`manipulator` library for **Maxima** CAS

Stanislav Mikhel

2021

1 About

The `manipulator` library is developed for working with kinematic trees such as robotic manipulators in the **Maxima** computer algebra system. It is focused on the generation of "raw" equations, additional simplification of results can be done later if needs. The library allows to do the following operations:

- define the kinematic tree structure manually or import it from URDF file
- solve forward kinematics problem, find the Jacobian
- solve inverse dynamics problem, find matrices M, C, and G
- visualize robot structure and print the tree structure
- use different parameterization for rotations
- save result in form of **Matlab** or **C** code

2 Creating a robot model

2.1 Kinematic tree

The library deals with tree-like structures, which represent a sequence of links connected via joints. Each joint can be connected to only one link, but a link can have several joints (Fig. 1). A joint can be fixed, rotational (revolute), or translational (prismatic). A link can be real (with inertial parameters) or virtual. The proper tree model in this library must begin with the joint element.

2.2 Explicit tree structure definition

The user can explicitly define links and joints and connect them to make a tree. This method is the most flexible but time-consuming technique.

Links and joints are defined in form of the **Maxima** structures. Each joint object stores information about its location concerning to the previous joint and indexes of the predecessor, successor, and the object itself (`id`). If the join is movable, it includes a function of transformation and torque value.

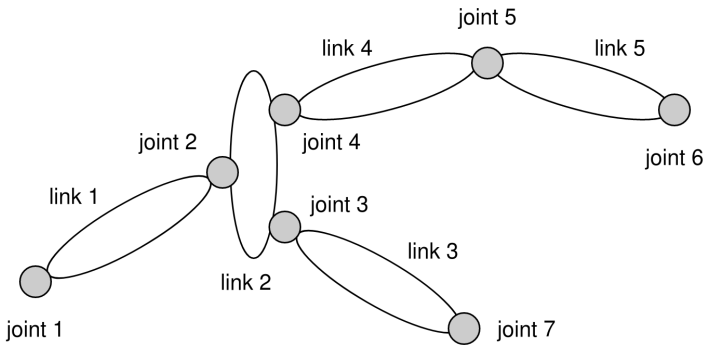

Figure 1: Kinematic tree example.

The joint can be defined with function `Joint`. It takes 3 arguments: the first one is a transformation function, the second and third are rotation and position of the joint in the previous joint frame, the rotation is a matrix and position is a vector. The function of transformation defines not only the type of motion (such as `rotx` or `tranz`) but also the name of the joint angle and its derivatives. For example, it you define transformation function as `rotz(q1)`, then the joint velocity and acceleration called `dq1` and `ddq1` respectively. If you want to skip any argument of the transformation function, just use empty list, for example, `Joint([],[],[])` creates fixed joint with a unit transformation matrix and zero translation vector.

The link structure stores information about the object `id`, its predecessor and list of successors. If can additionally contain list of 10 inertial parameters:

- $I_{xx}, I_{xy}, I_{xz}, I_{yy}, I_{yz}, I_{zz}$ - elements of the inertia tensor
- r_{mx}, r_{my}, r_{mz} - position of mass center w.r.t. to parent joint
- m - mass of the link

The link can be created using function `Link`, which takes the list of inertial parameters of the empty list in the case of the virtual link.

Links and joints can be connected with the help of function `add_child`. It is more convenient to use the infix operator "`++`". It returns a pointer to the last term that allows connecting a long chain of sequential elements. When you connect a link to a link or joint to joint, the intermediate element (virtual joint or link respectively) is generated automatically.

Pointer to the object (link or joint) can be obtained using function `by_id` if the object `id` is known. If the joint variable is known, the corresponding joint can be found with the help of `by_name` function.

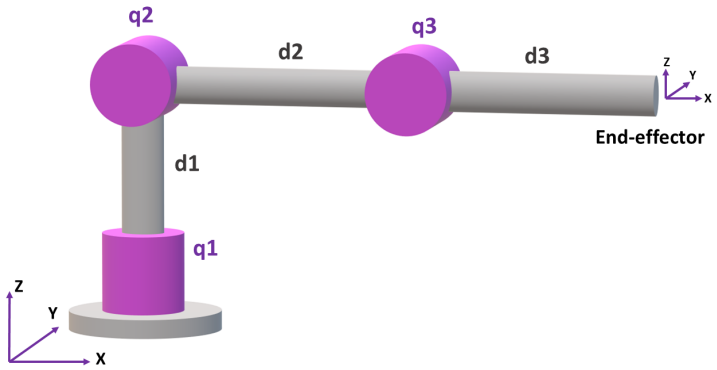

Figure 2: Manipulator with 3 links.

2.2.1 Example: making tree

The following code allows to create sequence shown in figure 1 and print its structure. All the links here can be defined implicitly except the "link2" that is required to make the second branch.

```
lnk2 : Link([])$
/* Save pointer to the last joint */
seq : Joint([],[],[]) ++ Joint([],[],[]) ++ lnk2
   ++ Joint([],[],[]) ++ Joint([],[],[]) ++ Joint([],[],[])$
/* Add second branch */
lnk2 ++ Joint([],[],[]) ++ Joint([],[],[])$
/* Show the structure */
print_tree(seq);
```

2.2.2 Example: 3 link manipulator

Let us define the structure of a simple 3-link manipulator. Figure 2 shows its initial initial configuration. Inertial parameters here are calculated using auxiliary functions from the file *inertia.mac*.

```
/* link parameters: length, external and internal radius, m */
d1 : 0.3; r1e : 0.03; r1i : 0.02;  /* 1st */
d2 : 0.5; r2e : 0.03; r2i : 0.02;  /* 2nd */
d3 : 0.5; r3e : 0.02; r3i : 0.015; /* 3rd */
rho : 2700;  /* kg/m^3, alluminium */
/* links */
lnk1 : Link(inertia_cylinder_z(d1,r1e,r1i,rho))$
lnk2 : Link(inertia_cylinder_x(d2,r2e,r2i,rho))$
lnk3 : Link(inertia_cylinder_x(d3,r3e,r3i,rho))$
/* joints */
```

```
jnt1 : Joint(rotz(q1),[],[])$
jnt2 : Joint(roty(q2),[],Pos(0,0,d1))$
jnt3 : Joint(roty(q3),[],Pos(d2,0,0))$
ee  : Joint([],[],Pose(d3,0,0))$
/* connect */
rbt : jnt1 ++ lnk1 ++ jnt2 ++ lnk2 ++ jnt3 ++ lnk3 ++ ee$
/* find elements */
by_id( lnk1@id );   /* the same as "lnk1" */
by_name(q2);        /* the same as "jnt2" */
```

2.3 List of transformations

You can use the known sequence of transformations along the manipulator to define its structure with the help of `from_list` function. It has two arguments, the first one is the list of transformations, while the second is the list of joint names. The first argument can include any number of translations and rotations with variable or constant parameters. Sequential constant matrices are multiplied to simplified the result. The list of arguments allows to include additional parameters in the robot structure.

2.3.1 Example: list of transformations

The kinematical structure of the manipulator (Ex. 2.2.2) can be defined as follows.

```
seq : [rotz(q1),tranz(0.3),roty(q2),tranx(0.5),
       roty(q3),tranx(L)]$  /* last length is a parameter */
rbt : from_list(seq, [q1,q2,q3])$
```

2.4 Denavit-Hartenberg convention

If you prefer DH convention for the manipulator description, send it to the function `from_dh`. The order of parameters for each link is following: translation in X (a), rotation around X (α), translation in Z (d), rotation around Z (θ).

2.4.1 Example: list of DH parameters

Let us rewrite the kinematic structure of Ex. 2.2.2 using DH parameters.

```
seq : [
/*  a  alpha    d theta */
 [  0,-%pi/3, 0.5,   q1],
 [0.5,     0,   0,   q2],
 [  L,     0,   0,   q3]    /* length is a parameter */
];
rbt : from_dh(seq, [q1,q2,q3])$
```

2.5 URDF file

Universal Robot Description Format (URDF) is a popular XML-based file format for robot description. The `manipulator` includes *fromxml.mac* library with simple XML parser that allow to deal with valid URDF files. If you don't want to deal with "raw" XML node, use function `from_urdf`.

2.5.1 Example: read URDF

Assume, you want to read file from the *tests* directory. Then just call the function.

```
rbt : from_urdf("tests/3links.urdf")$
```

3 Main operations

Definition of the kinematic tree structure does not include any additional operations other than element connection. For further analysis, you have to use additional functions. Almost in any case, the first procedure should be `update_state`. It calculates the kinematic relations between the tree components. The argument of this function is any joint of the robot (if you defined the manipulator using any `from_` function, it is the last joint).

3.1 Forward kinematics

For each joint, you can call function `get_pos` to get the position vector of this joint in the robot base frame. Likewise, the `get_rot` returns the joint rotation matrix.

3.1.1 Example: FK

For the manipulator (Ex. 2.2.2) we can solve the FK problem.

```
/* calculate kinematics */
update_state(rbt);
/* get joint position as expression */
get_pos(jnt2);
/* find rotation for some joint state */
ev(get_rot(jnt2), q1:%pi/4, q2:%pi/3);
```

3.2 Differential kinematics

The Jacobian allows connecting the velocity of the end-effector (or another part of the robot) with the rate of change in joint space. Use function `Jacobian` to calculate it.

3.2.1 Example: Jacobian

Continue to work with Ex. 2.2.2.

```
/* find Jacobian for the robot end-effector */
jac : Jacobian(rbt)$
/* numerical evaluation */
res : ev(jac, q1:%pi/2, q2:%pi/3, q3:%pi/4);
```

3.3 Inverse dynamics

Dynamics calculation in the `manipulator` is based on the recursive Newton-Euler algorithm and implemented in the function `update_dynamics`. It uses the results of `update_state` evaluation and has to be called only when the kinematic tree was created. To extract the joint torque expression for any joint from the tree structure one can call function `get_torque`.

3.3.1 Example: dynamics

Assume, that in the 2.2.2 we didn't call the method `update_state`.

```
/* find kinematics */
update_state(rbt)$
/* find dynamics */
update_dynamics(rbt)$
/* define parameters */
par : [q1=0.1,dq1=0.5,ddq1=1,q2=-0.2,dq2=0.5,ddq2=1,
       q3=0.3,dq3=0.5,ddq3=1,grav=9.81]$
/* find torques */
res : ev([get_torque(jnt1),get_torque(jnt2),
          get_torque(jnt3)],par)$
```

3.4 Dynamics matrices

The results, obtained in the section 3.3 can be transformed into the matrix representation with the help of function `M_C_G_J`. It returns a list with four elements: the inertia matrix, the matrix of Coriolis and centrifugal terms, vector of gravity components, and the list of joints that corresponds to the sequence of elements in the found matrices.

3.4.1 Example: matrices

Assume that `update_dynamics` has been called previously.

```
/* find matrices */
[M,C,G,jnts] : M_C_G_J(rbt)$
/* check the result */
/* get velocities */
```

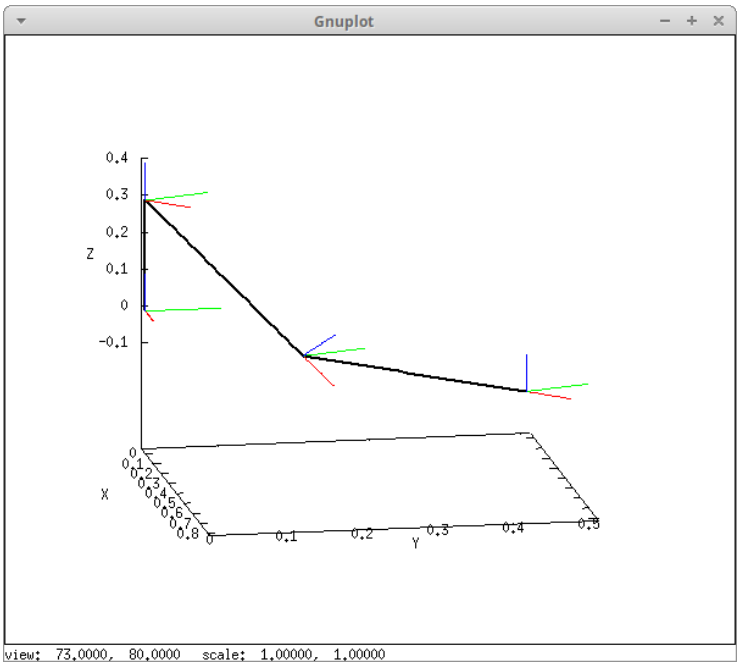

Figure 3: Robot drawing.

```
dqs : makelist(get_dq(x),x,jnts)$
/* get accelerations */
ddqs : makelist(get_ddq(x),x,jnts)$
/* must be equal to the previous result */
res : ev(M . transpose(ddqs) + C . transpose(dqs) + G, par)$
```

3.5 Drawing

Use the function `draw_tree` to plot the robot structure. It takes two arguments: the robot object and the list of joint angles. If the joint list is empty (or some angles are skipped) they are assumed to be zero.

3.5.1 Example: drawing

Let us visualize the manipulator in the Ex. 2.2.2 when the first joint is $\pi/6$, the second is $\pi/4$, and the third is $-\pi/4$. Assume that `update_state` method has been called. The result is shown in Figure 3. Red, green, and blue lines correspond to X, Y, and Z respectively.

```
draw_tree(rbt,[q1=%pi/6,q2=%pi/4,q3=-%pi/4]);
```

4 Rotations

The `manipulator` package contains methods for transformation between different object orientation descriptions. These functions are located in the *rotations.mac* file. The main form of the orientation description in this package is the 3×3 rotation matrix. It can be used in both numerical and symbolical forms. Other parameterization methods, such as Euler angles, quaternions, or pairs of axis and angles can be found from the matrix representation numerically only.

4.0.1 Example: rotations

```
/* get some matrix */
R : Rx(random(1.0)) . Ry(random(1.0)) . Rz(random(1.0));
/* Euler angles in form Y-Z-Y, 2 solutions */
[v1,v2] : R2Euler(R_YZY, R);
/* axis - angle, also 2 solutions */
[v1,v2] : R2AA(R);
/* quaternion */
qu : R2Qu(R);
```

5 Export

The obtained result (matrix, list or variable) can be saved in file if form of **Matlab** function or **C** expression using functions from the *save_as.mac* file. After the function call in your directory will appear the file with name `get_`*varname* and the extension *c* or *m*.

5.0.1 Example: saving

Assume that you got matrix *foo* that you would like to save.

```
/* save as a Matlab function 'get_Foo'*/
save_as_matlab("Foo", foo);
/* save as a C array with name 'Foo' */
save_as_c("Foo", foo);
```

6 Inertial parameters

There are some auxiliary functions for calculation the inertia parameters of simple bodies, defined in the *inertia.mac* file. They allow finding elements of the inertia tensor, total mass, and center of gravity location based on geometric parameters and the material density. Only two types of robot links are assumed here: cylinders and cuboids, and the parameters are calculated for the edge with lower coordinate.

6.0.1 Example: inertial parameters

```
/* cylindrical link along Y axis, length 0.5m */
/* aluminium, radius exter 0.05m, inter 0.03m */
L : inertia_cylinder_y(0.5, 0.05, 0.03, 2700);
/* full beam in X, length 0.3m, size 0.03m, aluminium */
L : inertia_beam_x(0.3, 0.03, 0, 2700);
```

7 List of functions

7.1 manipulator.mac

- `add_child(v1,v2)` - make tree by connecting the two given elements (equal to "v1 ++ v2")
- `by_id(n)` - find object by its identifier n
- `by_name(obj,q)` - find joint using the position variable name
- `Cross(v1,v2)` - cross-product of two vectors
- `draw_tree(obj,qs)` - plot the robot structure for the given joint angles
- `find_base(obj)` - find the first element of the tree
- `from_dh(lst,vars)` - define the serial structure using list of DH parameters in form [a, α, d, θ]
- `from_list(lst,vars)` - define the serial structure using list of transformation functions and list of variables
- `from_urdf(name)` - make the robot using URDF file
- `get_axis(jnt)` - get axis of the joint transformation
- `get_joints(obj)` - get the list of movable joints in the tree
- `get_pos(jnt)` - get position vector
- `get_q(jnt)` - get position variable name
- `get_dq(jnt)` - get velocity variable name
- `get_ddq(jnt)` - get acceleration variable name
- `get_rot(jnt)` - get rotation matrix
- `get_torque(jnt)` - get joint torque value
- `Homo(fn)` - get equal homogeneous transformation matrix for the given function

- `Jacobian(jnt)` - find the Jacobian matrix from base to the given joint
- `Joint(fn,rot,pos)` - make joint structure using function of transformation fn, rotation matrix *rot* and deflection *pos* with respect to the previous joint
- `jointp(obj)` - check if the object is joint
- `Link(par)` - make link structure using the given inertial parameters *par* = $[I_{xx}, I_{xy}, I_{xz}, I_{yy}, I_{yz}, I_{zz}, r_x, r_y, r_z, m]$
- `linkp(obj)` - check if the object is link
- `M_C_G_J(obj)` - get dynamic matrices
- `movablep(jnt)` - check if the object is movable
- `Pos(x,y,z)` - get vector of arguments
- `print_tree(obj)` - print the tree structure
- `prismaticp(jnt)` - check if the joint is prismatic
- `revolutep(jnt)` - check if the joint is revolute
- `rotx(q)` - transformation function for rotation in X
- `roty(q)` - transformation function for rotation in Y
- `rotz(q)` - transformation function for rotation in Z
- `tranx(q)` - transformation function for translation in X
- `trany(q)` - transformation function for translation in Y
- `tranz(q)` - transformation function for translation in Z
- `update_dynamics(obj)` - find the dynamic equations for all joints
- `update_state(obj)` - find kinematic equations for all joints

7.2 rotations.mac

- `Qu_AA(k,q)` - get quaternion $[w, x, y, z]$ from the axis $k = [x, y, z]$ and angle q
- `Qu2AA(v)` - transform quaternion $v = [w, x, y, z]$ into the angle and axis form
- `R_AA(k,q)` - get rotation matrix from the axis $k = [x, y, z]$ and angle q
- `R_Qu(v)` - get rotation matrix from the quaternion elements $v = [w, x, y, z]$
- `R_RPW(r,p,w)` - get rotation matrix from the roll-pitch-yaw angles

- `R_XYZ(a,b,c)` - get rotation matrix from Euler angles a,b,c for X-Y-Z axes
- `R_XYX(a,b,c)` - get rotation matrix from Euler angles a,b,c for X-Y-X axes
- `R_XZX(a,b,c)` - get rotation matrix from Euler angles a,b,c for X-Z-X axes
- `R_XZY(a,b,c)` - get rotation matrix from Euler angles a,b,c for X-Z-Y axes
- `R_YXY(a,b,c)` - get rotation matrix from Euler angles a,b,c for Y-X-Y axes
- `R_YXZ(a,b,c)` - get rotation matrix from Euler angles a,b,c for Y-X-Z axes
- `R_YZX(a,b,c)` - get rotation matrix from Euler angles a,b,c for Y-Z-X axes
- `R_YZY(a,b,c)` - get rotation matrix from Euler angles a,b,c for Y-Z-Y axes
- `R_ZXY(a,b,c)` - get rotation matrix from Euler angles a,b,c for Z-X-Y axes
- `R_ZXZ(a,b,c)` - get rotation matrix from Euler angles a,b,c for Z-X-Z axes
- `R_ZYX(a,b,c)` - get rotation matrix from Euler angles a,b,c for Z-Y-X axes
- `R_ZYZ(a,b,c)` - get rotation matrix from Euler angles a,b,c for Z-Y-Z axes
- `R2AA(m)` - find angle and axis for the given matrix m
- `R2Euler(fn,m)` - find Euler angles for the given rotation type (function fn) and matrix m
- `R2Qu(m)` - find equivalent quaternion
- `Rx(q)` - rotation matrix for X axis
- `Ry(q)` - rotation matrix for Y axis
- `Rz(q)` - rotation matrix for Z axis

7.3 save_as.mac

- `save_as_c(nm,var)` - save variable to the file in C format
- `save_as_c_opt(nm,var)` - apply optimization and save result to the file in C format
- `save_as_matlab(nm,var)` - save variable into the Matlab function
- `save_as_matlab_opt(nm,var)` - apply optimization and save the Matlab function

7.4 inertia.mac

- `inertia_beam_x(L,wo,wi,rho)` - find inertial parameters for the cuboid of length L with external size wo, internal size wi and density rho, along X axis
- `inertia_beam_y(L,wo,wi,rho)` - find inertial parameters for the cuboid of length L with external size wo, internal size wi and density rho, along Y axis
- `inertia_beam_z(L,wo,wi,rho)` - find inertial parameters for the cuboid of length L with external size wo, internal size wi and density rho, along Z axis
- `inertia_cylinder_x(L,ro,ri,rho)` - find inertial parameters for the cylinder of length L with external radius ro, internal radius ri and density rho, along X axis
- `inertia_cylinder_y(L,ro,ri,rho)` - find inertial parameters for the cylinder of length L with external radius ro, internal radius ri and density rho, along Y axis
- `inertia_cylinder_z(L,ro,ri,rho)` - find inertial parameters for the cylinder of length L with external radius ro, internal radius ri and density rho, along Z axis
- `inertia_dist(I,vec,m)` - find new inertia matrix using the initial value I, vector of displacement vec and mass m
- `inertia_rot(I,R)` - find new inertia matrix using the initial value I and applied rotation matrix R
- `inertia_to_list(I,v,m)` - combine the inertia tensor I, center of mass vector v and mass m into the single list

7.5 fromxml.mac

- `xml_find(lst,tag)` - find node with the given tag in the list
- `xml_property(node,nm)` - get attribute value for the given node and attribute name
- `xml_properties(node)` - get the list of attributes
- `xml_read(name)` - read and parse XML file
- `xml_type(node)` - get XML tag for the given node
- `xml_value(node)` - get element of the node
- `urdf_read(name)` - parse URDF file and sort its components