2.14).

The method will be passed to a Matlab ODE solver *[SR97]* or other solvers for stiff differential equations and differential-algebraic systems (differential equations with constraints).

Parameters

- **t** (*double, scalar*) – Current time step of the given time interval of integration $[t_0, t_f]$ with $n > 1$ step elements in ascending order.
- **stvChi** (*double, vector*) – $((2n_{dof}+13) \times 1)$ state vector χ of the robot model at the current time step t that will be integrated by ODE solver.
- **fhTrqControl** (*function_handle*) – Function handle to a specified time-dependent *torque control function* that controls the dynamics of the robot system.
- **fhTotCWrench** (*function_handle*) – Function handle to a specific *total contact wrench function* in *contact space* $\mathrm{C_h} = \{\mathrm{C_1}, \ldots, \mathrm{C_n}\}$ of the hands that will be applied by the robot model.
- **foot_conf** (*struct*) – Configuration structure to specify the *qualitative state* of the feet.

  The data structure specifies which foot is currently in contact with the ground. It specifies also the *desired poses*, *angular velocities* and *control gains* for the position-regulation system of the feet.
- **hand_conf** (*struct*) – Configuration structure to specify the *qualitative state* of the hands.

  The data structure specifies which hand is currently in contact with the payload object.
- **f_cp** (*double, vector*) – Force vector or scalar applied to a grasped object at the *contact points* ${}^{O}p_{C_i}$ from the contact frames $\{\mathrm{C_1}, \ldots, \mathrm{C_n}\}$ of the hands to the origin frame O at the CoM of the object.

  The vector length l of the applied forces depends on the chosen *contact model* and if only one hand or both hands are involved in grasping an object, such that $l = h \cdot s$ with size $s \in \{1, 3, 4\}$ and the number of hands $h \in \{1, 2\}$.

Note: The z-axis of a contact frame $\mathrm{C}_{\mathrm{i}\in\{1,\ldots,\mathrm{n}\}}$ points in the direction of the inward surface normal at the point of contact ${}^{O}p_{C_i}$. If the chosen contact model is *frictionless*, then each applied force to the object is a scalar, otherwise a vector.

- **ac_f** (*double, vector*) – $(k \times 1)$ mixed acceleration vector for the specified *foot contact points* with the size of $k = 6$ (one foot) or $k = 12$ (both feet).

  Note: The given *foot accelerations* are either *constant* or *zero*.

Returns *dstvChi (double, vector)* – $((2n_{dof} + 13) \times 1)$ time-derivative vector (next state) of the state function $\chi(t)$ at the current time step t satisfying the above equation (2.14).

See also:

WBM.forwardDynamicsFPC() and *WBM.forwardDynamicsFPCEF()*.

References

- *[Fea08]*, p. 2 and p. 41, eq. (3.3).
- *[SR97]*

forwardDynamicsHPCEF (*t*, *stvChi*, *fhTrqControl*, *hand_conf*, *fe_h*, *ac_h*)

Computes the *forward dynamics* of the *whole body model* of the given floating-base robot with *hand pose corrections* and *external forces* (*HPCEF*) that are acting on the hands of the robot – *experimental*.

The forward dynamics describes the calculation of the acceleration response $\ddot{q}$ of a rigid-body system to a given torque force τ (*[Fea08]*, p. 2). The calculation can be expressed as an *ordinary differential equation* (ODE) of the form:

$$\dot{\chi} = \mathrm{FD}(model, t, \chi, \tau, f_e^{C_h})\ , \qquad (2.15)$$

where FD denotes the function for the forward dynamics calculations in dependency of the given time step $t \in [t_0, t_f]$ of a given time interval, $f_e^{C_h} aretheexternal forcesincontactspace : math$:mathrm{C}_h‘ of the hands and the vector variable χ to be integrated (compare also with *[Fea08]*, p. 41).

The method will be passed to a Matlab ODE solver *[SR97]* or other solvers for stiff differential equations and differential-algebraic systems (differential equations with constraints).

Note: This forward dynamics method is still untested and should be considered as *experimental*.

Parameters

- **t** (*double, scalar*) – Current time step of the given time interval of integration $[t_0, t_f]$ with $n > 1$ step elements in ascending order.
- **stvChi** (*double, vector*) – $((2n_{dof}+13)\times 1)$ state vector χ of the robot model at the current time step t that will be integrated by ODE solver.
- **fhTrqControl** (*function_handle*) – Function handle to a specified time-dependent *torque control function* that controls the dynamics of the robot system.
- **hand_conf** (*struct*) – Configuration structure to specify the *qualitative state* of the hands.

  The data structure specifies which hand is currently in contact with the ground, or an object, or a wall. It specifies also the *desired poses*, *angular velocities* and *control gains* for the position-regulation system of the hands.

- **fe_h** (*double, vector*) – $(k \times 1)$ vector of external forces (in contact space C_h) that are acting on the specified *contact points* of the hands.
- **ac_h** (*double, vector*) – $(k \times 1)$ mixed acceleration vector for the *hand contact points*.

The variable k indicates the *size* of the given *force* and *acceleration vectors* in dependency of the specified hands:

- $k = 6$ – only one hand is defined.
- $k = 12$ – both hands are defined.

The given *external forces* and *accelerations* are either *constant* or *zero*.

Returns *dstvChi (double, vector)* – $((2n_{dof} + 13) \times 1)$ time-derivative vector (next state) of the state function $\chi(t)$ at the current time step t satisfying the above equation (2.15).

See also:

WBM.forwardDynamicsFPCEF().

References

- *[Fea08]*, p. 2 and p. 41, eq. (3.3).
- *[SR97]*

forwardDynamicsFHPCEF (*t, stvChi, fhTrqControl, foot_conf, hand_conf, fe_h, ac_h, ac_f*)

Computes the *forward dynamics* of the *whole body model* of the given floating-base robot with *foot* and *hand pose corrections* and *external forces* (*FHPCEF*) that are acting on the hands of the robot – *experimental*.

The forward dynamics describes the calculation of the acceleration response $\ddot{q}$ of a rigid-body system to a given torque force τ (*[Fea08]*, p. 2). The dynamic model will computed with an *ordinary differential equation* (ODE) as defined in (2.15).

The method will be passed to a Matlab ODE solver *[SR97]* or other solvers for stiff differential equations and differential-algebraic systems (differential equations with constraints).

Note: This forward dynamics method is still untested and should be considered as *experimental*.

Parameters

- **t** (*double, scalar*) – Current time step of the given time interval of integration $[t_0, t_f]$ with $n > 1$ step elements in ascending order.
- **stvChi** (*double, vector*) – $((2n_{dof}+13) \times 1)$ state vector χ of the robot model at the current time step t that will be integrated by ODE solver.
- **fhTrqControl** (*function_handle*) – Function handle to a specified time-dependent *torque control function* that controls the dynamics of the robot system.
- **foot_conf** (*struct*) – Configuration structure to specify the *qualitative state* of the feet.

  The data structure specifies which foot is currently in contact with the ground.
- **hand_conf** (*struct*) – Configuration structure to specify the *qualitative state* of the hands.

  The data structure specifies which hand is currently in contact with the ground, or an object, or a wall.

- **fe_h** (*double, vector*) – $(k \times 1)$ vector of external forces (in contact space C_h) that are acting on the specified *contact points* of the hands.
- **ac_h** (*double, vector*) – $(k \times 1)$ mixed acceleration vector for the *hand contact points*.
- **ac_f** (*double, vector*) – $(k \times 1)$ mixed acceleration vector for the *foot contact points*.

The given configuration structures specifying also the *desired poses*, *angular velocities* and *control gains* for the position-regulation system of the feet and hands.

The variable k indicates the *size* of the given *force* and *acceleration vectors* in dependency of the specified feet and hands:

- $k = 6$ – only one foot/hand is defined.
- $k = 12$ – both feet/hands are defined.

The given *external forces* and *accelerations* are either *constant* or *zero*.

Returns *dstvChi (double, vector)* – $((2n_{dof} + 13) \times 1)$ time-derivative vector (next state) of the state function $\chi(t)$ at the current time step t satisfying the equation (2.15).

See also:

WBM.forwardDynamicsFHPCPL().

References

- *[Fea08]*, p. 2 and p. 41, eq. (3.3).
- *[SR97]*

forwardDynamicsFHPCPL (*t*, *stvChi*, *fhTrqControl*, *fhTotCWrench*, *foot_conf*, *hand_conf*, *f_cp*, *ac_f*)

Computes the *forward dynamics* of the *whole body model* of a humanoid robot with *foot* and *hand pose corrections* and *payload forces* (*FHPCPL*) that are acting on the hands of the robot – *experimental*.

The forward dynamics describes the calculation of the acceleration response $\ddot{q}$ of a rigid-body system to a given torque force τ (*[Fea08]*, p. 2). The dynamic model will computed with an *ordinary differential equation* (ODE) as defined in (2.14).

The method will be passed to a Matlab ODE solver *[SR97]* or other solvers for stiff differential equations and differential-algebraic systems (differential equations with constraints).

Note: This forward dynamics method is still untested and should be considered as *experimental*.

Parameters

- **t** (*double, scalar*) – Current time step of the given time interval of integration $[t_0, t_f]$ with $n > 1$ step elements in ascending order.
- **stvChi** (*double, vector*) – $((2n_{dof}+13) \times 1)$ state vector χ of the robot model at the current time step t that will be integrated by ODE solver.
- **fhTrqControl** (*function_handle*) – Function handle to a specified time-dependent *torque control function* that controls the dynamics of the robot system.
- **fhTotCWrench** (*function_handle*) – Function handle to a specific *total contact wrench function* in *contact space* $\mathrm{C_h} = \{\mathrm{C_1}, \ldots, \mathrm{C_n}\}$ of the hands that will be applied by the robot model.

- **foot_conf** (*struct*) – Configuration structure to specify the *qualitative state* of the feet.

  The data structure specifies which foot is currently in contact with the ground.
- **hand_conf** (*struct*) – Configuration structure to specify the *qualitative state* of the hands.

  The data structure specifies which hand is currently in contact with the payload object.
- **f_cp** (*double, vector*) – Force vector or scalar applied to a grasped object at the *contact points* ${}^{O}p_{C_i}$ from the contact frames $\{\mathrm{C}_1, \ldots, \mathrm{C_n}\}$ of the hands to the origin frame O at the CoM of the object.

  The vector length l of the applied forces depends on the chosen *contact model* and if only one hand or both hands are involved in grasping an object, such that $l = h \cdot s$ with size $s \in \{1, 3, 4\}$ and the number of hands $h \in \{1, 2\}$.

  Note: The z-axis of a contact frame $\mathrm{C}_{\mathrm{i}\in\{1,\ldots,\mathrm{n}\}}$ points in the direction of the inward surface normal at the point of contact ${}^{O}p_{C_i}$. If the chosen contact model is *frictionless*, then each applied force to the object is a scalar, otherwise a vector.
- **ac_h** (*double, vector*) – $(k \times 1)$ mixed acceleration vector for the specified *hand contact points*.
- **ac_f** (*double, vector*) – $(k \times 1)$ mixed acceleration vector for the specified *foot contact points*.

The given configuration structures specifying also the *desired poses*, *angular velocities* and *control gains* for the position-regulation system of the feet and hands.

The variable k indicates the *size* of the given *acceleration vectors* in dependency of the specified feet and hands:

- $k = 6$ – only one foot/hand is defined.
- $k = 12$ – both feet/hands are defined.

The given *acceleration vectors* are either *constant* or *zero*.

Returns *dstvChi (double, vector)* – $((2n_{dof} + 13) \times 1)$ time-derivative vector (next state) of the state function $\chi(t)$ at the current time step t satisfying the equation (2.14).

See also:

WBM.forwardDynamicsFHPCEF().

References

- *[Fea08]*, p. 2 and p. 41, eq. (3.3).
- *[SR97]*

intForwardDynamics (*tspan*, *stvChi_0*, *fhTrqControl*, *ode_opt*, *varargin*)

Setups and integrates the *forward dynamics differential equation* of the form $\dot{\chi} = FD(t, \chi)$ in dependency of the given interval of integration $[t_0, t_f]$ and the initial condition χ_0.

The method uses for the specified ODE-function the Matlab ODE solver `ode15s` for solving stiff differential equations and differential-algebraic systems *[SR97]*.

Following *pose correction types* for the position-regulation system of the forward dynamics will be supported:

- `none` – No pose corrections.
- `nfb` – No floating base and without pose corrections.
- `fpc` – Foot pose correction.

- `hpc` – Hand pose correction.
- `fhpc` – Foot and hand pose correction.

In dependency of the specified *pose correction*, the method can be called as follows:

none, nfb:

`intForwardDynamics` (*tspan*, *stvChi_0*, *fhTrqControl*, *ode_opt*[, *pc_type*])

none, fpc:

`intForwardDynamics` (*tspan*, *stvChi_0*, *fhTrqControl*, *ode_opt*, *foot_conf*, *clnk_conf*, *fe_c*, *ac*, *ac_f*[, *pc_type*])

none, fpc, fhpc:

`intForwardDynamics` (*tspan*, *stvChi_0*, *fhTrqControl*, *ode_opt*, *fhTotCWrench*, *foot_conf*, *hand_conf*, *f_cp*, *ac_f*[, *pc_type*])

nfb:

`intForwardDynamics` (*tspan*, *stvChi_0*, *fhTrqControl*, *ode_opt*, *clnk_conf*, *fe_c*, *ac*[, *pc_type*])

nfb:

`intForwardDynamics` (*tspan*, *stvChi_0*, *fhTrqControl*, *ode_opt*, *fhTotCWrench*, *hand_conf*, *f_cp*[, *pc_type*])

fpc:

`intForwardDynamics` (*tspan*, *stvChi_0*, *fhTrqControl*, *ode_opt*, *foot_conf*, *ac_f*[, *pc_type*])

hpc:

`intForwardDynamics` (*tspan*, *stvChi_0*, *fhTrqControl*, *ode_opt*, *hand_conf*, *fe_h*, *ac_h*[, *pc_type*])

fhpc:

`intForwardDynamics` (*tspan*, *stvChi_0*, *fhTrqControl*, *ode_opt*, *foot_conf*, *hand_conf*, *fe_h*, *ac_h*, *ac_f*[, *pc_type*])

Parameters

- **tspan** (*`double, vector`*) – :math:(1 times n) time interval of integration vector with a minimum size of two elements $[t_0, t_f]$, specifying the initial and final times.

  The variable n denotes the *number of evaluation points* of the given interval vector specified by a time step size t_{step}, such that $t_f = t_0 + n \cdot t_{step}$.

  Note: All elements of the interval vector `tspan` must be *increasing* or *decreasing*. If `tspan` has two elements, then the ODE-solver returns a solution at each internal time step within the interval. If `tspan` has more than two elements, then the solver returns a solution at each given time point.

- **stvChi_0** (*`double, vector`*) – $((2n_{dof} + 13) \times 1)$ initial state vector χ_0 of the robot model to specify the initial condition for the given forward dynamics ODE-function.

- **fhTrqControl** (*`function_handle`*) – Function handle to a specified time-dependent *torque control function* that controls the dynamics of the robot system.

- **ode_opt** (*`struct`*) – Option structure for fine-tuning, adjusting error tolerances, or passing additional information to the ODE-solver.

  Use the `odeset` function of Matlab to create or modify the options structure.

Other Parameters

- **fhTotCWrench** (*function_handle*) – Function handle to a specific *total contact wrench function* in contact space $\mathrm{C_h} = \{\mathrm{C_1}, \dots, \mathrm{C_n}\}$ of the hands that will be applied by the robot model (*optional*).
- **foot_conf** (*struct*) – Configuration structure to specify the *qualitative state* of the feet (*optional*).
- **hand_conf** (*struct*) – Configuration structure to specify the *qualitative state* of the hands (*optional*).
- **clnk_conf** (*struct*) – Configuration structure to specify the *qualitative state* of at most two *contact links* (*optional*).
- **fe_c** (*double, vector*) – $(k \times 1)$ vector of external forces (in contact space) that are acting on the specified contact links (*optional*).
- **fe_h** (*double, vector*) – $(k \times 1)$ vector of external forces (in contact space) that are acting on the specified *contact points* of the hands (*optional*).
- **f_cp** (*double, vector*) – Force vector or scalar applied to a grasped object at the *contact points* ${}^{O}p_{C_i}$ from the contact frames $\{\mathrm{C_1}, \dots, \mathrm{C_n}\}$ of the hands to the origin frame O at the CoM of the object (*optional*).

  The vector length l of the applied forces depends on the chosen *contact model* and if only one hand or both hands are involved in grasping an object, such that $l = h \cdot s$ with size $s \in \{1, 3, 4\}$ and the number of hands $h \in \{1, 2\}$.

  Note: The z-axis of a contact frame $\mathrm{C_{i \in \{1,\dots,n\}}}$ points in the direction of the inward surface normal at the point of contact ${}^{O}p_{C_i}$. If the chosen contact model is *frictionless*, then each applied force to the object is a scalar, otherwise a vector.
- **ac** (*double, vector*) – $(k \times 1)$ mixed acceleration vector for the *contact points* of the specified contact links (*optional*).
- **ac_f** (*double, vector*) – $(k \times 1)$ mixed acceleration vector for the *foot contact points* (*optional*).
- **ac_h** (*double, vector*) – $(k \times 1)$ mixed acceleration vector for the *hand contact points* (*optional*).
- **pc_type** (*char, vector*) – Specifies the *pose correction type* for the robot model that will be applied by the position-regulation system of the given forward dynamics method.

The variable k indicates the *size* of the given *force* and *acceleration vectors* in dependency of the specified contact links, feet and hands:

- $k = 6$ – only one link/foot/hand is defined.
- $k = 12$ – both links/feet/hands are defined.

The specified *external forces* and *accelerations* are either *constant* or *zero*.

Returns

[t, stmChi] –

2-element tuple containing:

- **t** (*double, vector*) – $(n \times 1)$ evaluation points vector of the given time interval to perform the integration.
- **stmChi** (*struct*) – $(n \times (2n_{dof} + 13))$ integration output matrix $\mathcal{X}$ with the next *forward dynamics states* (solutions) of the robot model. Each row in `stmChi` corresponds to a solution at the value in the corresponding row of `t`.

The variable n denotes the *number of evaluation points* of the given time interval.

See also:

WBM.getFDynVisData(), *WBM.getFDynVisDataSect()* and *WBM.visualizeForwardDynamics()*.

References

[SR97]

zeroCtcAcc(*clnk_conf*)

Returns a *zero contact acceleration vector* in dependency of the *contact state* of the given contact links configuration.

Parameters clnk_conf (`struct`) – Configuration structure to specify the *qualitative state* of at most two *contact links*.

The data structure specifies which link is currently in contact with the ground or an object. It specifies also the *desired poses*, *angular velocities* and *control gains* for the position-regulation system of the links.

Returns *ac_0 (double, vector)* – $(k \times 1)$ zero contact acceleration vector with the size of $k = 6$ or $k = 12$.

Note: If both contact links have contact with the ground, object or wall, (or both links have no contact at all) then the size of the vector is 12, otherwise the vector size is 6.

zeroExtForces(*clnk_conf*)

Returns a *zero external force vector* in dependency of the *contact state* of the given contact links configuration.

Parameters clnk_conf (`struct`) – Configuration structure to specify the *qualitative state* of at most two *contact links*.

The data structure specifies which link is currently in contact with the ground or an object. It specifies also the *desired poses*, *angular velocities* and *control gains* for the position-regulation system of the links.

Returns *fe_0 (double, vector)* – $(k \times 1)$ zero external force vector with the size of $k = 6$ or $k = 12$.

Note: If both contact links have contact with the ground, object or wall, (or both links have no contact at all) then the size of the vector is 12, otherwise the vector size is 6.

clnkConfigState(*varargin*)

Creates the *contact links configuration* in dependency of the given *contact state* of the contact links.

The method creates a data structure that specifies which contact link of the robot is currently in “contact” with the environment. If required, the structure specifies also the *desired poses*, *angular velocities* and the corresponding *control gains* for the position-regulation system of the contact links.

Note: The *contact links* are **not** representing the “contact points” where the robot is in contact with the environment. The contact links are those links of the robot that are “behind” the contact points.

The method can be called as follows:

- **clnkConfigState**(*cstate*, *clnk_idx*, *vqT_lnk*, *k_p*, *k_v*[, *rtype*])
- **clnkConfigState**(*cstate*, *clnk_idx*, *veT_lnk*, *k_p*, *k_v*[, *rtype*])
- **clnkConfigState**(*cstate*, *clnk_idx*, *q_j*, *k_p*[, *rtype*])

- **clnkConfigState** (*cstate*, *clnk_idx*[, *q_j*[, *rtype*]])
- **clnkConfigState** (*clnk_idx*)

Parameters

- **cstate** (*logical, vector*) – (1×2) boolean vector pair to indicate which contact link is in contact with the environment (ground, object or wall).

  First element: Contact state of the *left contact link*.

  Second element: Contact state of the *right contact link*.

  A contact link touches the ground if the corresponding value is set to *true*, otherwise *false*.

- **vqT_lnk** (*double, vector*) – (7×1) VQ-transformation frame (vector-quaternion frame) relative from the given link frame *lnk* to the world frame *wf* (*optional*).
- **veT_lnk** (*double, vector*) – (6×1) VE-transformation frame (vector-Euler frame) relative from the given link frame *lnk* to the world frame *wf* (*optional*).
- **k_p** (*double, scalar*) – Stiffness control gain for the closed-loop system (*optional*).
- **k_v** (*double, scalar*) – Damping control gain for the closed-loop system (*optional*).

  Note: If the control gain is not defined, then by default the method computes the gain value for *critical damping* with $k_v = 2 \cdot \sqrt{k_p}$.

- **rtype** (*char, vector*) – Specifies the *rotation representation type* for the desired poses of the links as reference. The rotation can be represented either in quaternions `quat` or in Euler-angles `eul` (default type: `eul`).

Returns *clnk_conf (struct)* – Configuration structure that specifies the current *qualitative state* of the contact links.

footConfigState (*cstate*, *varargin*)

Creates the *foot links configuration* in dependency of the given *contact state* of the feet.

The method creates a data structure that specifies which foot of the robot is currently in contact with the ground. If required, the structure specifies also the *desired poses*, *angular velocities* and the corresponding *control gains* for the position-regulation system of the feet.

The method can be called as follows:

- **footConfigState** (*cstate*, *vqT_lnk*, *k_p*, *k_v*[, *rtype*])
- **footConfigState** (*cstate*, *veT_lnk*, *k_p*, *k_v*[, *rtype*])
- **footConfigState** (*cstate*, *q_j*, *k_p*[, *rtype*])
- **footConfigState** (*cstate*[, *q_j*[, *rtype*]])

Parameters

- **cstate** (*logical, vector*) – (1×2) boolean vector pair to indicate which foot is in contact with the ground.

  First element: Contact state of the *left foot*.

  Second element: Contact state of the *right foot*.

  A foot touches the ground if the corresponding value is set to *true*, otherwise *false*.

- **vqT_lnk** (*double, vector*) – (7×1) VQ-transformation frame (vector-quaternion frame) relative from the given link frame *lnk* to the world frame *wf* (*optional*).

- **veT_lnk** (*double, vector*) – (6×1) VE-transformation frame (vector-euler frame) relative from the given link frame *lnk* to the world frame *wf* (*optional*).
- **k_p** (*double, scalar*) – Stiffness control gain for the closed-loop system (*optional*).
- **k_v** (*double, scalar*) – Damping control gain for the closed-loop system (*optional*).

  Note: If the control gain is not defined, then by default the method computes the gain value for *critical damping* with $k_v = 2 \cdot \sqrt{k_p}$.
- **rtype** (*char, vector*) – Specifies the *rotation representation type* for the desired poses of the links as reference. The rotation can be represented either in quaternions `quat` or in Euler-angles `eul` (default type: `eul`).

Returns *foot_conf (struct)* – Configuration structure that specifies the current *qualitative state* of the feet.

See also:

WBM.clnkConfigState().

handConfigState (*cstate*, *cmode*, *varargin*)

Creates the *hand links configuration* in dependency of the given *contact state* of the hands.

The method creates a data structure that specifies which hand of the robot is currently in contact with the ground, or an object, or a wall. If required, the structure specifies also the *desired poses*, the *angular velocities* and the corresponding *control gains* for the position-regulation system of the hands.

The method can be called as follows:

- **footConfigState** (*cstate*, *cmode*, *vqT_lnk*, *k_p*, *k_v*[, *rtype*])
- **footConfigState** (*cstate*, *cmode*, *veT_lnk*, *k_p*, *k_v*[, *rtype*])
- **footConfigState** (*cstate*, *cmode*, *q_j*, *k_p*[, *rtype*])
- **footConfigState** (*cstate*, *cmode*[, *q_j*[, *rtype*]])

Parameters

- **cstate** (*logical, vector*) – (1×2) boolean vector pair to indicate which hand is in contact with the ground, or an object, or a wall.

  First element: Contact state of the *left hand*.

  Second element: Contact state of the *right hand*.

  A hand touches the ground/object/wall if the corresponding value is set to *true*, otherwise *false*.
- **cmode** (*char, vector*) – Defines the *contact mode* of the hands either with the hand palms `hand` (*hand link frame*) or with the finger tips `gripper` (*hand_dh_frame*).
- **vqT_lnk** (*double, vector*) – (7×1) VQ-transformation frame (vector-quaternion frame) relative from the given link frame *lnk* to the world frame *wf* (*optional*).
- **veT_lnk** (*double, vector*) – (6×1) VE-transformation frame (vector-euler frame) relative from the given link frame *lnk* to the world frame *wf* (*optional*).
- **k_p** (*double, scalar*) – Stiffness control gain for the closed-loop system (*optional*).
- **k_v** (*double, scalar*) – Damping control gain for the closed-loop system (*optional*).

Note: If the control gain is not defined, then by default the method computes the gain value for *critical damping* with $k_v = 2 \cdot \sqrt{k_p}$.

- **rtype** (*char, vector*) – Specifies the *rotation representation type* for the desired poses of the links as reference. The rotation can be represented either in quaternions `quat` or in Euler-angles `eul` (default type: `eul`).

Returns *hand_conf (struct)* – Configuration structure that specifies the current *qualitative state* of the hands.

See also:

WBM.clnkConfigState().

getFDynVisData (*stmChi*, *fhTrqControl*, *varargin*)

Returns the *visualization data parameters* from from the forward dynamics of the given robot model.

The method returns a *dynamic data structure* with parameter fields of the computed *forward dynamics states* $\mathcal{X}$, in dependency of the given forward dynamics method and the torque controller. The calculated parameter values of the controller and the forward dynamics can be used for the visualization of certain parameters in a plot or for further analysis purposes.

Note: To obtain the correct parameter values of the robot's forward dynamics, the torque controller and the forward dynamics method must be the same, as used in the computation of the integration output matrix $\mathcal{X}$ (see).

Following *pose correction types* for the position-regulation system of the forward dynamics will be supported:

- `none` – No pose corrections.
- `nfb` – No floating base and without pose corrections.
- `fpc` – Foot pose correction.
- `hpc` – Hand pose correction.
- `fhpc` – Foot and hand pose correction.

In dependency of the specified *pose correction*, the method can be called as follows:

none, nfb:

getFDynVisData (*stmChi*, *fhTrqControl*[, *pc_type*])

none, fpc:

getFDynVisData (*stmChi*, *fhTrqControl*, *foot_conf*, *clnk_conf*, *fe_c*, *ac*, *ac_f*[, *pc_type*])

none, fpc, fhpc:

getFDynVisData (*stmChi*, *fhTrqControl*, *fhTotCWrench*, *foot_conf*, *hand_conf*, *f_cp*, *ac_f*[, *pc_type*])

nfb:

getFDynVisData (*stmChi*, *fhTrqControl*, *clnk_conf*, *fe_c*, *ac*[, *pc_type*])

nfb:

getFDynVisData (*stmChi*, *fhTrqControl*, *fhTotCWrench*, *hand_conf*, *f_cp*[, *pc_type*])

fpc:

getFDynVisData (*stmChi*, *fhTrqControl*, *foot_conf*, *ac_f*[, *pc_type*])

hpc:

getFDynVisData (*stmChi*, *fhTrqControl*, *hand_conf*, *fe_h*, *ac_h*[, *pc_type*])

fhpc:

getFDynVisData (*stmChi*, *fhTrqControl*, *foot_conf*, *hand_conf*, *fe_h*, *ac_h*, *ac_f*[, *pc_type*])

Parameters

- **stmChi** (*double, matrix*) – Integration output matrix $\mathcal{X}$ with the calculated *forward dynamics states* (row-vectors) of the given robot model.
- **fhTrqControl** (*function_handle*) – Function handle to a specified time-dependent *torque control function* that controls the dynamics of the robot system.
- **varargin** (*cell, vector*) – Variable-length input argument list to pass extra parameters for the specified forward dynamics function to be integrated by the ODE-solver (*optional*).

  The *extra parameters* to be passed are defined in the respective forward dynamics function (see *WBM.intForwardDynamics()* – *Other Parameters*).
- **pc_type** (*char, vector*) – String and last element of `varargin` to indicate the *pose correction type* to be used for the forward dynamics ODE-function, specified by one of the given string values: `'none'`, `'nfb'`, `'fpc'`, `'hpc'`, `'fhpc'`.

  The default string is `'none'` (*optional*).

Returns *vis_data (struct)* – Dynamic structure with parameter data of the torque controller and the forward dynamics of the given robot model for visualization and analysis purposes.

See also:

WBM.getFDynVisDataSect(), *WBM.intForwardDynamics()* and *WBM.visualizeForwardDynamics()*.

getFDynVisDataSect (*stmChi*, *fhTrqControl*, *start_idx*, *end_idx*, *varargin*)

Returns a specific section of the computed the *visualization data parameters* from the forward dynamics of the given robot model.

The method returns a *dynamic data structure* with parameter fields of a specified time period of the computed *forward dynamics states* $\mathcal{X}$, in dependency of the given forward dynamics method and the torque controller. The calculated parameter values of the controller and the forward dynamics can be used for the visualization of certain parameters in a plot or for further analysis purposes.

Following *pose correction types* for the position-regulation system of the forward dynamics will be supported:

- `none` – No pose corrections.
- `nfb` – No floating base and without pose corrections.
- `fpc` – Foot pose correction.
- `hpc` – Hand pose correction.
- `fhpc` – Foot and hand pose correction.

In dependency of the specified *pose correction*, the method can be called as follows:

none, nfb:

getFDynVisDataSect (*stmChi*, *fhTrqControl*, *start_idx*, *end_idx*[, *pc_type*])

none, fpc:

getFDynVisDataSect (*stmChi*, *fhTrqControl*, *start_idx*, *end_idx*, *foot_conf*, *clnk_conf*, *fe_c*, *ac*, *ac_f*[, *pc_type*])

none, fpc, fhpc:

getFDynVisDataSect (*stmChi*, *fhTrqControl*, *start_idx*, *end_idx*, *fhTotCWrench*, *foot_conf*, *hand_conf*, *f_cp*, *ac_f* [, *pc_type*])

nfb:

getFDynVisDataSect (*stmChi*, *fhTrqControl*, *start_idx*, *end_idx*, *clnk_conf*, *fe_c*, *ac* [, *pc_type*])

nfb:

getFDynVisDataSect (*stmChi*, *fhTrqControl*, *start_idx*, *end_idx*, *fhTotCWrench*, *hand_conf*, *f_cp* [, *pc_type*])

fpc:

getFDynVisDataSect (*stmChi*, *fhTrqControl*, *start_idx*, *end_idx*, *foot_conf*, *ac_f* [, *pc_type*])

hpc:

getFDynVisDataSect (*stmChi*, *fhTrqControl*, *start_idx*, *end_idx*, *hand_conf*, *fe_h*, *ac_h* [, *pc_type*])

fhpc:

getFDynVisDataSect (*stmChi*, *fhTrqControl*, *start_idx*, *end_idx*, *foot_conf*, *hand_conf*, *fe_h*, *ac_h*, *ac_f* [, *pc_type*])

Parameters

- **stmChi** (`double, matrix`) – Integration output matrix $\mathcal{X}$ with the calculated *forward dynamics states* (row-vectors) of the given robot model.
- **fhTrqControl** (`function_handle`) – Function handle to a specified time-dependent *torque control function* that controls the dynamics of the robot system.
- **start_idx** (`int, scalar`) – Time index position (row) of the integration output matrix $\mathcal{X}$, where the section starts.
- **end_idx** (`int, scalar`) – Time index position (row) of the integration output matrix $\mathcal{X}$, where the section ends.
- **varargin** (`cell, vector`) – Variable-length input argument list to pass extra parameters for the specified forward dynamics function to be integrated by the ODE-solver (*optional*).

  The *extra parameters* to be passed are defined in the respective forward dynamics function (see `WBM.intForwardDynamics()` – *Other Parameters*).
- **pc_type** (`char, vector`) – String and last element of `varargin` to indicate the *pose correction type* to be used for the forward dynamics ODE-function, specified by one of the given string values: `'none'`, `'nfb'`, `'fpc'`, `'hpc'`, `'fhpc'`.

  The default string is `'none'` (*optional*).

Returns *vis_data (struct)* – Dynamic structure with a parameter data section of the torque controller and the forward dynamics of the given robot model for visualization and analysis purposes.

See also:

`WBM.getFDynVisData()`, `WBM.intForwardDynamics()` and `WBM.visualizeForwardDynamics()`.

setupSimulation (*sim_config*, *rot3d*)

Setups the figure window and the draw properties of the preinitialized configuration object for the visualizer of the whole body model simulation.

Parameters

- **`sim_config`** (`wbmSimConfig`) – Preconfigured configuration object with the geometric data of the robot model (joints and body shapes) to be simulated for the whole body model visualizer.

  Note: The setup method assumes, that in the given configuration are following parameters defined:

  axes: *number*, *positions* and *colors*.

  figure window: *title*, *size*, *position* and *background colors*.

  If additionally the flag for not showing the window or making a video is enabled, the figure window will be disabled and is invisible on the screen.

- **`rot3d`** (*`logical, scalar`*) – Boolean flag to indicate if the mouse-base rotation should be enabled on all axes within the figure (default: *false*) – *optional*.

  If the flag is set to *false* (default), then the mouse-base rotation is *disabled* on all axes.

Returns *sim_config (struct)* – Configuration object with the initialized figure window, axes and draw properties for the whole body model visualizer.

`visualizeForwardDynamics` (*stmPos*, *sim_config*, *sim_tstep*, *vis_ctrl*)

Visualizes the computed *forward dynamics*, i.e. the motions, of a given robot model within a specified time interval $[t_0, t_f]$.

Parameters

- **`stmPos`** (*`double, matrix`*) – Position and orientation data from the *forward dynamics states* of the integration output matrix $\mathcal{X}$ of the given robot model.
- **`sim_config`** (`wbmSimConfig`) – Configuration object with the configuration settings for the visualizer of the robot simulation.
- **`sim_tstep`** (*`double, scalar`*) – Simulation time-step size to control the visualization speed of the robot simulation.

  In order to keep the simulation close to real time, the *simulation step size* t_{step_s} is the same *step size* t_{step} of the given time interval $[t_0, t_f]$ (see *`WBM.intForwardDynamics()`* – `tspan`).

  To slow down the visualization speed of the robot simulation (slow motion), t_{step_s} can be increased by a *factor* $x > 1$ such that $t_{step_s} = x \cdot t_{step}$.

- **`vis_ctrl`** (*`struct`*) – Data structure to control the graphic elements of the robot that should be drawn during the simulation (*optional*).

  The structure is specified by following data fields:

  – `drawJnts` (*logical, scalar*): Draw the joint nodes of the robot (default: *true*).

  – `drawCom` (*logical, scalar*): Draw the node of the center of mass (default: *true*).

  – `drawSkel` (*logical, scalar*): Draw the skeleton of the robot (default: *true*).

  – `drawBody` (*logical, scalar*): Draw the body parts, i.e. the hull, of the robot (default: *true*).

  – `vis_speed` (*double, scalar*): Visualization speed to keep the simulation close to real-time when the simulation step size is changed (default: 1.0).

  Note: If the control structure is not defined, then by default all graphic elements of the robot are enabled and will be drawn in the simulation. In dependency of the complexity of the given robot model and the performance of the computer this may affect the visualization speed of the robot simulation.

See also:

`WBM.simulateForwardDynamics()`, `WBM.intForwardDynamics()` and `WBM.getPositionsData()`.

`simulateForwardDynamics` (*stmPos*, *sim_config*, *sim_tstep*, *nRpts*, *vis_ctrl*)

Simulates and visualizes the computed *forward dynamics*, i.e. the motions, of a given robot model within a specified time interval $[t_0, t_f]$.

Parameters

- **`stmPos`** (`double, matrix`) – Position and orientation data from the *forward dynamics states* of the integration output matrix $\mathcal{X}$ of the given robot model.
- **`sim_config`** (`wbmSimConfig`) – Configuration object with the configuration settings for the visualizer of the robot simulation.
- **`sim_tstep`** (`double, scalar`) – Simulation time-step size to control the visualization speed of the robot simulation.

  In order to keep the simulation close to real time, the *simulation step size* t_{step_s} is the same *step size* t_{step} of the given time interval $[t_0, t_f]$ (see `WBM.intForwardDynamics()` – `tspan`).

  To slow down the visualization speed of the robot simulation (slow motion), t_{step_s} can be increased by a *factor* $x > 1$ such that $t_{step_s} = x \cdot t_{step}$.
- **`nRpts`** (`int, scalar`) – Number of simulation repitions to be displayed on the screen.
- **`vis_ctrl`** (`struct`) – Data structure to control the graphic elements of the robot that should be drawn during the simulation (*optional*).

  The structure is specified by following data fields:

  – `drawJnts` (*logical, scalar*): Draw the joint nodes of the robot (default: *true*).

  – `drawCom` (*logical, scalar*): Draw the node of the center of mass (default: *true*).

  – `drawSkel` (*logical, scalar*): Draw the skeleton of the robot (default: *true*).

  – `drawBody` (*logical, scalar*): Draw the body parts, i.e. the hull, of the robot (default: *true*).

  – `vis_speed` (*double, scalar*): Visualization speed to keep the simulation close to real-time when the simulation step size is changed (default: 1.0).

  Note: If the control structure is not defined, then by default all graphic elements of the robot are enabled and will be drawn in the simulation. In dependency of the complexity of the given robot model and the performance of the computer this may affect the visualization speed of the robot simulation.

See also:

`WBM.visualizeForwardDynamics()`, `WBM.intForwardDynamics()` and `WBM.getPositionsData()`.

`setTrajectoriesData` (*lnk_traj*, *stmPos*, *start_idx*, *end_idx*)

Sets the positions for the data points of the given link trajectories at each evaluation point of the defined time interval or a specific time period of it.

The method calculates the data points for the trajectory curves of some certain links of the robot. The trajectories will be shown in the visualization of the robot simulation.

Parameters

- **`lnk_traj`** (`wbmLinkTrajectory`, vector) – Array of trajectory objects to show the trajectory curves of specific links of the robot.

- **stmPos** (*double, matrix*) – Position and orientation data from the *forward dynamics states* of the integration output matrix $\mathcal{X}$ of the given robot model.
- **start_idx** (*int, scalar*) – Time index (row) of the position and orientation data matrix `stmPos`, where the interval starts.
- **end_idx** (*int, scalar*) – Time index (row) of the position and orientation data matrix `stmPos`, where the interval ends.

Returns lnk_traj (`wbmLinkTrajectory`, vector) – Array of trajectory objects with the calculated data points of each link trajectory.

See also:

`wbmLinkTrajectory` and *WBM.getPositionsData()*.

plotCoMTrajectory (*stmPos*, *prop*)

Plots the trajectory curve of the center of mass (CoM) of the simulated floating-base robot.

Parameters

- **stmPos** (*double, matrix*) – Position and orientation data from the *forward dynamics states* of the integration output matrix $\mathcal{X}$ of the given robot model.
- **prop** (*struct*) – Data structure to specify the plot properties of trajectory curve of the CoM, i.e. the appearance and behavior of the line object (*optional*).

  The plot properties are specified by following fields:

  – `fwnd_title` (*char, vector*): The figure-window title of the trajectory plot.

  – `title` (*char, vector*): The title of the trajectory curve of the CoM.

  – `title_fnt_sz` (*double, scalar*): Font size of the trajectory title, specified as a positive value in points (default value: 15).

  – `line_color` (*double/char, vector*): Color of the trajectory curve, specified by a RGB-triplet or a color name (default color: `'blue'`).

  – `spt_marker` (*double, scalar*): Marker symbol for the start point of the trajectory curve. The symbols for the marker are the same as specified in Matlab (default symbol: `'*'`).

  – `spt_color` (*double, scalar*): Color of the trajectory start point, specified by a RGB-triplet or a color name (default color: `'red'`).

  – `label_fnt_sz` (*double, scalar*): Font size of the axis labels, specified as a positive value in points (default value: 15).

  Note: If the plot properties are not given, then the default settings will be used.

See also:

WBM.getPositionsData().

setPayloadLinks (*pl_lnk_data*)

Assigns the data of the given payload objects to specific *payload links* of the floating-base robot.

Parameters **pl_lnk_data** (*struct/cell-array*) – $(1 \times n_{plds})$ data structures of the given *payload objects*, where each payload will be assigned to a specific link of the floating-base robot with $n_{plds} \geq 1$.

If the robot has multiple payload objects for the simulation, then the data structures of each payload must be stored in a cell-array.

The data structure for the payload objects contains following fields:

- `name` (*char, vector*): URDF-name of the specified *reference link frame* or *contact link frame*. The link frame can be an end-effector frame of a hand or a special frame on which a payload object is mounted.

- `lnk_p_cm` (*double, vector*): (3×1) Cartesian position vector relative from the center of mass *cm* of the payload object to the link frame *lnk*.
- `t_idx` (*uint8, scalar*): Index number of a specified *tool link object* to define that the grabbed payload object is also a tool (*optional*).

  Note: If the index value is 0, then the payload object is not linked with a tool.
- `vb_idx` (*uint8, scalar*): Index number of a *volume body object* that will be linked to the given link frame (*optional*).

  Note: If the index value is 0, then the payload is not assigned to any given geometric volume body of the simulation environment.
- `m_rb` (*double, scalar*): Mass of the rigid body (solid volume body) in [kg] (*optional*).
  - `I_cm` (*double, matrix*): (3×3) *inertia tensor* of a rigid body or geometric object with the origin of the coordinate system at the center of mass (CoM) of the object body.

  Note: The mass `m_rb` is linked with the inertia `I_cm` at the CoM of the given rigid body. So if `m_rb` is undefined or the value is zero, then also `I_cm` is undefined. This can be helpful if the mass of the object is currently unknown.

Note: The indices 1 and 2 are always reserved for the *left* and the *right hand* of a given humanoid robot. The default index that will be used by the *WBM* class is always 1. All further indices (> 2) of the array can be used for other links, on which additionally special payloads are mounted (e.g. a battery pack or a knapsack at the torso, special tools, etc.).

See also:

WBM.getPayloadLinks() and `wbmPayloadLink`.

getPayloadLinks()

Returns all defined payload links of the floating-base robot with the assigned payload objects.

Returns

[pl_links, nPlds] –

2-element tuple containing:

- **pl_links** (`wbmPayloadLink`, *vector*) – $(1 \times n_{plds})$ array of *payload link objects*, where each payload is assigned to a specific link of the robot (default: *empty*).
- **nPlds** (*uint8, scalar*) – Number of payloads that are assigned to specific links of the robot.

See also:

WBM.setPayloadLinks() and `wbmPayloadLink`.

getPayloadTable()

Creates a table of the defined payload links with the assigned payload objects.

The output table lists all data values of the each defined payload link, specified by following column names as variables: `link_name`, `lnk_p_cm`, `t_idx`, `vb_idx`, `mass`, `inertia`.

Returns *pl_tbl (table)* – Table array with named variables for the payload object data of all defined payload links of the robot.

See also:

WBM.setPayloadLinks() and `wbmPayloadLink`.

payloadFrame(*varargin*)

Computes the *frame*, i.e. the transformation matrix H, of a specified *payload link* w.r.t. the current joint configuration q_j.

The homogeneous transformation of the grabbed or selected payload, relative from the payload's center of mass *cm* to the world frame *wf*, will be obtained as follows:

$$^{wf}H_{cm} = {}^{wf}H_{lnk} \cdot {}^{lnk}H_{cm} \tag{2.16}$$

The method can be called by one of the given modes:

Normal mode – Computes the transformation matrix of the given payload link, specified by the base orientation and the positions:

> `payloadFrame` (*wf_R_b*, *wf_p_b*, *q_j*[, *pl_idx*])

Optimized mode – Computes the transformation matrix of the given payload link at the current state of the robot system:

> `payloadFrame` ([*pl_idx*])

Parameters

- **`wf_R_b`** (`double, matrix`) – (3×3) rotation matrix (orientation) from the base frame *b* to world frame *wf*.
- **`wf_p_b`** (`double, vector`) – (3×1) position vector from the base frame *b* to the world frame *wf*.
- **`q_j`** (`double, vector`) – $(n_{dof} \times 1)$ joint positions vector in [rad].
- **`pl_idx`** (`uint8, scalar`) – Index number of the specified *payload link object* of the robot (*optional*).

  Note: If the index of the payload link object is undefined, then the first element of the list will be used as default.

Note: It is assumed that the orientation of the payload frame *pl* has the same orientation as the given payload link frame *lnk* of the robot, i.e. the link of an end-effector (hand, finger, etc.) or another specific link (torso, leg, etc.) where the payload is mounted on that body part.

Returns *wf_H_cm (double, matrix)* – (4×4) homogeneous transformation matrix relative from the payload's center of mass *cm* to the world frame *wf*.

See also:

`WBMBase.transformationMatrix().`

`dynPayloadForce` (*pl_idx*, *wf_v_lnk*, *wf_a_lnk*, *w_e*)

Computes the *dynamic payload force* f_{pl}, i.e. the payload wrench that is acting on the contact point p_{C_i} of the payload object, in dependency of a given external wrench w_e.

For the simplification of the distances, the given payload link frame *lnk* represents the contact frame $\mathrm{C}_{\mathrm{i} \in \{1,\ldots,\mathrm{n}\}}$ at the contact point p_{C_i}.

Parameters

- **`pl_idx`** (`uint8, scalar`) – Index number of the specified *payload link object* of the robot (*optional*).
- **`wf_v_lnk`** (`double, vector`) – (6×1) mixed velocity vector from the link frame *lnk* to the world frame *wf*.
- **`wf_a_lnk`** (`double, vector`) – (6×1) mixed acceleration vector from the link frame *lnk* to the world frame *wf*.

- **w_e** (*double, vector*) – (6×1) external wrench vector that is acting on the contact point p_{C_i} (which is at the link frame *lnk*).

Note: All input values must be from the same reference frame *lnk*.

Returns *f_pl (double, vector)* – $(6{\times}1)$ dynamic payload force vector at the specified payload link of the robot.

See also:

`utilities.mbd.payloadWrenchRB()` and *`WBM.generalizedInertiaPL()`*.

`generalizedInertiaPL`(*pl_idx*)

Computes the *generalized inertia matrix* of the given payload object at the contact point p_{C_i} that is associated with a specified payload link *lnk*.

The method applies a simplified position and orientation determination for the center of mass (CoM) of the payload, such that

$$\begin{aligned} {}^{lnk}R_{cm} &= {}^{lnk}R_{C_i} \cdot {}^{C_i}R_{cm} \quad \text{and} \\ {}^{lnk}p_{cm} &= {}^{lnk}R_{C_i} \cdot {}^{C_i}p_{cm}\ , \end{aligned} \tag{2.17}$$

where ${}^{lnk}R_{cm} = {}^{lnk}R_{C_i} = {}^{C_i}R_{cm} = \mathbb{I}_3$ and ${}^{lnk}p_{cm} = {}^{C_i}p_{cm}$. The contact frame $\mathrm{C}_{\mathrm{i}\in\{1,\ldots,\mathrm{n}\}}$ and the payload frame *cm* (centered at CoM) have the same orientation as the reference link frame *lnk*. The contact point $p_{C_i} = {}^{lnk}p_{cm}$ is set at the origin o_{lnk} of the reference link frame.

Note: The given position for the object's CoM must be from the same reference frame *lnk*.

Parameters **pl_idx** (*uint8, scalar*) – Index number of the specified *payload link object* of the robot (*optional*).

Returns *M_pl (double, matrix)* – (6×6) generalized inertia matrix of the given payload object (from the payload frame *cm* to the link frame *lnk*).

See also:

`utilities.rb.generalizedInertia()`.

`setToolLinks`(*ee_lnk_names*, *ee_vqT_tt*, *pl_idx*)

Assigns the data of the given tools to specific *tool links* (end-effectors) of the floating-base robot.

The method can be called as follows:

`setToolLinks`(*ee_lnk_names*, *ee_vqT_tt*[, *pl_idx*])

It assigns the *tool-tip frames* (tt) of the given tools to specific *end-effector links* (ee) of the robot. If a given payload object is at the same time also a tool, then the method links the tool object with the corresponding payload object.

Parameters

- **ee_link_names** (*cellstr, vector*) – $(1{\times}n)$ array of string matching URDF-names of the *end-effector links*.
- **ee_vqT_tt** (*double, matrix*) – $(7 \times n)$ frame matrix for the VQ-transformation frames of the given tools, relative from the tool-tip frames *tt* to the link frames of the end-effectors *ee*.

- **pl_idx** (*uint8, vector*) – $(1 \times n)$ index number array of the specified *payload link objects*, to define that specific payload objects are also tools (default array: *array of zeros*) – *optional*.

  Note: If an index value of the array is set to 0, then the tool is not linked with a given payload object.

Note: All input arrays must have the same length n with $1 \leq n \leq$ MAX_NUM_TOOLS.

See also:

wbmToolLink, *WBM.getToolLinks()* and *WBM.MAX_NUM_TOOLS*.

getToolLinks()

Returns all defined tool links of the floating-base robot with the assigned tool objects.

Returns

[tool_links, nTools] –

2-element tuple containing:

- **tool_links** (wbmToolLink, *vector*) – $(1 \times n_{tools})$ array of *tool link objects*, where each tool is assigned to a specific end-effector link of the robot (default: *empty*).
- **nTools** (*uint8, scalar*) – Number of tools that are assigned to specific end-effector links of the robot.

See also:

WBM.setToolLinks() and wbmToolLink.

getToolTable()

Creates a table of the defined tool links with the assigned tool objects.

The output table lists all data values of each defined tool link, specified by following column names as variables: link_name, ee_vqT_tt, pl_idx.

Returns *tool_tbl (table)* – Table array with named variables for the tool object data of all defined tool links of the robot.

See also:

WBM.setToolLinks() and wbmToolLink.

updateToolFrame(*ee_vqT_tt*, *t_idx*)

Updates the tool frame, i.e. the VQ-transformation, of the selected tool.

Parameters

- **ee_vqT_tt** (*double, vector*) – (7×1) VQ-transformation frame of the tool, relative from the tool-tip frame *tt* to the link frame of the end-effector *ee*.
- **t_idx** (*uint8, scalar*) – Index number of the *tool link object* in the list that should be updated.

Note: The index number i_t of the tool link object must be in the range of $1 \leq i_t \leq$ MAX_NUM_TOOLS.

See also:

wbmToolLink, *WBM.getToolLinks()* and *WBM.MAX_NUM_TOOLS*.

toolFrame(*varargin*)

Computes the *tool frame*, i.e. the transformation matrix H, of a specified tool link w.r.t. the current joint configuration q_j.

The homogeneous transformation of the chosen tool, relative from the tool-tip frame *tt* to the world frame *wf*, will be obtained as follows:

$$^{wf}H_{tt} = {}^{wf}H_{ee} \cdot {}^{ee}H_{tt} \tag{2.18}$$

The method can be called by one of the given modes:

Normal mode – Computes the transformation matrix of the given tool link, specified by the base orientation and the positions:

toolFrame(*wf_R_b*, *wf_p_b*, *q_j*[, *t_idx*])

Optimized mode – Computes the transformation matrix of the given tool link at the current state of the robot system:

toolFrame([*t_idx*])

Parameters

- **wf_R_b** (*double, matrix*) – (3×3) rotation matrix (orientation) from the base frame *b* to world frame *wf*.
- **wf_p_b** (*double, vector*) – (3×1) position vector from the base frame *b* to the world frame *wf*.
- **q_j** (*double, vector*) – $(n_{dof} \times 1)$ joint vector in $[\text{rad}]$.
- **t_idx** (*uint8, scalar*) – Index number of the specified *tool link object* of the robot (*optional*).

  Note: If the index of the tool link object is not given, then the first element of the list will be used as default.

Note: In general the orientation of the tool frame *tt* has not the same orientation as the frame of the end-effector *ee* of a hand or of a finger.

Returns *wf_H_tt (double, matrix)* – (4×4) homogeneous transformation matrix relative from the tool-tip frame *tt* to the world frame *wf*.

See also:

`WBMBase.transformationMatrix().`

jacobianTool(*varargin*)

Computes the *Jacobian* of a specified *tool link* of the robot w.r.t. the current joint configuration q_j.

The Jacobian of the given tool link will be obtained by applying the *velocity transformation matrix*

$$^{tt[wf]}X_{ee[wf]} = \begin{bmatrix} \mathbb{I}_3 & \text{-}S(^{wf}R_{ee} \cdot {}^{ee}p_{tt}) \cdot \mathbb{I}_3 \\ \mathbf{0} & \mathbb{I}_3 \end{bmatrix}, \tag{2.19}$$

such that $^{wf}J_{tt} = {}^{tt[wf]}X_{ee[wf]} \cdot {}^{ee[wf]}X_{ee}$, where the notations $tt[wf]$ and $ee[wf]$ denoting the frames with origin o_{tt} and o_{ee} with the orientation $[wf]$. The transformation matrix X maps the velocities of the geometric Jacobian in frame *ee* to the velocities in frame *tt*.

The method can be called by one of the given modes:

Normal mode – Computes the Jacobian matrix of the given tool link, specified by the base orientation and the positions:

`jacobianTool` (*wf_R_b*, *wf_p_b*, *q_j*[, *t_idx*])

Optimized mode – Computes the Jacobian matrix of the given tool link at the current state of the robot system:

`jacobianTool` ([*t_idx*])

Parameters

- **wf_R_b** (*double, matrix*) – (3×3) rotation matrix (orientation) from the base frame *b* to world frame *wf*.
- **wf_p_b** (*double, vector*) – (3×1) position vector from the base frame *b* to the world frame *wf*.
- **q_j** (*double, vector*) – $(n_{dof} \times 1)$ joint positions vector in $[\text{rad}]$.
- **t_idx** (*uint8, scalar*) – Index number of the specified *tool link object* of the robot (*optional*).

  Note: If the index of the tool link object is not given, then the first element of the list will be used as default.

Returns *wf_J_tt (double, matrix)* – $(6 \times n)$ Jacobian matrix relative from the tool-tip frame *tt* to the world frame *wf* with $n = n_{dof} + 6$.

See also:

`WBMBase.jacobian()` and `utilities.tfms.adjoint()`.

References

- *[TS16]*, p. 6, eq. (27).
- *[SSVO10]*, p. 150, eq. (3.112).
- *[Cra05]*, p. 158, eq. (5.103).

`getStateJntChains` (*chain_names*, *q_j*, *dq_j*)
Returns the current *joint configuration states* of those chains of the robot body that are specified in the given list of *chain names*.

The method can be called as follows:

`getStateJntChains` (*chain_names*[, *q_j*, *dq_j*])

Parameters

- **chain_names** (*cellstr, vector*) – Column-array with $k \geq 1$ string matching *chain names* of the robot's effectors.
- **q_j** (*double, vector*) – $(n_{dof} \times 1)$ vector of the current joint positions of the robot in $[\text{rad}]$ (*optional*).
- **dq_j** (*double, vector*) – $(n_{dof} \times 1)$ vector of the current joint velocities of the robot in $[\text{rad/s}]$ (*optional*).

Returns

[chn_q, chn_dq] –

2-element tuple containing:

- **chn_q** (*cell, vector*) – $(k \times 1)$ array of joint position vectors of the specified chain names of the robot body.
- **chn_dq** (*cell, vector*) – $(k \times 1)$ array of joint velocity vectors of the specified chain names of the robot body.

Note: The order of the joint position and velocity vectors are in the same order as the given chain names list.

getStateJointNames (*joint_names*, *q_j*, *dq_j*)

Returns the current *joint configuration states* of those joints of the robot that are specified in the given *joint names list*.

The method can be called as follows:

getStateJointNames (*joint_names*[, *q_j*, *dq_j*])

Parameters

- **joint_names** (*cellstr, vector*) – Column-array with $k \geq 1$ string matching *joint names* of the robot.
- **q_j** (*double, vector*) – $(n_{dof} \times 1)$ vector of the current joint positions of the robot in [rad] (*optional*).
- **dq_j** (*double, vector*) – $(n_{dof} \times 1)$ vector of the current joint velocities of the robot in [rad/s] (*optional*).

Returns

[jnt_q, jnt_dq] –

2-element tuple containing:

- **jnt_q** (*double, vector*) – $(k \times 1)$ joint position vector of the specified joint names of the robot.
- **jnt_dq** (*double, vector*) – $(k \times 1)$ joint velocity vector of the specified joint names of the robot.

Note: The order of the joint positions and velocities are in the same order as the given joint names list.

getStateJointIdx (*jnt_idx*, *q_j*, *dq_j*)

Returns the current *joint configuration states* of those joints of the robot that are specified in the given *joint index list*.

The method can be called as follows:

getStateJointIdx (*jnt_idx*[, *q_j*, *dq_j*])

Parameters

- **jnt_idx** (*int, vector*) – $(k \times 1)$ vector with the index numbers of the specified joints of the robot model in ascending order.
- **q_j** (*double, vector*) – $(n_{dof} \times 1)$ vector of the current joint positions of the robot in [rad] (*optional*).
- **dq_j** (*double, vector*) – $(n_{dof} \times 1)$ vector of the current joint velocities of the robot in [rad/s] (*optional*).

Returns

[jnt_q, jnt_dq] –

2-element tuple containing:

- **jnt_q** (*double, vector*) – $(k \times 1)$ joint position vector of the specified joint indices of the robot.
- **jnt_dq** (*double, vector*) – $(k \times 1)$ joint velocity vector of the specified joint indices of the robot.

Note: The order of the joint positions and velocities are in the same order as the given joint names list.

`getStateParams`(*stChi*)

Returns the *state parameters* of the computed *forward dynamics states* of the robot model.

Parameters **`stChi`** (`double, vector/matrix`) – Integration output vector or matrix $\mathcal{X}$ containing the computed *forward dynamic states* of the robot model.

Note: In dependency of the situation, the integration output $\mathcal{X}$ represents either a *single state* (column-vector) or a *time sequence of states* (matrix).

Returns

stParams (`wbmStateParams`) – Data object with the state parameters of the forward dynamics states of the robot model.

Note: If the given integration output `stChi` is a *time sequence of states* then the state variables of the data object are matrices otherwise, column-vectors.

`getPositions`(*stChi*)

Returns the *base frames* and *joint positions* of the computed *forward dynamics states* of the floating-base robot.

Parameters **`stChi`** (`double, vector/matrix`) – Integration output vector or matrix $\mathcal{X}$ containing the computed *forward dynamic states* of the robot model.

Note: In dependency of the situation, the integration output $\mathcal{X}$ represents either a *single state* (column-vector) or a *time sequence of states* (matrix).

Returns

[vqT_b, q_j] –

2-element tuple containing:

- **vqT_b** (*double, vector/matrix*) – Base frames array (base VQ-transformations) of the floating-base robot.
- **q_j** (*double, vector/matrix*) – Joint positions array of the floating-base robot in $[\mathrm{rad}]$.

Note: If the given integration output `stChi` is a *time sequence of states*, then the return values are data matrices. If `stChi` represents only a *single state*, the return values are column-vectors.

`getPositionsData`(*stmChi*)

Returns the *position* and *orientation data* of a given time sequence of *forward dynamics states* of the robot model.

Parameters **`stmChi`** (`double, matrix`) – Integration output matrix $\mathcal{X}$ with the calculated *forward dynamics states* (row-vectors) of the given robot model.

Returns *stmPos (double, matrix)* – $(m \times (n_{dof} + 7))$ data matrix with position and orientation data $[x_b, qt_b, q_j]$ of the forward dynamics states of the robot model.

`getMixedVelocities`(*stChi*)

Returns the *mixed velocities* of the computed *forward dynamics states* of the robot model.

Parameters **`stChi`** (`double, vector/matrix`) – Integration output vector or matrix $\mathcal{X}$ containing the computed *forward dynamic states* of the robot model.

Note: In dependency of the situation, the integration output $\mathcal{X}$ represents either a *single state* (column-vector) or a *time sequence of states* (matrix).

Returns

[v_b, dq_j] –

2-element tuple containing:

- **v_b** (*double, vector/matrix*) – Generalized base velocities array (Cartesian and rotational velocities of the base).
- **dq_j** (*double, vector/matrix*) – Joint velocities array of the robot model in $[\mathrm{rad/s}]$.

Note: If the given integration output `stChi` is a *time sequence of states*, then the return values are data matrices. If `stChi` represents only a *single state*, the return values are column-vectors.

`getBaseVelocities`(*stChi*)

Returns the *generalized base velocities* of the computed *forward dynamics states* of the robot model.

Parameters **`stChi`** (`double, vector/matrix`) – Integration output vector or matrix $\mathcal{X}$ containing the computed *forward dynamic states* of the robot model.

Note: In dependency of the situation, the integration output $\mathcal{X}$ represents either a *single state* (column-vector) or a *time sequence of states* (matrix).

Returns

v_b (double, vector/matrix) – Generalized base velocities array of the forward dynamics states of the robot model.

Note: If the given integration output `stChi` is a *time sequence of states* then `v_b` is a matrix, otherwise a column-vector.

`dispConfig`(*prec*)

Displays the current configuration settings of the given floating-base robot.

Parameters **`prec`** (`int, scalar`) – Precision number to specify for the values the number of digits after the decimal point (default precision: 2) – *optional*.

BIBLIOGRAPHY

[Cor17] Peter Corke. *Robotics, Vision and Control: Fundamental Algorithms In MATLAB*. Volume 118 of Springer Tracts in Advanced Robotics. Springer, 2nd edition, 2017.

[Cra05] John J. Craig. *Introduction to Robotics: Mechanics and Control*. Pearson/Prentice Hall, 3rd edition, 2005.

[Fea08] Roy Featherstone. *Rigid Body Dynamics Algorithms*. Springer-Verlag, Berlin, Heidelberg, 2008.

[Kha11] Wisama Khalil. Dynamic modeling of robots using newton-euler formulation. In Juan Andrade Cetto, Jean-Louis Ferrier, and Joaquim Filipe, editors, *Informatics in Control, Automation and Robotics: Revised and Selected Papers from the International Conference on Informatics in Control, Automation and Robotics 2010*, volume 89 of Lecture Notes in Electrical Engineering, 3–20. Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[Kha87] Oussama Khatib. A unified approach for motion and force control of robot manipulators: the operational space formulation. *IEEE Journal of Robotics and Automation*, RA-3(1):43–53, February 1987. URL: https://cs.stanford.edu/group/manips/publications/pdfs/Khatib_1987_RA.pdf.

[Lil92] Kathryn Lilly. *Efficient Dynamic Simulation of Robotic Mechanisms*. The Springer International Series in Engineering and Computer Science. Springer, 1992.

[MLS94] Richard M. Murray, Zexiang Li, and S. Shankar Sastry. *A Mathematical Introduction to Robotic Manipulation*. CRC Press, Boca Raton, FL, USA, 1994.

[Par06] Jaeheung Park. *Control Strategies for Robots in Contact*. PhD thesis, Artificial Intelligence Laboratory, Department of Computer Science, March 2006. URL: http://cs.stanford.edu/group/manips/publications/pdfs/Park_2006_thesis.pdf.

[SS00] Lorenzo Sciavicco and Bruno Siciliano. *Modelling and Control of Robot Manipulators*. Advanced Textbooks in Control and Signal Processing. Springer-Verlag, Berlin, Heidelberg, 2nd edition, 2000.

[SSD12] Suril Vijaykumar Shah, Subir Kumar Saha, and Jayanta Kumar Dutt. *Dynamics of Tree-Type Robotic Systems*. Volume 62 of Intelligent Systems, Control and Automation: Science and Engineering. Springer, 2012.

[SR97] Lawrence F. Shampine and Mark W. Reichelt. The matlab ode suite. *SIAM Journal on Scientific Computing*, 18(1):1–22, January 1997. URL: http://dx.doi.org/10.1137/S1064827594276424.

[SSVO10] Bruno Siciliano, Lorenzo Sciavicco, Luigi Villani, and Giuseppe Oriolo. *Robotics: Modelling, Planning and Control*. Advanced Textbooks in Control and Signal Processing. Springer, 2010.

[TS16] S. Traversaro and A. Saccon. Multibody dynamics notation. Technical Report, Technische Universiteit Eindhoven, July 2016. Dept. of Mechanical Engineering. Report locator DC 2016.064. URL: https://pure.tue.nl/ws/portalfiles/portal/25753352.

MATLAB MODULE INDEX

INDEX