MSc Course Modelling and Simulation of Mectronic Systems - Francesco Biral - Enrico Bertolazzi

Summary

Author: Simone Zamboni - 2019

1: Basics

Remember: we use 3d tools to work in 2d, everything we do is in 2d, the position values in the Z axis is always 0 and rotations are only in the Z axis.
If you need help with a command simply write Describe(command);

1.1 Setup Maple

At the start of every file we need to call restart:

```
> restart:
```

We then need to call the libraries that we need.

Standard libraries that we usually use: plots and Linear Algebra

```
> with(plots):
  with(LinearAlgebra):
```

Special libraries that we use: MBSymba_r6

```
> with(MBSymba_r6):
```

1.2 Reference systems

We usually define a reference system for every body plus auxiliary reference systems when needed.
The always exist the **ground** reference system, wich is attatched to the ground, is an inertial reference system and it is the reference system to wich everything is expressed in.
Every reference system describes the transformation that applied to a point of the body returns that point in the global reference frame: P10 := RF1.P11
There are three ways to define reference systems, in the global, recursive or natural approach:

1.2.1 Global Approach

In the global approach every body its defined by its own three degrees of freedom (that are functions of time).

If we have two bodies, for example:

```
> RF1 := translate(x10(t),y10(t),0).rotate('Z',theta1(t));
```

$$RF1 := \begin{bmatrix} \cos(\theta 1(t)) & -\sin(\theta 1(t)) & 0 & x10(t) \\ \sin(\theta 1(t)) & \cos(\theta 1(t)) & 0 & y10(t) \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \textbf{(1.2.1.1)}$$

```
> RF2 := translate(x20(t),y20(t),0).rotate('Z',theta2(t));
```

$$RF2 := \begin{bmatrix} \cos(\theta 2(t)) & -\sin(\theta 2(t)) & 0 & x20(t) \\ \sin(\theta 2(t)) & \cos(\theta 2(t)) & 0 & y20(t) \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \textbf{(1.2.1.2)}$$

In this case we clearly have three times the number of bodies degrees of freedom (6 in this case).

1.2.2 Recursive approach

In the recursive approach only the first body has its own three degrees of freedom, the other bodies are connected to one another with one coordinare that represent the displacement between the two bodies. For example let's say that we have three bodies, the first is free, the second is connected to the first with a revolute joint and the third is connected to the second with a prismatic joint:

```
> RF1 := translate(x10(t),y10(t),0).rotate('Z',theta1(t));
```

$$RF1 := \begin{bmatrix} \cos(\theta 1(t)) & -\sin(\theta 1(t)) & 0 & x10(t) \\ \sin(\theta 1(t)) & \cos(\theta 1(t)) & 0 & y10(t) \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \textbf{(1.2.2.1)}$$

```
> RF2 := RF1.rotate('Z',theta2(t));
```

$$\begin{aligned} RF2 := [\,&[\cos(\theta 1(t))\cos(\theta 2(t)) - \sin(\theta 1(t))\sin(\theta 2(t)), -\cos(\theta 1(t))\sin(\theta 2(t)) \\ &- \sin(\theta 1(t))\cos(\theta 2(t)), 0, x10(t)\,], \\ &[\sin(\theta 1(t))\cos(\theta 2(t)) + \cos(\theta 1(t))\sin(\theta 2(t)), \cos(\theta 1(t))\cos(\theta 2(t)) \\ &- \sin(\theta 1(t))\sin(\theta 2(t)), 0, y10(t)\,], \\ &[0, 0, 1, 0\,], \\ &[0, 0, 0, 1\,]\,] \end{aligned} \quad \textbf{(1.2.2.2)}$$

```
> RF3 := RF2.translate(s(t),0,0);
```

$$\begin{aligned} RF3 := [\,&[\cos(\theta 1(t))\cos(\theta 2(t)) - \sin(\theta 1(t))\sin(\theta 2(t)), -\cos(\theta 1(t))\sin(\theta 2(t)) \\ &- \sin(\theta 1(t))\cos(\theta 2(t)), 0, (\cos(\theta 1(t))\cos(\theta 2(t)) - \sin(\theta 1(t))\sin(\theta 2(t)))s(t) \\ &+ x10(t)\,], \\ &[\sin(\theta 1(t))\cos(\theta 2(t)) + \cos(\theta 1(t))\sin(\theta 2(t)), \cos(\theta 1(t))\cos(\theta 2(t)) \\ &- \sin(\theta 1(t))\sin(\theta 2(t)), 0, (\sin(\theta 1(t))\cos(\theta 2(t)) + \cos(\theta 1(t))\sin(\theta 2(t)))s(t) \\ &+ y10(t)\,], \\ &[0, 0, 1, 0\,], \\ &[0, 0, 0, 1\,]\,] \end{aligned} \quad \textbf{(1.2.2.3)}$$

In this case we have three bodies but only five degrees of freedom.

1.2.3 Natural Coordinates approach

In the natural coordinate approach we define a body with a point and a direction. We use two coordinates for the point (x and y) and two for the direction, but a body can be defined by only three coordinates, so four are redundant and therefore we will have to put a constraint equation for every body.

The two coordinate for the direction are the x and y components of the i vector in the rotation matrix:

```
> RFnatural1 := <<u1__x(t),u1__y(t),0,0>|<-u1__y(t),u1__x(t),0,0>|<0,0,1,0>|<x1(t),y1(t),0,1>>;
```

$$RFnatural1 := \begin{bmatrix} u1_x(t) & -u1_y(t) & 0 & x1(t) \\ u1_y(t) & u1_x(t) & 0 & y1(t) \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \textbf{(1.2.3.1)}$$

In combination with this we have to put the rigid body constraint equations that says that there are not deformation in this body and therefore evecy vector is a unitary vector (so of lenght one):

```
> Phi1_natural := [u1__x(t)^2 + u1__y(t)^2 - 1];
```

$$Phi1\_natural := \left[u1_x(t)^2 + u1_y(t)^2 - 1 \right] \quad \textbf{(1.2.3.2)}$$

We can extract vectors from the reference systems, with the MBSymba command uvec_X(Y or Z):

```
> project(uvec_X(RF1),ground);
```

$$\text{table}\left(\left[frame = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, obj = VECTOR, comps = \begin{bmatrix} \cos(\theta 1(t)) \\ \sin(\theta 1(t)) \\ 0 \\ 0 \end{bmatrix} \right] \right) \quad \textbf{(1.2.1)}$$

1.3 Points

In our representation points are described by four values (in a **coloumn**) : X coordiante, Y coordinate, Z coordinate (always zero because we work in 2d) and the last coordinate wich is a one for points and a zero for everything else: vectors, forces ecc...

1.3.1 Point Naming conventions

In order to avoid getting confused by a large number of points represented in the various reference systems we need a strong naming convention.

The rules are simple: every point stars with P plus the number of the point in the drawing plus the number of reference system in wich the point is expressed:

- P11: refers to point 1 expressed in the reference system of body 1
- P12: refers to point 1 expressed in the reference system of body 2
- P21: refers to point 2 expressed in the reference system of body 1
- P20: refers to point 2 expressed in the ground reference system

1.3.2 Points in Maple

In Maple we can express points as a coloumn vector of four coordinates:

```
> RF1 := translate(x10(t),y10(t),0).rotate('Z',theta1(t)):
> P11 := <1,2,0,1>;
```

$$P11 := \begin{bmatrix} 1 \\ 2 \\ 0 \\ 1 \end{bmatrix} \quad \textbf{(1.3.2.1)}$$

In order to express this point in the ground referece system we have to:

> P10 := RF1.P11;

$$P10 := \begin{bmatrix} \cos(\theta 1(t)) - 2\sin(\theta 1(t)) + x10(t) \\ \sin(\theta 1(t)) + 2\cos(\theta 1(t)) + y10(t) \\ 0 \\ 1 \end{bmatrix} \quad \textbf{(1.3.2.2)}$$

1.3.3 Points in MBSymba

With this library we can create points with the command make_POINT:

> P11 := make_POINT(RF1,1,2,0);

$$P11 := \text{table}\left(\left[frame = \begin{bmatrix} \cos(\theta 1(t)) & -\sin(\theta 1(t)) & 0 & x10(t) \\ \sin(\theta 1(t)) & \cos(\theta 1(t)) & 0 & y10(t) \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, obj = POINT, coords = \begin{bmatrix} 1 \\ 2 \\ 0 \\ 1 \end{bmatrix}\right]\right) \quad \textbf{(1.3.3.1)}$$

Every point has its own four coordinates plus it is of type point plus it hase explicitly stated the reference system in wich it is expressed.

In order to express this point in the ground referece system we have to:

> P10 := project(P11,ground);

$$P10 := \text{table}\left(\left[frame = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, obj = POINT, coords = \begin{bmatrix} \cos(\theta 1(t)) - 2\sin(\theta 1(t)) + x10(t) \\ \sin(\theta 1(t)) + 2\cos(\theta 1(t)) + y10(t) \\ 0 \\ 1 \end{bmatrix}\right]\right) \quad \textbf{(1.3.3.2)}$$

1.4 Vectors

A vector is a direction. A normalize vector is a vector that has lenght one, the dot product of the vector times the vector is one.

Vectors can be applied to points and the columns of the reference systems are normalized vectors.

Vectors have four coordinates, XYZ and the last one is zero, indicating that is not a point.

1.4.1 Vectors in MBSymba

We can create vectors in MBSymba joining two points:

```
> RF1 := translate(x10(t),y10(t),0).rotate('Z',theta1(t)):
  RF2 := translate(x20(t),y20(t),0).rotate('Z',theta2(t)):
> P11 := make_POINT(RF1,1,1,0):
  P12 := make_POINT(RF2,0.5,0.5,0):
  P21 := make_POINT(RF1,0,0,0):
  P22 := make_POINT(RF2,3,2,0):
> vec1 := join_points(P11,P12);
```

$$vec1 := \text{table}\left(\left[\left[frame = \begin{bmatrix} \cos(\theta 1(t)) & -\sin(\theta 1(t)) & 0 & x10(t) \\ \sin(\theta 1(t)) & \cos(\theta 1(t)) & 0 & y10(t) \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, obj = VECTOR, comps = \left[\left[-1 \right.\right.\right. \quad \textbf{(1.4.1.1)}$$

$$+ (0.5\cos(\theta 2(t)) - 0.5\sin(\theta 2(t)) + x20(t) - 1.x10(t))\cos(\theta 1(t)) + (0.5\sin(\theta 2(t))$$
$$+ 0.5\cos(\theta 2(t)) + y20(t) - 1.y10(t))\sin(\theta 1(t))],$$
$$[-1 + (0.5\sin(\theta 2(t)) + 0.5\cos(\theta 2(t)) + y20(t) - 1.y10(t))\cos(\theta 1(t)) + ($$
$$-0.5\cos(\theta 2(t)) + 0.5\sin(\theta 2(t)) - 1.x20(t) + x10(t))\sin(\theta 1(t))],$$
$$[0.],$$
$$[0]]])$$

```
> vec2 := join_points(P21,P22);
```

$$vec2 := \text{table}\left(\left[\left[frame = \begin{bmatrix} \cos(\theta 1(t)) & -\sin(\theta 1(t)) & 0 & x10(t) \\ \sin(\theta 1(t)) & \cos(\theta 1(t)) & 0 & y10(t) \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, obj = VECTOR, comps = \left[\left[(\right.\right.\right. \quad \textbf{(1.4.1.2)}$$

$$-2\sin(\theta 2(t)) - x10(t) + x20(t) + 3\cos(\theta 2(t)))\cos(\theta 1(t)) + 3\sin(\theta 1(t))\left(\sin(\theta 2(t))\right.$$
$$\left. - \frac{y10(t)}{3} + \frac{y20(t)}{3} + \frac{2\cos(\theta 2(t))}{3}\right)\Bigg],$$
$$\left[(3\sin(\theta 2(t)) - y10(t) + y20(t) + 2\cos(\theta 2(t)))\cos(\theta 1(t))\right.$$
$$\left. + 2\sin(\theta 1(t))\left(\sin(\theta 2(t)) + \frac{x10(t)}{2} - \frac{x20(t)}{2} - \frac{3\cos(\theta 2(t))}{2}\right)\right],$$
$$[0],$$
$$[0]]])$$

We can also extract components from the vectors:

```
> comp_X(vec1);
```

$$-1 + (0.5\cos(\theta 2(t)) - 0.5\sin(\theta 2(t)) + x20(t) - 1.x10(t))\cos(\theta 1(t)) + (0.5\sin(\theta 2(t)) \quad \textbf{(1.4.1.3)}$$
$$+ 0.5\cos(\theta 2(t)) + y20(t) - 1.y10(t))\sin(\theta 1(t))$$

1.4.2 Operations with Vectors

We can do various operations with vectors, but the most importat ones are the following:

Dot Product:

v1 = (x1,y1,z1,0), v2=(x2,y2,z2,0)

v1 dot product v2 = x1*x2 + y1*y2 + z1*z2

The result is a number, wich is 0 if the two vectors are **orthogonal**.

In MBSymba:

```
> dot_prod(vec1,vec2);
```

$$\left(-1+\left(0.5\cos\left(\theta 2(t)\right)-0.5\sin\left(\theta 2(t)\right)+x20(t)-1.x10(t)\right)\cos\left(\theta 1(t)\right)+\left(0.5\sin\left(\theta 2(t)\right)+0.5\cos\left(\theta 2(t)\right)+y20(t)-1.y10(t)\right)\sin\left(\theta 1(t)\right)\right)\left(\left(-2\sin\left(\theta 2(t)\right)-x10(t)+x20(t)+3\cos\left(\theta 2(t)\right)\right)\cos\left(\theta 1(t)\right)+3\sin\left(\theta 1(t)\right)\left(\sin\left(\theta 2(t)\right)-\frac{y10(t)}{3}+\frac{y20(t)}{3}+\frac{2\cos\left(\theta 2(t)\right)}{3}\right)\right)+\left(-1+\left(0.5\sin\left(\theta 2(t)\right)+0.5\cos\left(\theta 2(t)\right)+y20(t)-1.y10(t)\right)\cos\left(\theta 1(t)\right)+\left(-0.5\cos\left(\theta 2(t)\right)+0.5\sin\left(\theta 2(t)\right)-1.x20(t)+x10(t)\right)\sin\left(\theta 1(t)\right)\right)\left(\left(3\sin\left(\theta 2(t)\right)-y10(t)+y20(t)+2\cos\left(\theta 2(t)\right)\right)\cos\left(\theta 1(t)\right)+2\sin\left(\theta 1(t)\right)\left(\sin\left(\theta 2(t)\right)+\frac{x10(t)}{2}-\frac{x20(t)}{2}-\frac{3\cos\left(\theta 2(t)\right)}{2}\right)\right) \quad \textbf{(1.4.2.1)}$$

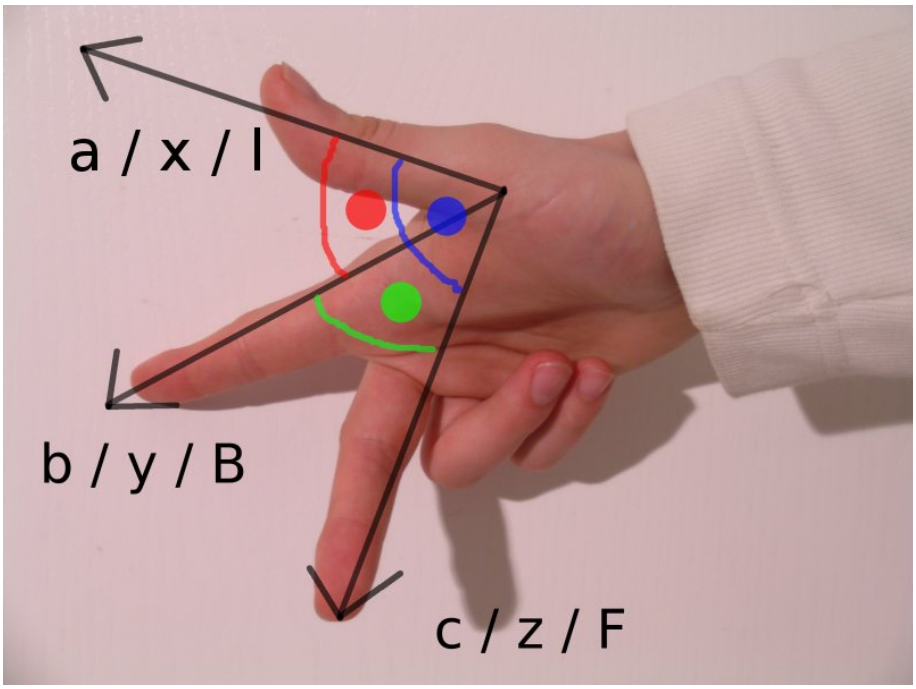
Cross Product

The result of the cross product between two vectors is a vector, with this form:

$$\mathbf{c}=\mathbf{a}\times\mathbf{b}=\begin{vmatrix}\mathbf{i} & \mathbf{j} & \mathbf{k}\\ a_1 & a_2 & a_3\\ b_1 & b_2 & b_3\end{vmatrix}=$$

$$=(a_2b_3-a_3b_2)\mathbf{i}+(a_3b_1-a_1b_3)\mathbf{j}+(a_1b_2-a_2b_1)\mathbf{k}$$

For the resulting vector direction we use the right-hand rule:



The result is the null vector only if the two vectors are in the **same direction**
There is an MBSymba command for that:

```
> Describe( cross_prod);

# Cross product <u> (a VECTOR) by <v> (either a VECTOR or
a FORCE).
cross_prod( u::VECTOR, v::{FORCE, VECTOR}, $ )
```

2: Constraint Equations

We have contraint equations in every connection point between two bodies if we choose the global approach. If we choose the recursice approach we will have constraint equations in connection between two bodies that we can't cover with the recursive approach, therefore we will always have contraint equations when we use the recursive formulation in a closed system.

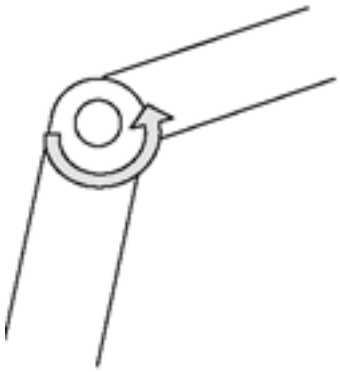
2.1 Types of joits and relative contraint equations

For these example we assume to have some bodies:

```
> RF1 := translate(x10(t),y10(t),0).rotate('Z',theta1(t)):
  RF2 := translate(x20(t),y20(t),0).rotate('Z',theta2(t)):
  RF3 := translate(x30(t),y30(t),0).rotate('Z',theta3(t)):
  RF4 := translate(x40(t),y40(t),0).rotate('Z',theta4(t)):
```

2.1.1 Revolute Joint

A revolutie joint is a joint that permits only a relative rotation between one body and the other, because they have a fixed point in common that cannot move in the two reference frame of the bodies



Revolute Joint

In order to define this contraint we fix that the position of the conjuntion point (that we will call in this case P1) must be the same for the two bodies, therefore the vector between the two points must be equal to zero (expressed in the ground ref. frame).

```
> P11 := make_POINT(RF1,-1,0,0):
  P12 := make_POINT(RF2,1,0,0):
> vec1 := project(join_points(P11,P12),ground);
```

$$vec1 := \text{table}\left(\left[\left[frame = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, obj = VECTOR, comps \right.\right.\right. \tag{2.1.1.1}$$

$$\left.\left.\left. = \begin{bmatrix} \cos(\theta 2(t)) + x20(t) - x10(t) + \cos(\theta 1(t)) \\ \sin(\theta 2(t)) + y20(t) - y10(t) + \sin(\theta 1(t)) \\ 0 \\ 0 \end{bmatrix} \right]\right]\right)$$

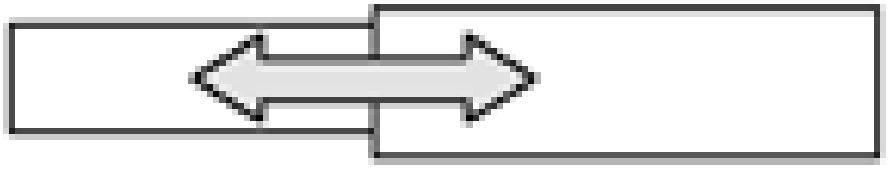
```
> rev_joint_const_eq := [comp_X(vec1),comp_Y(vec1)]: <%>;
```

$$\begin{bmatrix} \cos(\theta 2(t)) + x20(t) - x10(t) + \cos(\theta 1(t)) \\ \sin(\theta 2(t)) + y20(t) - y10(t) + \sin(\theta 1(t)) \end{bmatrix} \tag{2.1.1.2}$$

This joint will produce two contraint equations, because it restrict two degrees of freedom in the second body

2.1.2 Prismatic Joint

A prismatic joint is a joint that permits only translation along an axis. Therefore it restrict two degrees of freedom and it will be composed by two equations.



Prismatic Joint

In order to model this contraint we need to first define:

1. An auxiliary reference frame from the first body (in this case body 2) wich has the X axis on the prismatic joint sliding axis (if the first body reference frame does not have this property)

> RF2aux := RF2.rotate('Z',beta1): # beta1 is the fixed misalignement between the reference system of body 2 and the sliding axis

2. A point in the first body on the sliding reference axis (for example the origin of the auxiliary reference frame that we have just created)

> P22 := origin(RF2aux):

3. An auxiliary reference frame from the second (moving) body (in this case body 3) wich hase the X axis on the prismatic joint sliding axis (if the second body reference frame does not have this property)

> RF3aux := RF3.rotate('Z',beta2): # beta2 is the fixed misalignement between the reference system of body 3 and the sliding axis

4. A point in the second body on the sliding reference axis (for ecample the origin of the auxiliary reference frame that we have just created)

> P33 := origin(RF3aux):

Now the constraint equations are two:

- The Y axis of RF2aux and the X axis of RF3aux (or the contrary) must be perpendiculat: their dot product must be equal to 0

> pris_joint_constr_1 := dot_prod(uvec_Y(RF2aux),uvec_X(RF3aux));

$$pris\_joint\_constr\_1 := (-\cos(\theta 2(t))\sin(\beta 1) - \sin(\theta 2(t))\cos(\beta 1))(\cos(\theta 3(t))\cos(\beta 2) - \sin(\theta 3(t))\sin(\beta 2)) + (\cos(\theta 2(t))\cos(\beta 1) - \sin(\theta 2(t))\sin(\beta 1))(\sin(\theta 3(t))\cos(\beta 2) + \cos(\theta 3(t))\sin(\beta 2)) \quad \textbf{(2.1.2.1)}$$

- The vector from the point P22 to the point P33 must be parallel to the sliding axis, and therefore perpendiculat to the y-axis of RF2aux or to the y-axis of RF3aux

> pris_joint_constr_2 := dot_prod(uvec_Y(RF2aux),join_points(P22,P33));

$$pris\_joint\_constr\_2 := ((-y20(t) + y30(t))\cos(\beta 1) + \sin(\beta 1)(x20(t) - x30(t)))\cos(\theta 2(t)) + \sin(\theta 2(t))((x20(t) - x30(t))\cos(\beta 1) + \sin(\beta 1)(y20(t) - y30(t))) \quad \textbf{(2.1.2.2)}$$

> pris_joint_constr_eq := [pris_joint_constr_1, pris_joint_constr_2]: <%>;

$$\left[\begin{array}{c} [(-\cos(\theta 2(t))\sin(\beta 1) - \sin(\theta 2(t))\cos(\beta 1))(\cos(\theta 3(t))\cos(\beta 2) - \sin(\theta 3(t))\sin(\beta 2)) + (\cos(\theta 2(t))\cos(\beta 1) - \sin(\theta 2(t))\sin(\beta 1))(\sin(\theta 3(t))\cos(\beta 2) + \cos(\theta 3(t))\sin(\beta 2))], \\ [((-y20(t) + y30(t))\cos(\beta 1) + \sin(\beta 1)(x20(t) - x30(t)))\cos(\theta 2(t)) + \sin(\theta 2(t))((x20(t) - x30(t))\cos(\beta 1) + \sin(\beta 1)(y20(t) - y30(t)))] \end{array}\right] \quad \textbf{(2.1.2.3)}$$

2.1.3 Rod Joint

The rod joint is a joint connecting two bodies that imposes that the relative distance between the two points of the bodies needs to remain constant.
This joint therefore eliminate only one degrees of freedom, and therefore will have only one contraint equation:
v45 = P43 - P54 ==> ||v45|| = L^2

```
> P43 := make_POINT(RF3,1,0,0):
  P54 := make_POINT(RF4,2,3,0):
> v45 := project(join_points(P43,P54),ground);
```

$$v45 := \text{table}\left(\left[\left[frame = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}\right], obj = VECTOR, comps\right.\right. \tag{2.1.3.1}$$

$$\left.\left. = \begin{bmatrix} 2\cos(\theta4(t)) - 3\sin(\theta4(t)) + x40(t) - x30(t) - \cos(\theta3(t)) \\ 3\cos(\theta4(t)) + 2\sin(\theta4(t)) + y40(t) - y30(t) - \sin(\theta3(t)) \\ 0 \\ 0 \end{bmatrix}\right]\right)$$

```
> rod_joint_constr_eq := [ dot_prod(v45,v45)-L^2]:  <%>;
  # or rod_joint_constr_eq := [sqrt( dot_prod(v45,v45))-L]:  <%>;
```

$$\left[(2\cos(\theta4(t)) - 3\sin(\theta4(t)) + x40(t) - x30(t) - \cos(\theta3(t)))^2 + (3\cos(\theta4(t)) \right. \tag{2.1.3.2}$$

$$\left. + 2\sin(\theta4(t)) + y40(t) - y30(t) - \sin(\theta3(t)))^2 - L^2 \right] ,$$

Also a rope can be modeled as a rod joint in some cases.

2.1.3.4 Cam Joint

A cam joint is a type of joint where there is a point sliding on an axis. Is basically a prismatic joint but instead of being on the entire body is only on that point. Being less restrictive that a normal prismatic joint it only restricts a degree of freedom and therefore it has only one constraint equation.
In order to define the constraint equation of the cam joint we need three things:
1. The reference frame where the sliding is on the x-axis (RFa)
2. A point on the non-sliding body on the sliding axis (Pa)
3. The sliding point (Pb)

The constraint is that the vector joining the two points whould be parallel to the sliding axis. We use the dot product with the vector tangent to the sliding axis for the parallel condition

```
> # cam_joint_constr := dot_prod( uvec_Y(RFa, project( join_points(Pa,Pb) ,ground) );
```

2.1.5 Natural coordinates constraint

As we stated in chapter 1 for every body defined with natural coordinate we will have a constraint equation limiting the values of u__x and u__y:

```
> Phi1_natural := [u1__x(t)^2 + u1__y(t)^2 - 1];
```

$$Phi1\_natural := \left[u1_x(t)^2 + u1_y(t)^2 - 1 \right] \tag{2.1.5.1}$$

Then we can compact all the constraint equations in a single variable, Phi, in order to have them all in one place:

```
> Phi := rev_joint_const_eq union pris_joint_constr_eq union rod_joint_constr_eq: <%>;
```

$$\left[\left[\cos(\theta 2(t)) + x20(t) - x10(t) + \cos(\theta 1(t))\right],\right. \tag{2.1.1}$$
$$\left[\sin(\theta 2(t)) + y20(t) - y10(t) + \sin(\theta 1(t))\right],$$
$$\left[\left(-\cos(\theta 2(t))\sin(\beta 1) - \sin(\theta 2(t))\cos(\beta 1)\right)\left(\cos(\theta 3(t))\cos(\beta 2) - \sin(\theta 3(t))\sin(\beta 2)\right)\right.$$
$$\left.+\left(\cos(\theta 2(t))\cos(\beta 1) - \sin(\theta 2(t))\sin(\beta 1)\right)\left(\sin(\theta 3(t))\cos(\beta 2) + \cos(\theta 3(t))\sin(\beta 2)\right),\right],$$
$$\left[\left(\left(-y20(t) + y30(t)\right)\cos(\beta 1) + \sin(\beta 1)\left(x20(t) - x30(t)\right)\right)\cos(\theta 2(t))\right.$$
$$\left.+\sin(\theta 2(t))\left(\left(x20(t) - x30(t)\right)\cos(\beta 1) + \sin(\beta 1)\left(y20(t) - y30(t)\right)\right)\right],$$
$$\left[\left(2\cos(\theta 4(t)) - 3\sin(\theta 4(t)) + x40(t) - x30(t) - \cos(\theta 3(t))\right)^2 + \left(3\cos(\theta 4(t))\right.\right.$$
$$\left.\left.+2\sin(\theta 4(t)) + y40(t) - y30(t) - \sin(\theta 3(t))\right)^2 - L^2\right] \quad ,$$

And we can easily see the number of constraint equations (5 in this case), also using the command nops:

```
> nops(Phi);
```

$$5 \tag{2.1.2}$$

2.2 Indipendent and dependent coordinates

Indipendent coordinate are in equal numbers to the degrees of freedom of the system. All other coordinates are dependent coordiantes, because we can identify a function that given the indipendent coordinates gives us the dependent one:
qD = F(qI)
Therefore we only need that function and the indipendent coordinates in order to fully describe the system. How we identify the indipendent coordinates depends in part if we used global formulation or recursive formulation, but the idea is the same.

We start gathering all our coordinates:

- If we used a pure global approach the coordinates will be three (x position, y position and rotation) times the number of bodies that we have;
- If we used a pure recursice approach the coordinates will be the coordinates describing the relative movement of the bodies between each other;
- If we used an hybrid formulation (or the natural approach) we collect all the coordinates that we used, bot to describe the relative motion between bodies and the one we used to describe the bodies.

Now that we have all the coordinates we need to understand how many of them are indipendent:
N° indipendent coordinates (qI) = N° generalized coordinates (q) - N° constraint equations (Phi)
So if we have 6 coordinates but 5 contraint equations we will have 1 indipendent coordinate, and 1 DOF. The number of generalized coordinates is computed by us, but we can derive the number of constrait equations using the nops() command.

We have now to choose (wisely) the coordinate between the generalized coordinate that we want to use as indipendent. This coordinate should respect our ability to influence the system, if at a joint we have a motor the coordinate of that joint should be indipendent, because we can directly control them.

After doing that we will have two variable:
qI := [list of indipendent coordinates]
qD := [list of dipendent coordinates]

2.3 Analitically solve the constraint equations

Once we have the constraint equations and we identified the dependent and indipendent coordinates, we can try to solve the constraint equations analitically:

```
> # sol_kine_all := simplify( solve(Phi,qD), trig): nops(sol_kine_all);
> # sol_1 := sol_kine_all[1]:
```

Now if there are no RootOfs or other strange stuff we have in sol_kine_all all the differnt solutions for the constraints. We can check the number of solutions that we have with the nops() command, and select the onel that we want, in this case the first.
If we have trouble in this step we can use the options of the solve() command:
• **allsolutions** = true : returns all the solutions of the problem
• **explicit** = true : try to solve the RootOfs
If, even with these options, we are not yet able to obtain an analytical solution we can call the **allvalues**() command on the expression, like allvalues(Phi), but we should avoid arriving at this stage.

In sol_1 we will have all the dependent coordinates expressed as a function of the indipendent coordinate, our qD = F(qI).

2.3.1 Find the correct kinematic solution

When solving the constraint equations is possible that we will end up with multiple solutions, nops (sol_kine_all) is greater than one, and we have to choose the correct way to assembre the mechanism. A wise choice could be to evaluate all the various solutions with some easily identifiable indipendent coordiante and see the values of the indipendent coordinate for every solution. Then we choose the kinematic solution that produces the values of the dependent coordinate that we expect.

```
> # subs(data, ics_qI, sol_kine_all);
```

2.4 Numerically solve the constraint equations

It may happen that we have to describe some variables respect to other variables but this is not possible. In this case we must perform a kinematic analysis in a numerical way.
First we solve the system in a way that we know, so to have some initial conditions:

```
> # initials :=evalf(subs(data,x1(t)=-0.13,sol_kine)) union [x1(t)=-0.13];
  # insts := convert(initials,set);
  # insts := subs( x1(t)=x1,theta1(t)=theta1,theta__p(t)=theta__p,theta3(t)=theta3);
```

Then we prepare a variable to store the numerical solutions of the variables and we iterate using fsolve to find the solutions (remember that fsolve doesn't want variables in the form of x(t), he wants x):

```
> # results := Matrix(10,4):
> for i from 1 to 10 do:
    Phi_temp := subs(s(t)=0.5+0.03*i,data,x1(t)=x1,theta1(t)=theta1,theta__p(t)=theta__p,theta3(t)
  =theta3,Phi);
    insts := fsolve(Phi_temp, insts);
    results[i,1] := rhs(insts[1]):
    results[i,2] := rhs(insts[2]):
    results[i,3] := rhs(insts[3]):
    results[i,4] := rhs(insts[4]):
  end do:
Error, invalid input: subs received data, which is not valid
```

for its 2nd argument

In this way in each coloumn of results we will have a indipendent variable.

3: Position and Velocity Analisys

At this point the bodies, the constraints and the coordinates are defined, now it is time to analyze the kinematic of the system.
Let's say for conveniece that we hade defined a variable called data with all the static data and numbers of the system:

```
> data :={L=1,beta1=Pi/2, L2 = 4 };
```

$$data := \left\{ L = 1, L2 = 4, \beta 1 = \frac{\pi}{2} \right\} \quad \textbf{(3.1)}$$

3.1 Initial Position Problem

We want to know the initial position of the system given the data and the values of the indipendent coordinates at the start.
There are two ways of doing that, depending on if we have solved analitically the constraint equations or not.

3.1.1 Initial position in an analytical way

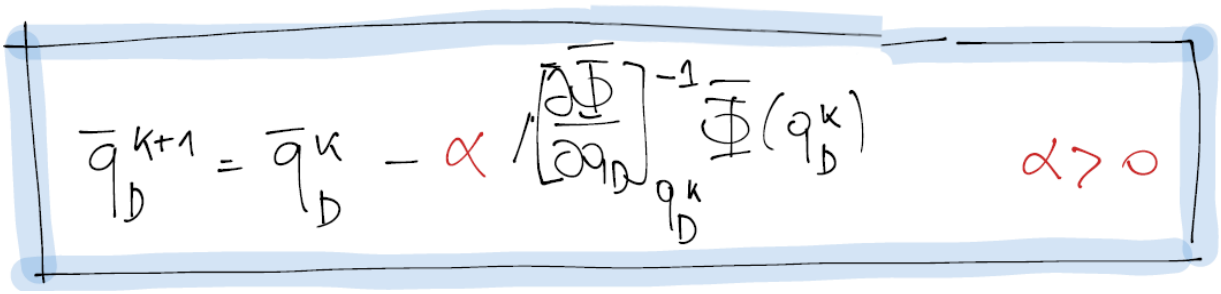
We assume we where able to complete point 2.3 and therefore we have a function that describes the dependent coordinates give the indipendent coordinates, we have qD = F(qI).
This is the easiest case, because we have to simply substitute the data and the values of the indipendent coordinates in that function and we will have the initial values of the dependend coordinates:

```
> # initial_configuration := evalf( subs (data, qI_initial , sol_1)  );
```

This is very easy, but we need to keep in mind that there ca be more than one solution for the function F, solving the constraint equations, and therefore we can have more than one initial position possible, but if we use the method describe in 2.3.1 we shoul have identified the correct kinematic solution and therefore there will be only one initial condition.

3.1.2 Initial position in a numerical way

In this case it is not possible (because is too computationally expensive or it is forbidden) to solve the constrants in an analytical way, and therefore to obtain the initial position we have to use a numerical method.
The two numerical methods to use are minimization functions (available on Mathlab) or the Newton-based methods (the method used by Maple).
In the Newton based method we start guessing and then continuosly improving our guess at every iteration k moving in the direction of the gradient, following the law:



This method is the one used by fsolve() in Maple, where we can specify both the intial guesses and the ranges:

```
fsolve(e::{set},q::{set('=',symbol)},r::{..},opts)
```

e:: list / set of constraint equations

q:: list/set of unknowns

↓

can be a set of variables: $\{q_1, q_2, q_3, \dots q_n\}$

or a set of equalities: (specifies INITIAL GUESS) $\{q_1 = q_{10}, q_2 = q_{20}, \dots, q_n = q_{n0}\}$

→ numerical parameters

$\bar{r}$ → set of ranges: $\bar{r} = \{q_1 = q_1^{min} .. q_1^{max}, \dots, q_n = q_n^{min} .. q_n^{max}\}$

An example of calling this function would be:

```
> Describe (fsolve);
  # fsolve( Phi, qI );

module fsolve( eqs, vars )

    fsolveT::symbol = fsolve:-fsolveT

    getsingul( e )

    PrintTable( id )

    refine2( app, p, p1, lm )


> # fsolve( subs(data, a(t)=1,Phi), {b(t)=1, c(t) =2, d(t)=3};
```

3.2 Velocity Analisys

We now move to the velocity analisys. There are various ways to do that, depending also if we have the

analytical description of the dependent coordinates or not.

3.2.1 Velocity Analisys 1: differentiate the constrain equations

This method is possible only if it is possible to have the analytical solution for the dependent coordinate.

In this case we can simply differentiate the kinematic solution that we have found, plug in the data and the kinematic solution again and we have the velocity of each dependent coordinate:

```
> # sol_1_diff := rhs( diff(sol_1,t) );
> # velocities := subs(data,sol_1,sol_1_diff);
```

Or we can do this procedure starting directly from the constraint equations if we don't have the kinematic solution (with the kinemaric solutio is easier): we derive them, we solve for the derivative of the dependent coordinate and then we put the data and the kinematic solution in:

```
> # diff_Phi := diff(Phi,t);
> # diff_Phi_sol := simplify(solve(diff_Phi ,diff(qD,t) ) , trig );
> # velocity_sol_1 := diff_Phi_sol[1];
> # velocities := subs(data,sol_1, velocity_sol_1);
```

3.2.2 Velocity Analisys 2: the Tau Matrix

In this case we will extract the velocities from the Jacobian of the constraint equations, but still with the need of a analitycal kinematic solution.
We start by computing the Jacobians of the constraints:

```
> # JPhi_D,a := GenerateMatrix(map( diff, Phi, t) ,diff( qD,t ));
> # JPhi_I,a := GenerateMatrix(map(diff,Phi,t),diff(qI,t));
```

We then create the Tau vector, wich is the inverse of JPhi_qD times JPhi_qI:

```
> # tau := - simplify( MatrixInverse( JPhi_D ) . JPhi_I ) ;
```

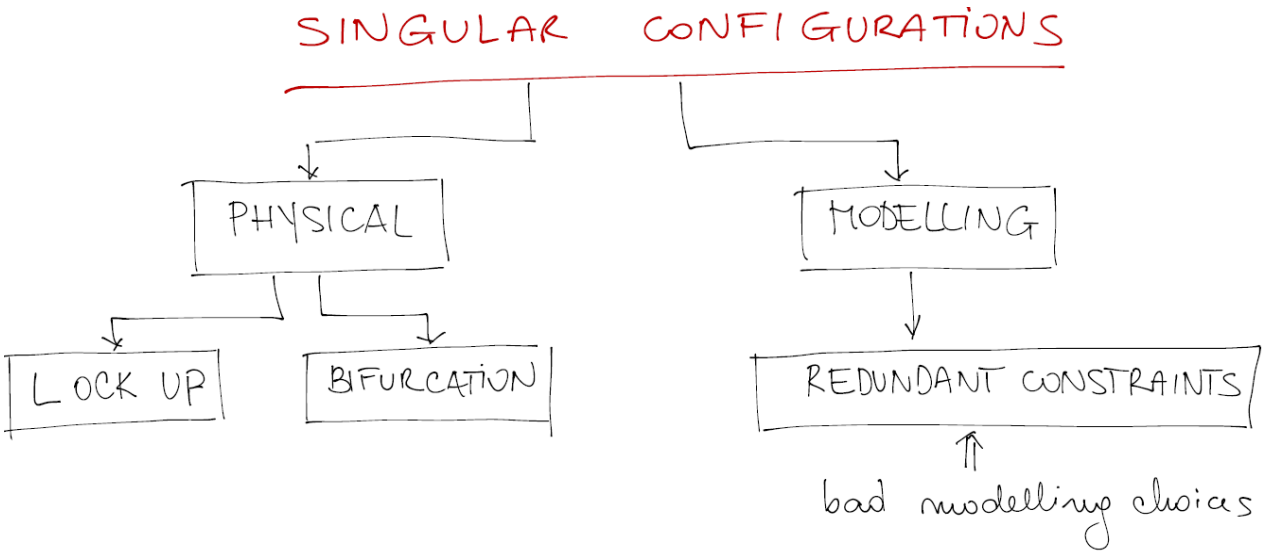
From the Tau vector we can substitute the kinematic solution and the data and we will have the velocities:

```
> # velocities := subs(data,sol_1,tau) . <diff(qI,t)>;
```

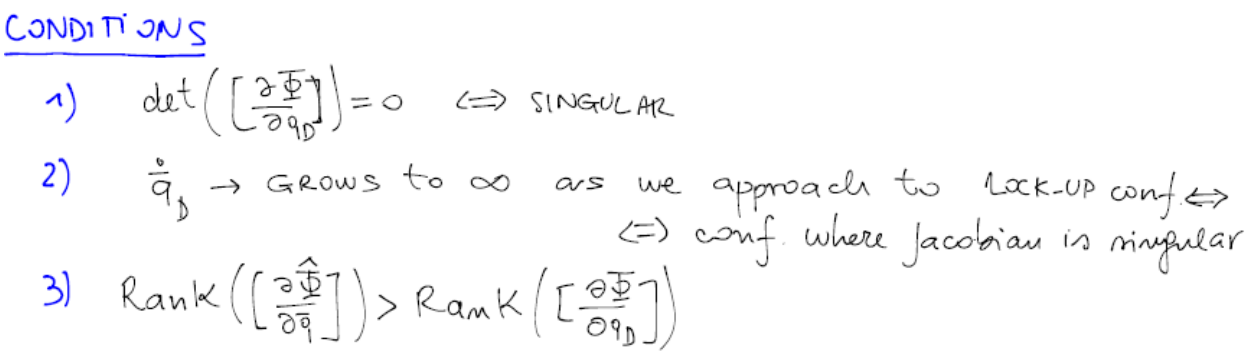
Note that this process is impossible when is impossible to find the inverse of JPhi_dD, and that is impossible when we are in a singular configuration.
This is easier and better method.

3.3 Singular configurations

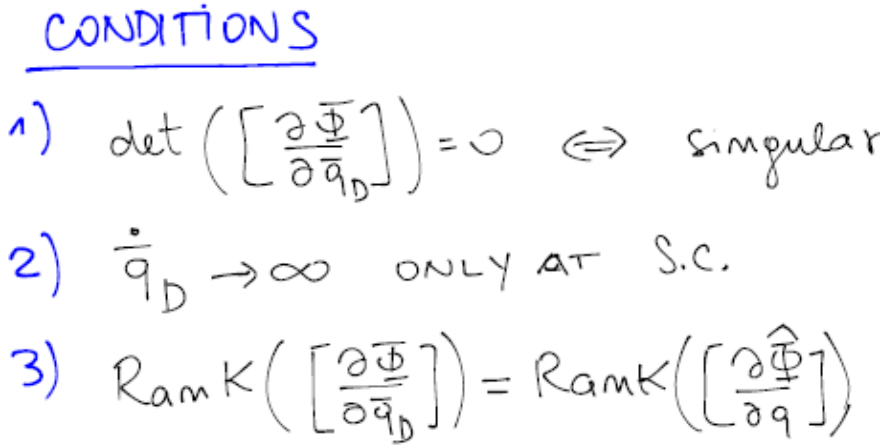
Singular configuration can happen for three reasons:



The conditions for the Lock Up:



The conditions for the Bifurcation:



As we can see, a singulars configurations happen when the determinant of the Jacobian is zero. We can find when the determinant is zero with these commands, solving the determinant of the Jacobian__qD in respect to all the variables:

```
> # JPhi_D,a := GenerateMatrix(map( diff, Phi, t) ,diff( qD,t )); # Jacobian computation
> # JPhi_I,a := GenerateMatrix(map(diff,Phi,t),diff(qI,t));
> # JPhi_D_det := simplify(Determinant(J_Phi_D));      # find the determinant
> # sing_conf := solve(JPhi_D_det, qD union qI);      # find out when the determinant is zero: the
  singular configurations
```

Now we know when the determinant is zero, and therefore the various singular configurations. It's not true that even if the determinant can be zero there are singular configurations in our system, because maybe the structure of our system prevents that. In order to check if there are really singular configuration in our system we have to solve the constraints once again:

```
> # sing_constraints := subs(sing_conf[1],Phi);
> # solve(sing_constraints,qDtot); nops(%);
```

We could also have united the determinant and the constraint equations and solve everything together to find if the singulars configurations are possible in this mechanism:

```
> # big_system := [ JPhi_D_det ] union JPhi_D;
> # sing_confs := simplify( solve( big_system, qvars ) );
```

This last method is easier and more robust.

If we obtain something from the last computation, we will have singular configurations in this system.

4: Dynamics

The aim of the dynamic analysis is to obtain the equation of motions of the bodies, and therefore of the coordinate that describe them.
Starting let's analyze how MBSymba deals with the dynamic representations of bodies and forces.

4.1 Bodies and Forces in MBSymba

4.1.1 Bodies in MBSymba

To define bodies in MBSymba we use the command make_BODY, wich has two different ways of being recalled:

```
> # b := make_BODY( RF, m, Ixx, Iyy, Izz );
```

Where:

- RF is the reference system that has **origin in the center of mass** of the body;
- m is the mass of the body;
- Ixx, Iyy, Izz is the inertia of the body. We work in 2d so the body can only have inertia in the Izz component.

The other way is:

```
> # b := make_BODY( G, m, Ixx, Iyy, Izz );
```

Where the only difference as before is that G is the point describing the center of mass of the body, usually in the referece system of the body

4.1.2 Forces and Torques in MBSymba

To define a force in MBSymba we use the make_FORCE command:

```
> # f := make_FORCE( RF, Fx, Fy, Fz, P, B1, B2);
```

Where:

- RF is the reference system where the components of the forces are expressed;
- Fx,Fy,Fz are the components of the force (we will only have Fx and Fy in 2d);
- P is the point where the force is applied;
- B1 is the body where the force is applied;
- B2 is optional. Is the body where the same force is applied with negative sign, where there will be basically the reaction force. This is very usefoul in the Newton approach.

There is also the option of substituting the reference frame and the components with a vector.

To define torques is MBSymba we use the make_TORQUE command:

> # t:= make_TORQUE(RF, Tx, Ty, Tz, B1, B2);

Where:

- RF is the reference system where the components of the force are expressed;
- Tx,Ty,Tz are the components of the torque (in 2d we will have only Tz);
- B1 is the body where the torque is applied;
- B2 is optional. It is the body where the torque reacts.

4.1.3 Spring force

The force of a spring depends on its elongation over the spring natural lenght and the stiffness of the string. Usually the spring natural lenght is called Ls0 and the stiffness Ks.
The spring will always try to return to its natural length: therefore is the sping is compressed is will push s foreward, while if it is stretched it will push inwards.
Let's say we have a sping pushing on the x-axis a body, with the following point of contact:

> # PC := make_POINT(ground, Ls0 + s(t) ,0 ,0);

The variable s(t) represents the sping elungation, and the force that the spring will apply is:

> # spring_f := make_FORCE(ground, -s(t) * Ks, 0, 0, PC , b1);

A spring can also have dumping. The dumping depends on the spring speed and it slows down the spring, with a constant Cs describing the how much the damping is strong. In the previous case we needed to specify the sign of the force, but in this case it depends only on the direction of the velocity.
An example of spring with dumping force would be :

> # spring_f := make_FORCE(ground, -s(t) * Ks - Cs*diff(s(t),t) , 0, 0, PC , b1);

Remember: normal spring force and dumping should have the same direction!

4.2 The Newton-Euler Approach

With the newton approach we obtain the equation of motion using two basic laws:

$$m\,\bar{a}_G = \sum_{i=1}^{n} \bar{F}_i^e$$

=> the mass times the accelleration (in the three directions) is equal to the sum of every external force applied to that body. This is the Newton Equation.

$$[I_G]\dot{\bar{\omega}} + \bar{\omega} \wedge [I_G]\bar{\omega} = \sum_{j=4}^{M} \bar{M}_j^e + \sum_{i=1}^{N} \bar{r}_{GP_i} \wedge \bar{F}_i^e$$

=> The inertia tensor times the angular accelleration (in the three directions) plus the velocity cross

product the inertia tensor times the angular veocity is equal to the sum of every pure momentum applied to the body plus the sum of every external force applied to the body in cross product with its application point. This is the Euler Equation.

In 3d we have three linear accellerations and three angular accellerations, and therefore we will have three Newton equations and three Euler equations for each body.
We usually work in 2d, therefore we will have two Newton equations (for the x and y linear accellerations) and one Euler equation (for the z angular accelleration) for each body.
These are differential equations, and solving them we will have the equations describing the position, and therefore the velocity and accelleration, of the coordinates describing every body.

Before that we have to first identify correctly all the forces applied to every body.

Step1: Define external forces
Create all the pure external forces, specifying on wich body the are applied. This is based on the particularity of your problem. Alro remember friction forces if present.

Step2: Define gravity forces
Create all the gravify forces. Let's say gravity is on the -y axis, for every body we will have the w force:
> # w1 := make_FORCE(ground, 0, -m1*g, 0, G1, B1);
Instead of G1 we can put origin(RF1), if the origin of RF1 is the center of mass of body1.

Step3: Define constraint forces
Every constraint has its own force, in order to enforce the constraint. Maybe force in this case is not the most exact word to use, every constraint has a component on a force to enforce the constraint. This mean that a force can have two components and therefore enforce two constraint equations. Moreover, we are using the term forces with the meaning both of a force and of a torque, because some constraints are enforced by a torque.

Remember that this part is to do indipendently of the method that we choose: recursive, global or natural.
A force acts on a body, but it could also react on another body, if these two are in contacts in the point of application of the force (or the torque).

Let's start with a **revolute joint constrain**t.
A revolute joint constraint will have a force with two components (because it has two constraint equations and it restricts two degrees of freedom) acting on the point of the revolute joint on both bodies, on the first actiong as an action force and on the second acting as a reaction force. For example:
> # RV1_force := make_FORCE(ground, Rvf1x(t) , Rvf1y(t), 0, P1, B1);
This force is the force create by a revolute joint constraint. It has two components, one on the x axis and one on the y axis, expressed in the ground reference frame. It is applied on the point P1, that is the point where is the revolute joint, and this joint connect the body B1, where the force act as a normal force, and the body B2, there the force act as a reaction force.

Now we describe the forces generated by a **prismatic joint**.
A prismatic joint constraint two degrees of freedom, and it will create a force with one component and a torque (because it has two constraint equations and it restricts two degrees of freedom).

There can also be **cam joints.**
In this case we have only one force that is perpendicular to the sliding axis of the point
> # Cjf := make_FORCE (RFsliding_on_x, 0, cjf(t), 0, P1, B1);

Now the **rod joint**.

The rod joint constraint force will have only one component because the rod joint restricts only one degree of freedom. Let's assume that the rod joint is connecting the point P11(on body 1) and P22 (on body 2), the rod joint force will act in the direction of the vector v12. Moreover, this vector need to be normalize in a unit vector, and therefore we need to divide every component of the vector for the lenght of the rod (L). To simulate this constraint in MBSymba we need to define two force, because the first is acting on P11 and the body 1, and the other is acting on P22 on body 2 in the opposite direction.

```
> # v12 := project( join_points(P11,P22),ground);
> # Rdjf1 := make_FORCE(ground, comp_X(v12) * Rd(t) / L, comp_Y(v12)* Rd(t) / L, 0, P11, b1)
  ;
> # Rdjf2 := make_FORCE(ground, -comp_X(v12) * Rd(t) / L, -comp_Y(v12)* Rd(t) / L, 0, P22,
  b2);
```

Step 4: Create the equations

We first put all the forces in a variable as a list, called in this case fg_mbs:

```
> # fg_mbs := [w1, w2, RV1_force, PS1_force, PS1_torque ];
```

We than call these two commands on evey body (in the euler equations put the origin of the reference system of the body in order to have simpler equations):

```
> # neq_b1 := newton_equations ( [B1] union fg_mbs );
> # eeq_b1 := euler_equations ( [B1] union fg_mbs, origin(RF1) )
```

Then we put everything together:

```
> # eq := comp_X(neq_b1) union comp_Y(neq_b1) union comp_Z(eeq_b1) union comp_X(neq_b2)
  union comp_Y(neq_b2) union comp_Z(eeq_b2);
```

For some applications we can stop here.

Step 5: Create the matrices

We have to define the M, V and Fe matrices. Before doing that we will put all the coordinate variables together and also all the reaction forces together:

```
> # qvars := qI union qD;
> # Reaction_forces := [ Rvf1x(t), Fvf1y(t), Psf(t), Pst(t) ];
```

Then we can start generating the M matrix:

```
> # M,a := GenerateMatrix(eq ,diff ( diff( qvars, t ), t ) );
```

And then the V and Fe matrix from what is remained:

```
> # V, Fe := GenerateMatrix( convert(a,list), Reaction_forces);
```

Step 6: Create the final system

In order to create the final system we have to define a new set of variables, as many as the number of qvars, that will represent the velocities of these, so that we have a normal system with only first order derivatives.

```
> # Vvars := [v1(t),v2(t),v3(t),v4(t),v5(t),v6(t)];
> # first_part := [Vvars[1] = diff(qvars[1],t),Vvars[2] = diff(qvars[2],t),Vvars[3] = diff(qvars[3],t),
  Vvars[4] = diff(qvars[4],t),Vvars[5] = diff(qvars[5],t),Vvars[6] = diff(qvars[6],t)];
```

Then we use the equations of motions found (only if M is invertible, it must be full rank therefore):

```
> # second_part := MatrixInverse(M).(V.<Reaction_forces>+Fe)-<diff(Vvars,t)>;
```

And then we have finally everything together:

```
> # complete_equations := first_part union convert(second_part,list) union Phi;
```

This is a index 3 DAE.

Alternatively we can use the first_order MBSysmba command do downgrade the the **first_order** the

equations of motion:

> # newvars, eq_first := first_order(eq,qvars);

This command will return the new variables and the new equation with the substituted variables.
Therefore the complete equations will be:

> # complete_equations := convert(eq_first,list) union Phi;

4.3 Lagrange

The Lagrange method is a method to find the equations of motion that produces as many equations as the number of coordinate that we use.
In the Lagrange approach we first define the Lagrange function, that is kinetic energy - potential energy:

Lagrangian function:

$$L(\bar{q}) = \underbrace{\frac{1}{2}\dot{\bar{q}}^T [M(\bar{q})]\dot{\bar{q}}}_{=E_K(\bar{q},\dot{\bar{q}})} - E_P(\bar{q})$$

And then we apply the drivative respect qdot, time and q, plus we add the constraint equations to the formula. The constraint equations are at the bottom and in the lagrange function we add them using the lagrange multipliers that multiply the Jacobian of the constraints. All of this is equal to the external forces acting on the system:

$$\frac{d}{dt}\frac{\partial L}{\partial \dot{\bar{q}}} - \frac{\partial L}{\partial \bar{q}} - \left[\frac{\partial \bar{\Phi}}{\partial q}\right]^T \bar{\lambda} = \bar{Q}$$

$$0 = \bar{\Phi}(q)$$

MBSymba already has a lot of functions that we can call that will automatically create the Lagrange system for the multibodies.
Some of these commands are:

> Describe (gravitational_energy);

```
# Compute the gravitational energy of a multibody system by
evaluating the
# force field associated to the <_gravity> acceleration.
```

```
gravitational_energy( bodies::{BODY, list, set} )
```

> Describe (kinetic_energy);

```
# Compute the kinetic energy of a multibody system. Set the
optional flag
# <nat_coords> = true for a constant mass matrix even while
using natural
# (dependent) coordinates.
kinetic_energy( bodies::{BODY, list, set},
nat_coords::boolean := false )
```

Now let's see how in practice we can derive the equations of motion with the Lagrange approach:

Step 1: Define the bodies

Define the bodies in the system

> # b1 := make_BODY(RF1, m1, 0, 0, Iz1);
** # b2 := make_BODY(RF2, m2, 0, 0, Iz2);**

Step 2: Define the external and friction forces

Define the external forces and the friction forces (if presents)

> # fe := make_FORCE(ground, 0, fa(t), 0, P11, b1);

Step 3: Define gravity

MBSymba had its own gravity variable that we can simply set, and in doing that it will take in to account the gravity in the next calculations

> _gravity := make_VECTOR(ground, 0, -g, 0);

Step 4: Define constraints and multipliers

We need to define as many Lagrange multipliers (usually represented by the lette lambda) as many constraint equations as we have. Therefore this step can be skipped if we don't have any constraint equation.

> # lvars := [seq(lambda||i(t), i = 1.. 3)];
** # newvars := qvars union lvars;**

Now we have to put the constraints in a way that MBSymba will take in to account them when generating the Lagrange equations, using the MBSymba make_CONSTRAINT command:

> #joint_constr := make_CONSTRAINT(Phi,lvars): show(%);

Step 5: Create the equations

First we put all the bodies, external forces and constraint in a single set:

> # mbd := {b1, b2, fe, joint_constr};

And then call the lagrange_equations command that accepts the list of bodies, forces and constraints, the complete variables (so qI plus qD plus the multipliers lambda) and t as the time:

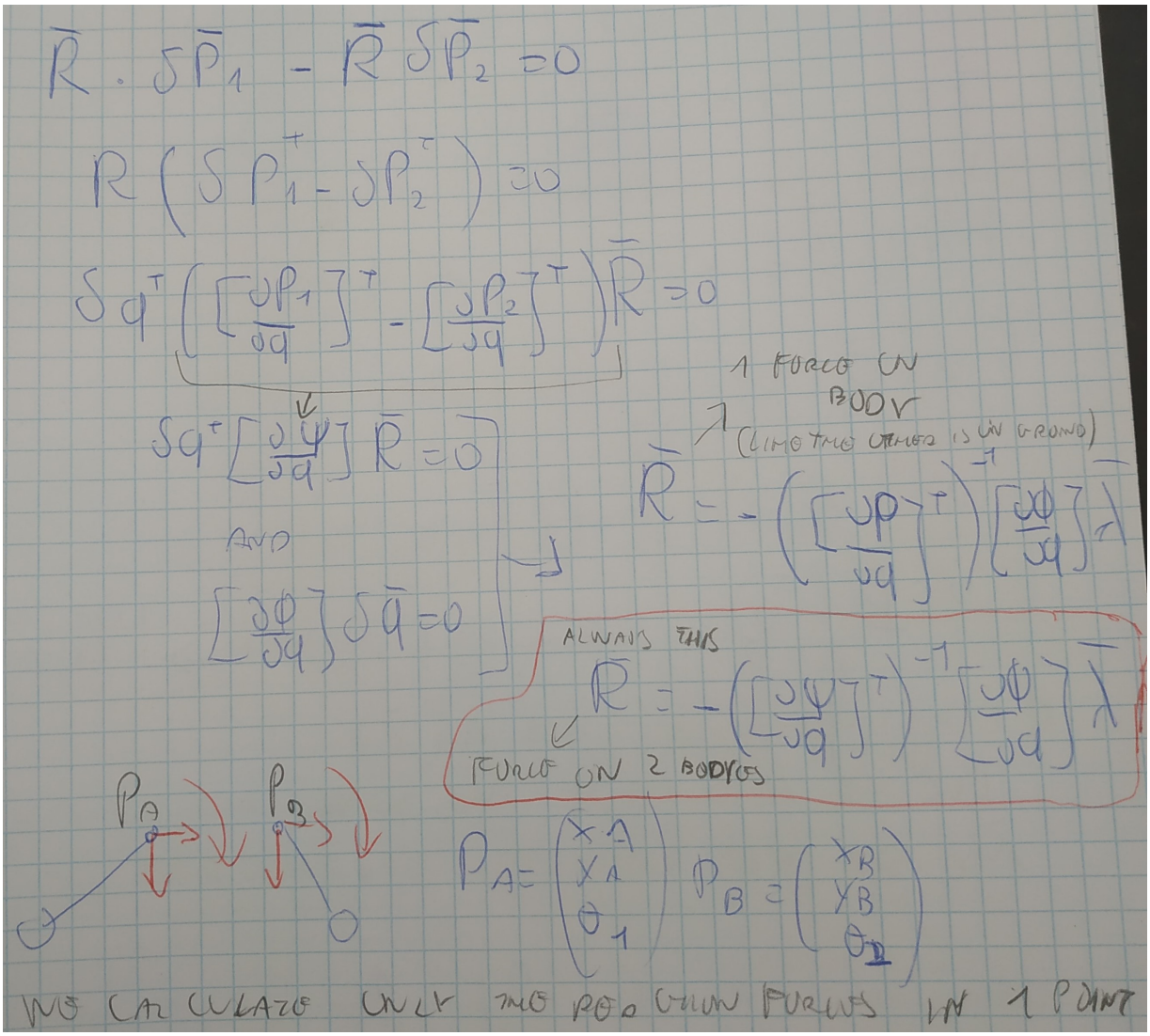
> # Lgr_eq := simplify(lagrange_equations(mbd,newvars,t),size): nops(%);

This will create a system of equation of dimension number of coordinates + number of constraints, and it will contain at the start the equation of motion for the generalized coordinates and then the constraint equations.

We could also put the equation in the first order with the command first_order(), but if we solve it with dsolve() and calculate the intial condition in the special way there is no need to do that.

4.3.1 Compute reaction forces with Lagrange

Using the following formula is possible to find the reaction forces of the system starting from the Lagrange multipliers.
R is the reaction force in a point, P1 and P2 are the point of the two bodies that have to be in the same place and the delta at the start is their virtual displacement.



At the start (first line) we do something similar to the principle of virtual work: we find the velocities of the points where reaction forces are applied and the angular velocities of the reference frames where a reaction torque is applied, remember that being reaction forces every velocity(point) is multiplied by a +reaction forces in the first reference system, and by -reaction in the second reference system, if this is the ground we can avoid doing this passage because the velocity of a point in ground or the angular velocity of the ground is 0.
After we define the vectors of the reaction forces and torques. Then we do like the principle of virtual work: we multiply every point by the vector of the force applied there.
Then we collect the expression of the reaction forces and the velocities (diff(qvars,t)) using the GenerateMatrix command. We are therefore left with the JPsi matrix.

The reaction forces will be the result of LinearSolve(Transpose(JPsi) , Jacobian).<lvars>;

4.4 Initial conditions

4.4.1 Initial conditions (normal way)

Now that we have all the equations of the system in one variable and or in the first order form the result is an index 3 DAE index 1. To find the constraint equations we could use a method, for example the one with the LU decomposition, to reduce the index of the system and get the invariants:

```
> # newq := qvars union Vvars union  Reaction_forces];
> # dae1,alg1 := LUMethod(complete_equations, diff(newq,t) );
> # dae2,alg2 := LUMethod(dae1, diff(newq,t) );
> # ode,alg3 := LUMethod(dae2, diff(newq,t) );
> # alg_constr := alg1 union alg2 union alg3;
```

Remember that in order to do that we need the system in the first order, and we can do that with the first_order() command.
Now we have the algebraic constraints all in one place we can use them to find the initial condition of the system.
The assumption is the following: we know the initial position and the initial velocity of the indipendent variables, and we want to find the initial position and the initial velocity of the dependent cordiantes and the intial values of the reaction forces.
In order to do that we solve the algebraic constraints to respect the unknowns that we have, and after we substitute in this solutions the intial known values:

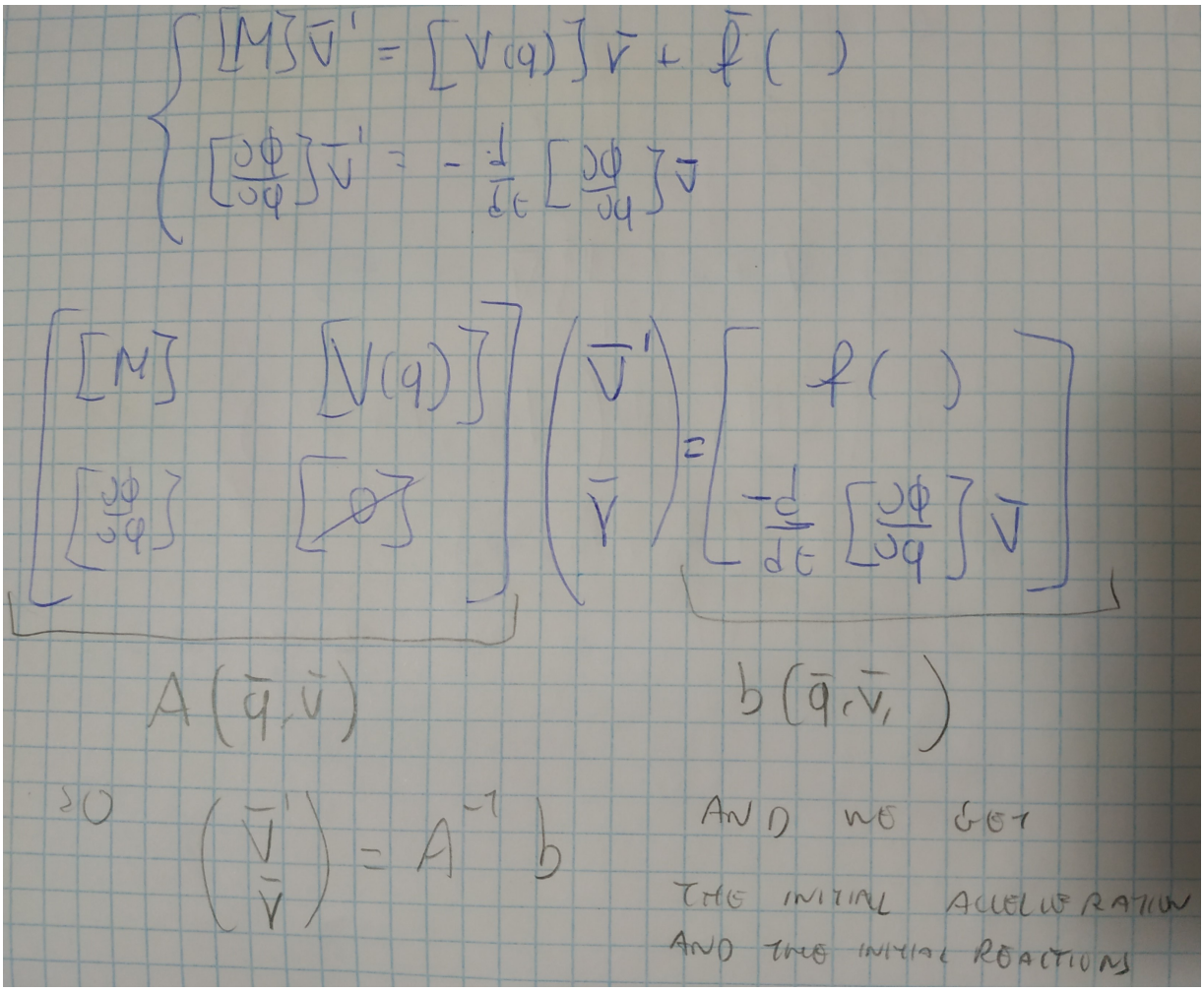
```
> # qI_initial := [theta1(t) = 0.2, y20(t) = 2];
> # velI_initial := [v1(t) = 0, v2(t) = 0.1]; # with v1(t)= diff(theta1(t),t) and v2(t) = diff(y20(t),t)
> # unknowns := [y10(t), theta1(t), x20(t), y20(t), v3(t), v4(t), v5(t), v6(t)] union
  Reaction_forces;

> # sol_alg := solve(alg_constr , unknowns);
> # initial_conditions := subs(data,sol_1,velI_initial,qI_initial,sol_alg);
```

Let's make an example. We have two bodies, a revolute joint, a prismatic joint and a rod joint, with five constraint equations that leave only one degree of freedom. We use the glocal approach and therefore we have six coordinates for the bodies. We compose the equations of the system with the Newton approach: in the first part we have the six v variables, in the second part the six equations of motion and in the third part the five constraints, for a total of 17 equations.
We reduce the index and we end-up with three algebric hidden constraints of five equations each, for a total of 15 equations. The unknowns at the start are the five initial values of the dependendent coordinates, the five initial velocities of the indipendent coordinates and the five initial values of the reactions forces, for a total of 15 unknowns.
It is clear that we can solve the 15 algebraic contraints of the 15 unknowns, and then subsitute velocity and position of the indipendent coordinate and have the initial condition of the system.

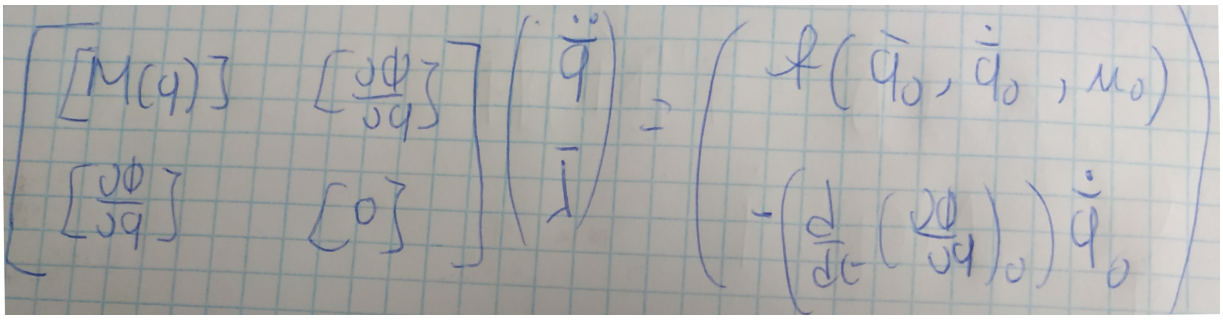
4.2.2 Initial conditions (special way)

Using the Newton approach to describe the system we end up with a special DAE. One that have in its last part the first algebraic part, the constraints. Therefore we can calculate the invariants without the need of a special method:
ALG1 => Phi(t) = 0
ALG2 => JPhi*qdot = JPhi*v = 0
ALG3 => JPhi*vdot + diff(JPhi,t)*v = 0

We can then put everything on one system and for the Newton method we will have something like this:



While for the Lagrange method we will end up with this formulation, beacause the structure of the Lagrange equations is similar to the one of the Newton equations, so that we can put the system in this state derivating two times the constraint equations:



Then we can invert the matrix on the left and we have in initial conditions of the accelleration and of the multipliers. We could also directly solve the accelleration as an ODE but that solution would **drift** a lot (see Baumgarten stabilization to solve this). The same for the Newton approach, but instead of the multipliers we have the reaction forces.
Utilizing this method we don't need the equation in the first order form.

Let's start by putting the Lagrange system in that form:

```
> # aa,bb := GenerateMatrix(Lgr_eq union diff(Phi,t,t), diff(qvars,t,t) union lvars);
  # remember that diff(Phi,t,t) = ALG3 = JPhi*vdot + diff(JPhi,t)*v
```

The procedure with Newton is the same:

```
> # AA, BB := GenerateMatrix(equations union diff(Phi,t,t), diff(qvars,t,t) union reactions);
```

If we differentiate once again the last invariant (alg3, or diff(Phi,t,t)) we will obtain the final part of the equations in order to have and ODE.
Let's assume we have the initial position and velocities of the indipendent coordinate, and thanks to the analistical solution of the constraint equations we have the dependent coordinate position and thanks to the tau matrix and the velocity analysis also the dependent coordinate velocities.

To avoid to invert giant matrix let's calculate numerically aa and bb and then solve numerically the linear system:

```
> # evaluated_aa := subs(data, ics_vI, ics_vD, ics_qI, ics_qD, aa);
  # evaluated_bb := subs(data, ics_vI, ics_vD, ics_qI, ics_qD, bb);
> # sol_initial := LinearSolve(evaluated_aa,evaluated_bb);
```

Or we can invert the giant matrix in the analitical way:

```
> # initial_acc_react_sol := LinearSolve(AA,BB):
> # initial_acc_react := evalf(subs(data, ics_vI, ics_vD, ics_qI, ics_qD, ,initial_acc_react_sol) ) ;
```

Example on how to find the initial conditions for the dependent coordinates:
In order to find the initial condition of the system we don't need the first and second algebraic part, because we can use the analytical solution for the constraint equations for the indipendent coordiantes and the tau matrix for the velocity of the indipendent coordinates.

```
> # ics_qI := [theta1(t) = 0.2, y20(t) = 2];
> # ics_vI := [diff(theta1(t),t) = 0, diff(y20(t),t) = 0.1];
> # ics_qD := subs(data, ics_qI , sol_kine );
> # ics_vD := subs(data, sol_kine, ics_vI , ics_qI , ics_qD, velocities);
```

4.5 Direct Dynamics Solution

In direct dynamics we have the equations of motion and the initial condition of the system, and we want to find out how the system moves after the initial point. To do this we have to solve the DAE of the system and find the function describing the coordinates and the reaction forces / lagrange multipliers.

4.5.1 DSolve

The main command used by Maple to solve differential equations is **dsolve**. It can solve both ordinary differential equations and differential algebric equations in a numerical way. It can solve ODE in an analytical way.

To use it we first need to specify the initial conditions.

```
> # initial_conditions := [ x1(0) = 0, x1__dot(0) = 0];
```

In the following way we are solving analytically the ODE

```
> # sol := dsolve( ode union initial_conditions);
```

We can also solving the ode numerically:

```
> # sol := dsolve( ode union initial_conditions, numerical);
```

This permits us also to solve the DAE numerically:

```
> # sol := dsolve( dae union initial_conditions, numerical);
```

In some cases there is the need to also use the implicit optional parameter:

```
> # sol := dsolve( dae union initial_conditions, numerical, implicit=true);
```

For the DSolve there would be also the way output=listprocedure, but maybe in this way is easier

4.5.1.1 Numerical plots

For the numeric solution we have to use the command odeplot() to plot the solution.
This command is structured like this:

1. The variable containing the solution (mandatory)
2. A list or a list of list, with this structure: [variable on x-axis, function to plot, color="Red" (optional)
3. The variable on the x-axis and its interval, in a form like t=a..b
4. legend = x1 or a list, like legend = [x1,x2,x3] (optional)

```
> # odeplot(sol_numerical, [ [t,x1(t),color="Red" ] , [t,x2(t),color="Green" ] , [t,x3(t),color="Blue" ] ], t=0..2, legend = [x1,x2,x3]);
```

We can also use other options, like axes= boxed, gridlines = true and the size in order to make prettier plots:

```
> # odeplot(sol, [ [t, x1(t), color="DarkOrange"],[t, x1__dot(t), color="Blue"] ], t=0..2, axes=boxed,  gridlines= true, size=[800,300],legend=[x__1,x__1__dot]);
```

In this case we are plotting both x1(t) and x1__dot(t), wich are both variables of the DAE that we have solved. But let's say we have an expression (Phi1) depending on these two values and we want to plot it based on the values of x1(t) and x1__dot(t) that we have found on the solution, this is what we do:

```
> # odeplot(sol, [ [t, Phi1, color="DarkOrange"] ], t=0..2, axes=boxed,  gridlines= true, size=[800,300],legend=Phi1]);
```

4.5.1.2 Normal plots

For tha analitical solution we will use the plot() command.
This comman is structured like this:

1. A function or a list of function to plot (mandatory) - only the function, not x1(t) = f(t) but only f(t), in case of x1(t)=f(t) use the command rhs()
2. The variable on the x-axis and its interval, in a form like t=a..b (mandatory)
3. color = "Red" or a list, like color = ["Red","Green"] (optional)
4. legend = x1 or a list, like legend = [x1,x2,x3] (optional)

```
> #plot( [rhs(sol_analitical[1]), rhs(sol_analitical[2]), rhs(sol_analitical[3]) ] ,t=0..2 , color=
  ["Red","Green","Blue"], legend=[x1,x2,x3] );
```

4.5.2 Baumgarte stabilization

In order for our index-3 DAE solution to respect better the algebraic constraints that we have we could use the Baumgarte stabilization:

- Thus equation

$$\frac{\mathrm{d}^2}{\mathrm{d}t^2}\mathbf{\Phi}(\boldsymbol{q}(t),t) = \mathbf{0}$$

  is substituted with

$$\frac{\mathrm{d}^2}{\mathrm{d}t^2}\mathbf{\Phi}(\boldsymbol{q}(t),t) + 2\zeta\omega\frac{\mathrm{d}}{\mathrm{d}t}\mathbf{\Phi}(\boldsymbol{q}(t),t) + \omega^2\mathbf{\Phi}(\boldsymbol{q}(t),t) = \mathbf{0}$$

- parameters ζ and ω are choosed in such a way to decay error drift as soon as possible maintaining stable the numerical method.
- In general ζ is choosen in $[0, 1]$ and small (but not too much, for example 0.1).

Therefore we substitute in the original constraint equations with this new formula:

```
> # Phi_Baum := diff(Phi ,t,t)+2*zeta*omega__n*diff(Phi,t)+omega__n^2*Phi;
  # Phi_Baum := alg3[1] + 2*zeta*omega__n*alg2[1] + omega__2^2*alg1[1];
  # Phi_Baum := expand(subs(dae[1..2],diff(dae[3],t,t))) + 2*zeta*omega__n*expand(subs(dae
  [1..2],diff(dae[3],t)))+ omega__n^2*dae[3]; # or we can compute it directly from the dae
> # we can also do:
  # Phi_Baum := <diff(Phi,t,t)> + zeta*omega__n*<diff(Phi,t)> + omega__n^2*<Phi>;
  # or
  #Phi_Baum [seq(diff(Phi[i],t,t)+2*zeta*omega__n*diff(Phi[i],t)+omega__n^2*Phi[i],i=1..nops
  (Phi))];
```

After substutuing Phi with Phi_Baum with the correct values of zeta and omega__n in our DAE we will have an index-1 DAE.
This new DAE can be solved with index reduction and then like an ODE or with the special method for index-1 DAE.

Canonical values for Baumgarte: **Wn = 20, eta=0.95**

4.5.3 Index-1 DAE special solution (not very used...)

Index-1 DAE can be solved in a special way. These kind of DAE arise when we use recursive formulation in an open-chain mechanism with the Newton approach or in index-3 DAE using the

Baumgarten stabilization or in normal index-3 DAE where we reduce the index two times.

The key concept here is that you can create the following system: Matrix.<vel + accellerations + reactions> = vector. From this we obtain <vel + accelleations + reations> = Matrix^-1.vector . In this way we can divide accellerations and reactions, solving the reactions as an ODE and then find the reactions forces with the result of the accellerations.
This approach is very similar to the one explained in 4.2.2, where we basically transform the system in a index-1 DAE and then we solve instead for the initial conditions for all the equations of motion and instead of collecting only the accellerations and the reaction forces we collect the velocities, accellerations and reaction forces.

In practice this is done like this:

```
> # M, b := GenerateMatrix(dae_index1, diff(first_vars,t) union reactions):
> # Z_expl := simplify( LinearSolve(M ,b ), trig);
```

Now we have the expression for the variables that we need, but we need to extract the velocities and accellerations (that we assume are in the first 6 rows) and solve them with Maple:

```
> (-Z_expl + <diff(first_vars,t) union reactions>)[1..6]:
  subs(data,%):
  sol_maple := dsolve(convert(%,list) union initial_conditions , numeric);
```

Now we can plot the variables:

```
> # odeplot(sol_maple, [ [t, x1(t), color="DarkOrange"],[t, x1__dot(t), color="Blue"] ], t=0..2,
  axes=boxed,  gridlines= true, size=[800,300],legend=[x__1,x__1__dot]);
```

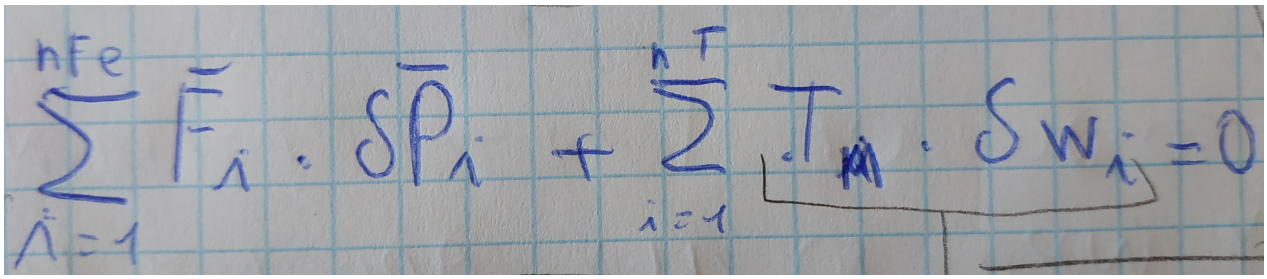
And the reaction forces:

```
> # odeplot(sol, [ [t, Z_expl[7], color="DarkOrange"] ], t=0..2, axes=boxed,  gridlines= true,
  size=[800,300],legend=r1]);
```

4.6 Principle of Virtual Work

For a movement of the coordinate that is: arbitrary, small (infinitesimal) and satisfy the constraints the work generated must be zero. Moreover this operation is done when time is frozen.
Remember that the reactions forces for the constraint equations don't perform any work, so we can ignore them. Moreover, it is impossible to use the principle of virtual work in the presence of friction.

The general formula of the virtual work is the following:



We first find the relationship between the virtual displacement of the coordinates:

```
> # virtual_qD := [ seq (cat ( 'delta__', op(0, qD[i] ) ) (t)
                = tau[i]*delta__theta1(t),
                i=1..nops(qD))]: <%>;
```

Which is basically the velocity ratios of the tau matrix:

```
> # vel_ratios := combine( op( solve( diff (Phi,t), diff(qD,t) ) ) ) : <%>;
> # tau := simplify( -MatrixInverse( JPhi__d). JPhi__i) ); # other way to compute the velocities
  ratios
  # tau.<diff(qI,t)>;
  # vel_ratios := [ seq( diff(qD[i],t) = %[i], i=1..nops(qD) ) ];
```

Then we define bodies and forces

```
> # body1 := make_BODY(G1,m1,0,0,iz1);
  # body2 := make_BODY(G2,m2,0,0,iz2);
  # body3 := make_BODY(G3,m3,0,0,iz3);
  # w1 := make_FORCE(ground, 0, -m1*g, 0, G1, body1);
  # w2 := make_FORCE(ground, 0, -m2*g, 0, G2, body2);
  # w3 := make_FORCE(ground, 0, -m3*g, 0, G3, body3);
  # Fe := make_FORCE(ground,-fe(t),0,0,P21,body1); # an external force
```

OR we can directly define the vectors of the forces:

```
> # w1 := make_VECTOR(ground, 0, -m1*g, 0);
> # w2 := make_VECTOR(ground, 0, -m2*g, 0);
> # w3 := make_VECTOR(ground, 0, -m3*g, 0);
> # Fe := make_VECTOR(ground, -fe(t), 0, 0);
```

Then for every point where a force is applied we direrive the velocity of that point (that represents the infinitesimal displacement):

```
> # V_G1 :=  velocity(G1):
  # V_G2 :=  velocity(G2):
  # V_G3 :=  velocity(G3):
  # V_P21 :=  velocity(P21):
```

Now we are ready to compute the virtual work:

```
> # VW := dot_prod(V_P21,Fe) + dot_prod(V_G1,w1) + dot_prod(V_G2, w2) + dot_prod(V_G3,
  w3);
```

And then we can solve for the external force to find its value in order to keep the system in that position

```
> # sol_fe := solve( VW , fe(t) ) ;
```

Substitute in that the data, the velocities and the kinematic solution and we have the numerical solution, that we can plot (if you find some derivatives of qI still in there try to simplify them, they should not be present(?)):

```
> # final_sol_fe := subs(data, vel_ratios, sol_kine, data, sol_fe );
  # plot( subs(theta1(t) = X, final_sol_fe) , X = 0..Pi/2 )
```

4.7 Equilibrium condition

When a body is in equilibrium the velocity (and therefore the accelleration) of all the coordinates and of all the reaction forces (or multipliers) is equal to 0.
Therefore to find the equilibrium condition (if it exists) we have to take our equation of motions plus constraint equations, therefore our DAE, not of first order, and impose all the velocities and accellerations equals to 0.

Create the equilibrium conditions:

```
> # [ seq( diff(qvars[i],t) = 0, i=1..nops(qvars) ) ];
  # [ seq( diff(qvars[i],t,t) = 0, i=1..nops(qvars) ) ];
  # eq_cond := %% union %;
```

Then in the DAE, with the constraint equations and the equations of motions, we substitute the equilibrium condition and the DAE will became a non-linar system.

```
> # eqilibrium_eq := subs(data, eq_cond, dae);
```

Then we can solve the dae, in both the analitical and numerical way:

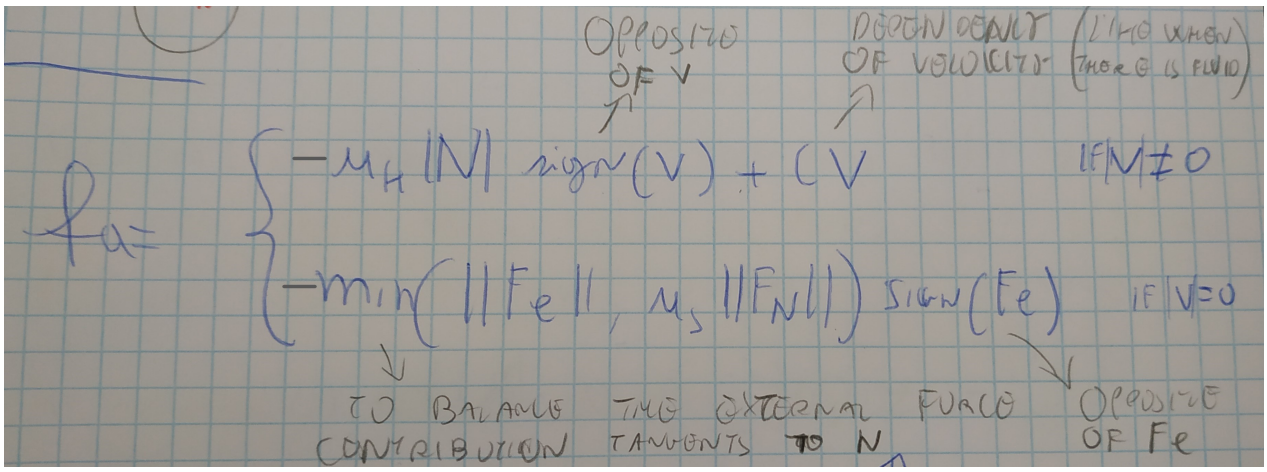
```
> # solve(equilibrium_eq, qvars union reactions);
  # fsolve(equilibrium_eq);
```

There are 2 ways in which we can compute the equilibrium point:

1. Substitute in the equation of motion the equilibrium condition, the data and the kinematic solution. Solve the system for the reaction forces/external forces/ qI. All the solution in this case will be valid but if the kinematic solution ic somplex resolving the system analitically might be complex, and we might have to resolve it only numerically
2. Substitute in the equation of motion + constraint equation (DAE) the data. Solve for all the forces and for all the variables. In this case we avoid substituting the kinematic solution that might increase too much the complexity of the problem, but the solutions obtained with this method might not be valid in our kinematic solution, so we have to check.

4.8 Friction models

With Coulomb friction model we can model friction. This model doesn't take into account the pre-sliding fase and the Stribeck effect, but only the maximum static friction and the dynamic friction.
Describing friction we have two constants: Us and Uk, the friction coefficient in static condition and in dynamic condition.
We use this formula to model friction:



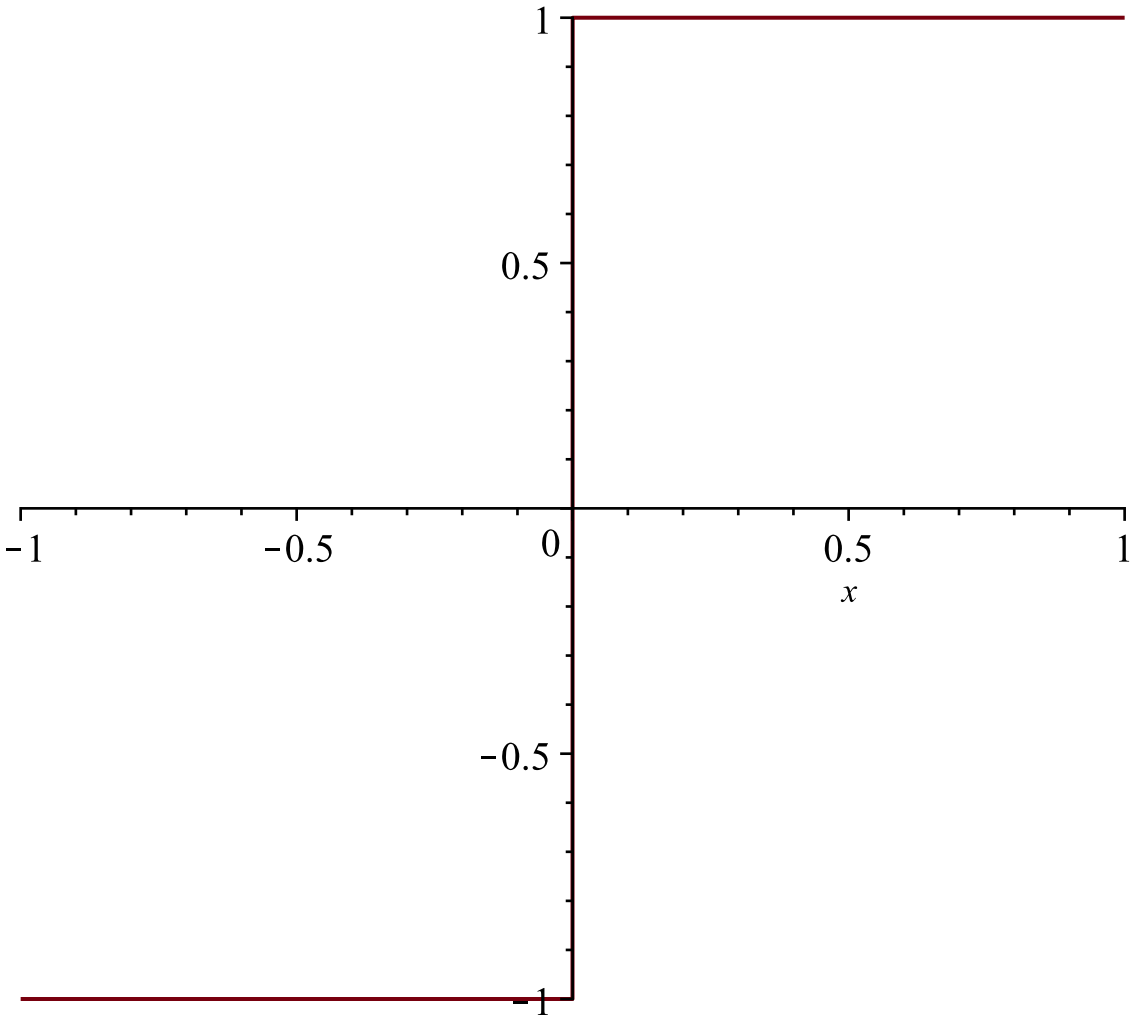
If we know we are in static condition we can set the friction equal to the sum of all the external forces acting on the direction where there is friction. With Nweton we can have for example m*a = external_forces + friction, in this case the static condition is when -friction = external forces. But we have to remember that also friction has a maximum value, wich is abs(N)*Us in static conditions, which is the normal force to the direction where there is friction in absolute value times the static friction coefficient. If

the external forces surpass this value we have that friction is not sufficient anymore to keep the system still.
To understand when this happens we can plot the external forces in that direction and N*Us, when the first surpass the second we have that the system will move.
In case of systems always in movement we can use only the dynamic friction model, which will produce a force opposite to the velocity.

Sometimes is better to not use the sing(x) function (signum() in Maple) and instead use its approximation: sin(arctan(x/x0)), with x0=0.001 or 0.01:

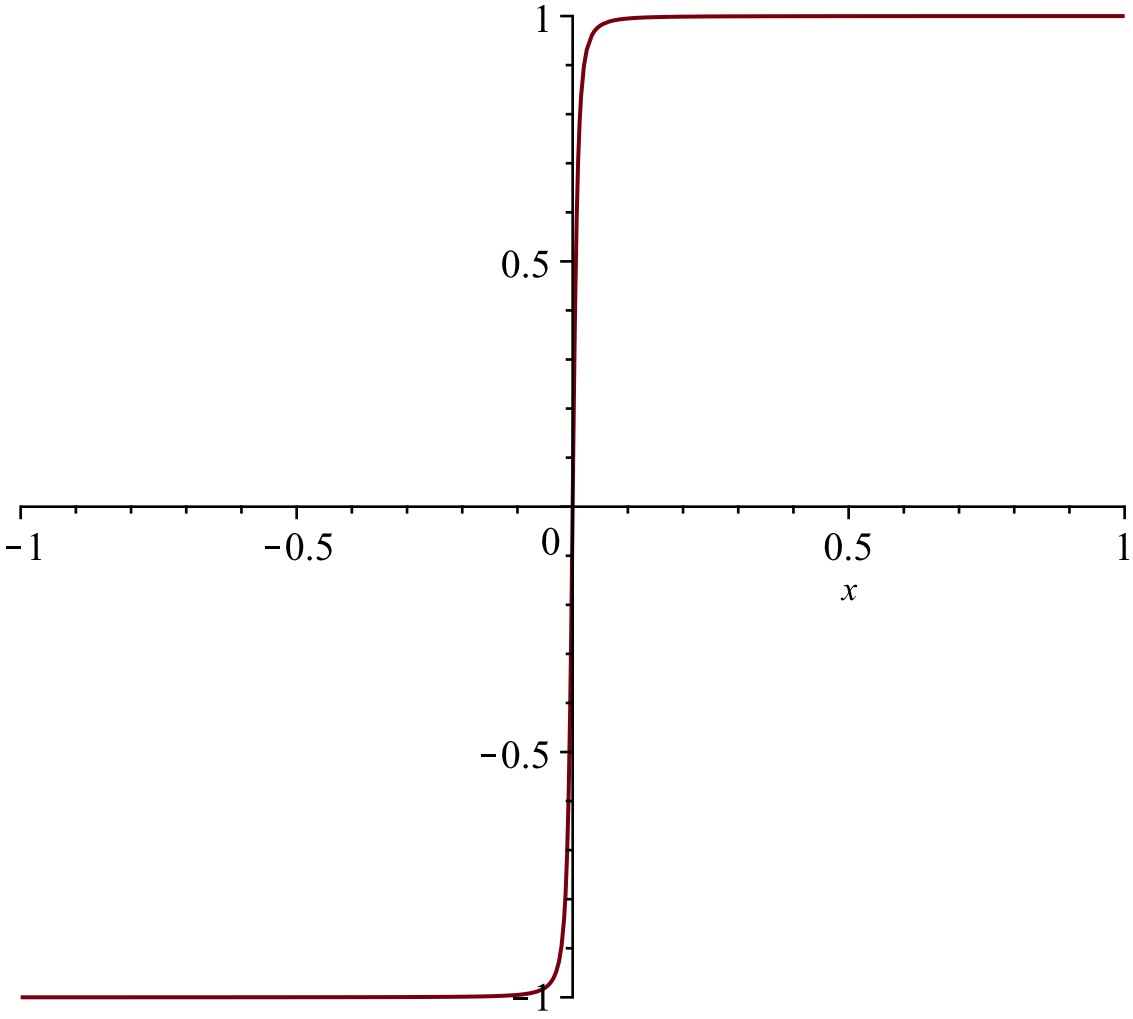
```
> signum(x);
  plot(%,x=-1..1);
```

$$\mathrm{signum}(x)$$




```
> sin(arctan(x/x0));
  plot(subs(x0=0.01,%),x=-1..1);
```

$$\frac{x}{x0\sqrt{1+\frac{x^2}{x0^2}}}$$



So dynamic friction would be:

> dynafrict := -mu__c * abs(N(t)) * sin(arctan(diff(s(t),t) / v0));

$$dynafrict := -\frac{\mu_c \left| N(t) \right| \left(\frac{d}{dt} s(t) \right)}{v0 \sqrt{1 + \frac{\left(\frac{d}{dt} s(t) \right)^2}{v0^2}}} \tag{4.8.1}$$

> staticfrics := -mu__s * abs(N(t))* sin(arctan(Fe / x0));

$$staticfrics := -\frac{\mu_s \left| N(t) \right| Fe}{x0 \sqrt{1 + \frac{Fe^2}{x0^2}}} \tag{4.8.2}$$

4.9 Inverse Dynamics

In inverse dynamics we know how the system should move, because for example we have an expression for the indipendent variable that we want to follow (ex: the driving body rotate at a certaint constant speed omega), and therefore, using the kinematic solution, we also know how the other coordinates should move.

What we don't know is the values of the reaction forces/lagrange multipliers and more importantly how the control should behave (for example to rotate the driving body we have a control torque that we need to calculate).
In the case of direct dynamics all the external forces are know and using Lagrange (but is also very similar with Newton) we have the expression of the qvars and of the lambdas unknown, and we have a corresponding number of equations (the equations of motion for the qvars and the constraint equations for the lambdas) in the DAE, as said before all the external forces are known.
In the case of indirect dynamics we know the qvars expressions, and therefore the only unknowns are the lambdas (the same number as constraint equations) and the external forces (in number of the degreesof freedom). These unknowns are present only in the equations of motion and should be the same number as the equation of motions, and therefore we can discard the constraint equations for the inverse dynamics.
The practical procedure is the following:
1. Take only the equation of motion
2. Solve them for the unknowns (lambdas/reactions + external control)
3. Substitute the data (the velocity solution) and the kinematic solution
4. Substitute the equation of motion of the indipendent variables
5. Plot

Following we have some examples of the procedure , first using Lagrange and ther using the Newton approach

```
> # subs( data, lgr_eq[1..7] ) : <%>;
  # eval( solve( %%, lvars union [Tm(t)] ) ) ;
  # sol_inv_dyna := eval( subs (sol_kine, data, sol_theta1,%[1] ) );

> # eval( subs ( data,  dae)):<%>;
  # solve(%%[1..6], reactions union [Tm(t)] ) ;
  # sol_inverse_dyna := eval( subs( sol_kine, sol_theta1, data,%[1] ) ) ;
```

It is also possible to do the inverse: find the equation of motion, substitute velocities, kinematic solution, data, and the time law and then solve for the unknowns(control and reactions/lambdas). This is easier to do when there is friction and the kinematic solution is easy.

4.10 Linearization

When we linearize we substitute the current equation of motion with one that has appearing the variables linearly. It is possible doing this only around a equilibrium point, that yes can be generalized but when computing it has to be substituted with a value.
Which means that when we find in the equation of motion f(x) we substitute it as f(x0)*(x-x0).
This is because we want to put the system in a form like **Ax' + Bx + c =0** (with the x vector representing a small perturbation now), which is easy to manipulare. In order to do that everything should appear linearly: no cos, sin, arctan, x^2 ecc...
We can linearize only around equilibrium points.
There is a command in MBSymba that does this for you: linearize.
After we linearize we put the system in the previously written form, then we solve for x' (therefore we multiply B and C by A(-1)), and then we calulate the eivenvalues of the resulting matrix A(-1)*B, and if the real part is all negative we have a stable equilibrium point, otherwise if only one of the eigenvalues has the real part positive we have an unstable equilibrium.

Here there is a pendulum example:

```
> restart: with(plots): with(LinearAlgebra):
```

```
with(MBSymba_r6):
> RF1 := rotate('Z',theta(t)):
  G1  := make_POINT(RF1,0,-L,0):
> _gravity := make_VECTOR(ground,0,-g,0):
> pendulum := make_BODY(G1,m,0,iz):
```

Equations with lagrange approach

```
> vars := [theta(t)]:
> eqns := lagrange_equations({pendulum},vars,t);
```

$$eqns := \left[m L^2 \left(\frac{d^2}{dt^2} \theta(t) \right) + m \sin(\theta(t)) L g \right] \tag{4.10.1}$$

```
> vars1,eqns1 := first_order(eqns,vars,t,flag):<eqns1>;
```

$$\begin{bmatrix} m L^2 \left(\frac{d}{dt} \theta_{dot}(t) \right) + m \sin(\theta(t)) L g \\ -\theta_{dot}(t) + \frac{d}{dt} \theta(t) \end{bmatrix} \tag{4.10.2}$$

```
> eval(subs(theta__dot(t)=0,eqns1)); # search for the equilibrium points
  solve(%[1],[theta(t)],explicit=true,allsolutions=true); # 0 and Pi are equilibrium points
```

$$\left[m \sin(\theta(t)) L g, \frac{d}{dt} \theta(t) \right]$$

$$[[\theta(t) = \pi\_Z1\sim]] \tag{4.10.3}$$

```
> lin_point := [theta(t) = theta0, theta__dot(t)=dottheta0];          # the generalized linearization
  poin
```

$$lin\_point := [\theta(t) = \theta 0, \theta_{dot}(t) = dottheta0] \tag{4.10.4}$$

```
> eqns1_lin := linearize(eqns1,lin_point): <%>;          # linearization with MBSymba_r6
```

$$\begin{bmatrix} m L^2 \left(\frac{d}{dt} \theta_{dot}(t) \right) + m \sin(\theta 0) L g + m \cos(\theta 0) (\theta(t) - \theta 0) L g \\ -\theta_{dot}(t) + \frac{d}{dt} \theta(t) \end{bmatrix} \tag{4.10.5}$$

Now that we have the linearized equation we can set the numeric value of the linearization point (theta0) and the initial conditions (theta1(0) and D(theta1)(0)), and solve the system.

```
> ics0 := [theta__dot(0)=.1,theta(0)=0.1]; # initial conditions of the system
```

$$ics0 := [\theta_{dot}(0) = 0.1, \theta(0) = 0.1] \tag{4.10.6}$$

```
> # linearized systems around the two equilibrium points: theta = 0 and theta = Pi
  evaluated_eq0 := subs( [L=1,m=0.1,iz = 0.08,g=9.81], theta0=0, eqns1_lin):<%>;
  evaluated_eqPi := subs( [L=1,m=0.1,iz = 0.08,g=9.81], theta0=Pi, eqns1_lin):<%>;
```

$$\begin{bmatrix} 0.1 \frac{d}{dt} \theta_{dot}(t) + 0.981 \theta(t) \\ -\theta_{dot}(t) + \frac{d}{dt} \theta(t) \end{bmatrix}$$

$$\begin{bmatrix} 0.1 \frac{d}{dt} \theta_{dot}(t) + 3.081902394 - 0.981 \theta(t) \\ -\theta_{dot}(t) + \frac{d}{dt} \theta(t) \end{bmatrix} \tag{4.10.7}$$

```
> # solve the odes of the two linerized systems and of the real one
  sol_linearized0 := dsolve( evaluated_eq0 union ics0,numeric);
  sol_linearizedPi := dsolve( evaluated_eqPi union ics0,numeric);
```

```
sol_real := dsolve( subs( [L=1,m=0.1,iz = 0.08,g=9.81],eqns1) union ics0, numeric);
```

$$sol\_linearized0 := \mathbf{proc}(x\_rkf45) \ \ldots \ \mathbf{end\ proc}$$

$$sol\_linearizedPi := \mathbf{proc}(x\_rkf45) \ \ldots \ \mathbf{end\ proc}$$

$$sol\_real := \mathbf{proc}(x\_rkf45) \ \ldots \ \mathbf{end\ proc} \qquad \textbf{(4.10.8)}$$

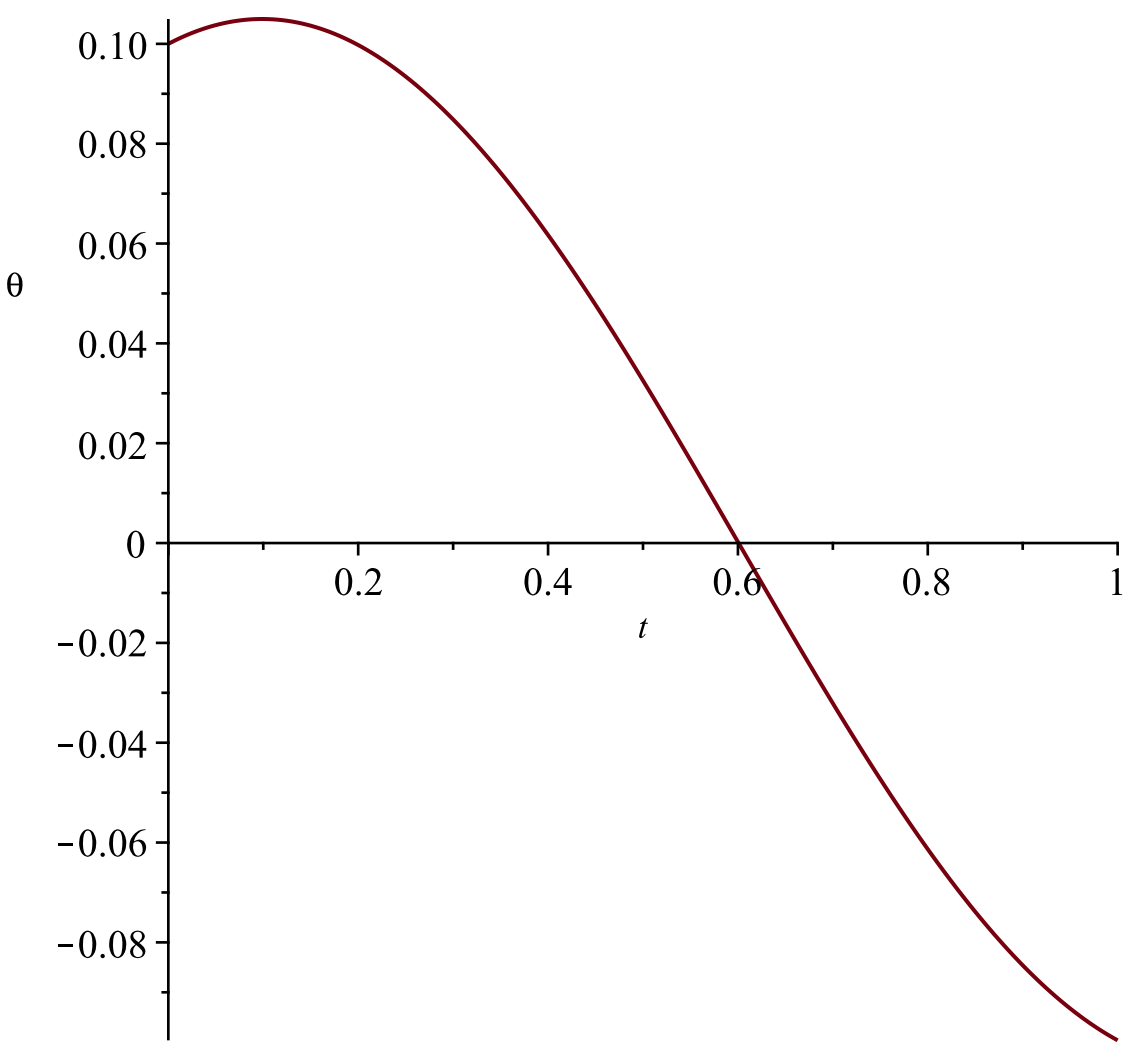
```
> Describe(state_space); # MBSymba_r6 command to extract the aa,bb and cc matrices
  aa,rest := GenerateMatrix(eqns1_lin,diff([theta(t),theta__dot(t)],t)):
  bb,cc := GenerateMatrix(convert(rest,list),[theta(t),theta__dot(t)]):
  aa,<diff([theta(t),theta__dot(t)],t)>,bb,<[theta(t),theta__dot(t)]>,cc;        # the system in the
  form aa*x' + bb*x + cc =0
  aa.<diff([theta(t),theta__dot(t)],t)> + bb.<[theta(t),theta__dot(t)]> + cc;

state_space( eqns0::list, t::name, x::list )
```

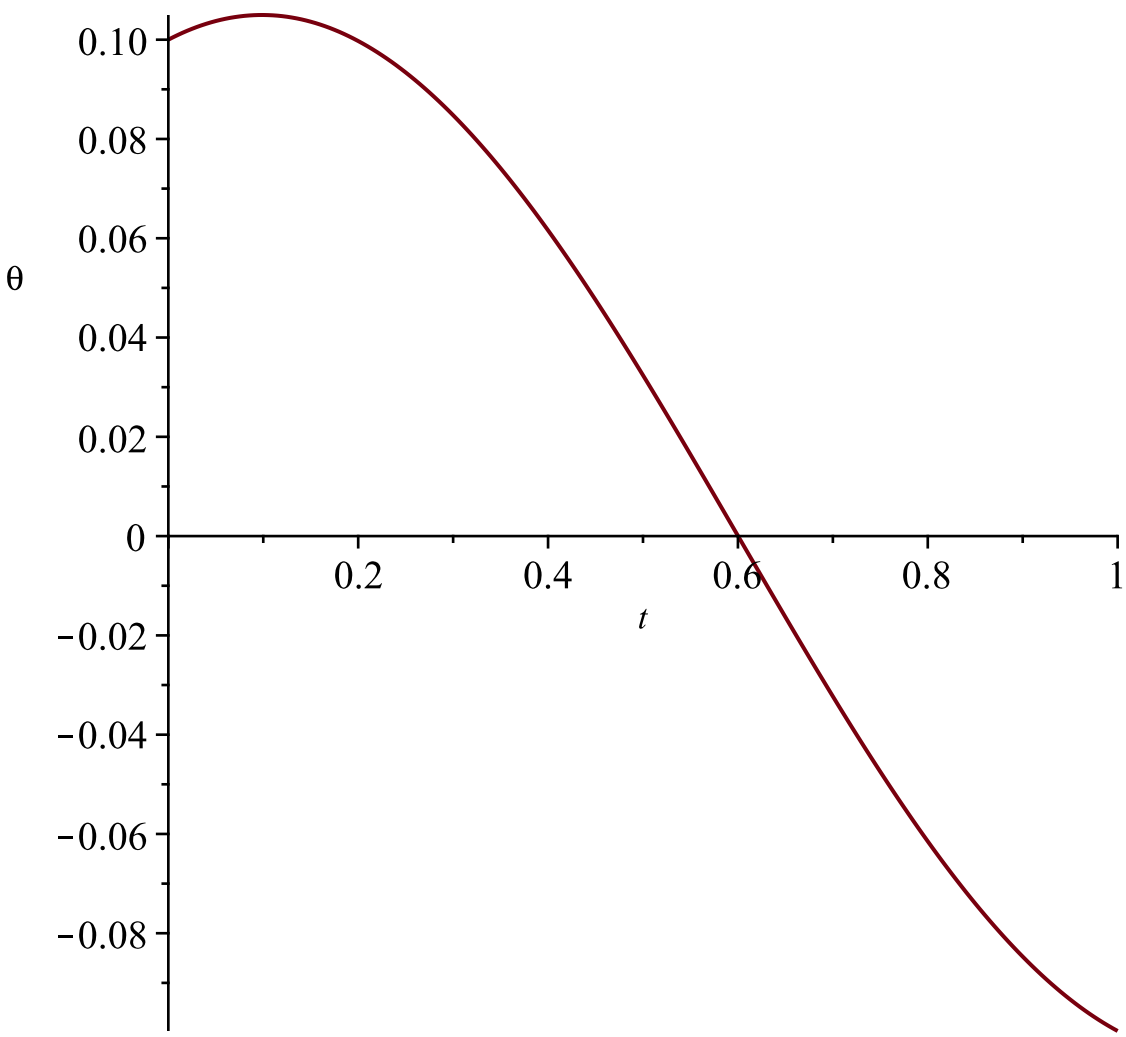
$$\begin{bmatrix} 0 & m L^2 \\ 1 & 0 \end{bmatrix}, \begin{bmatrix} \frac{d}{dt}\,\theta(t) \\ \frac{d}{dt}\,\theta_{dot}(t) \end{bmatrix}, \begin{bmatrix} -m\cos(\theta 0)\,L g & 0 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} \theta(t) \\ \theta_{dot}(t) \end{bmatrix},$$

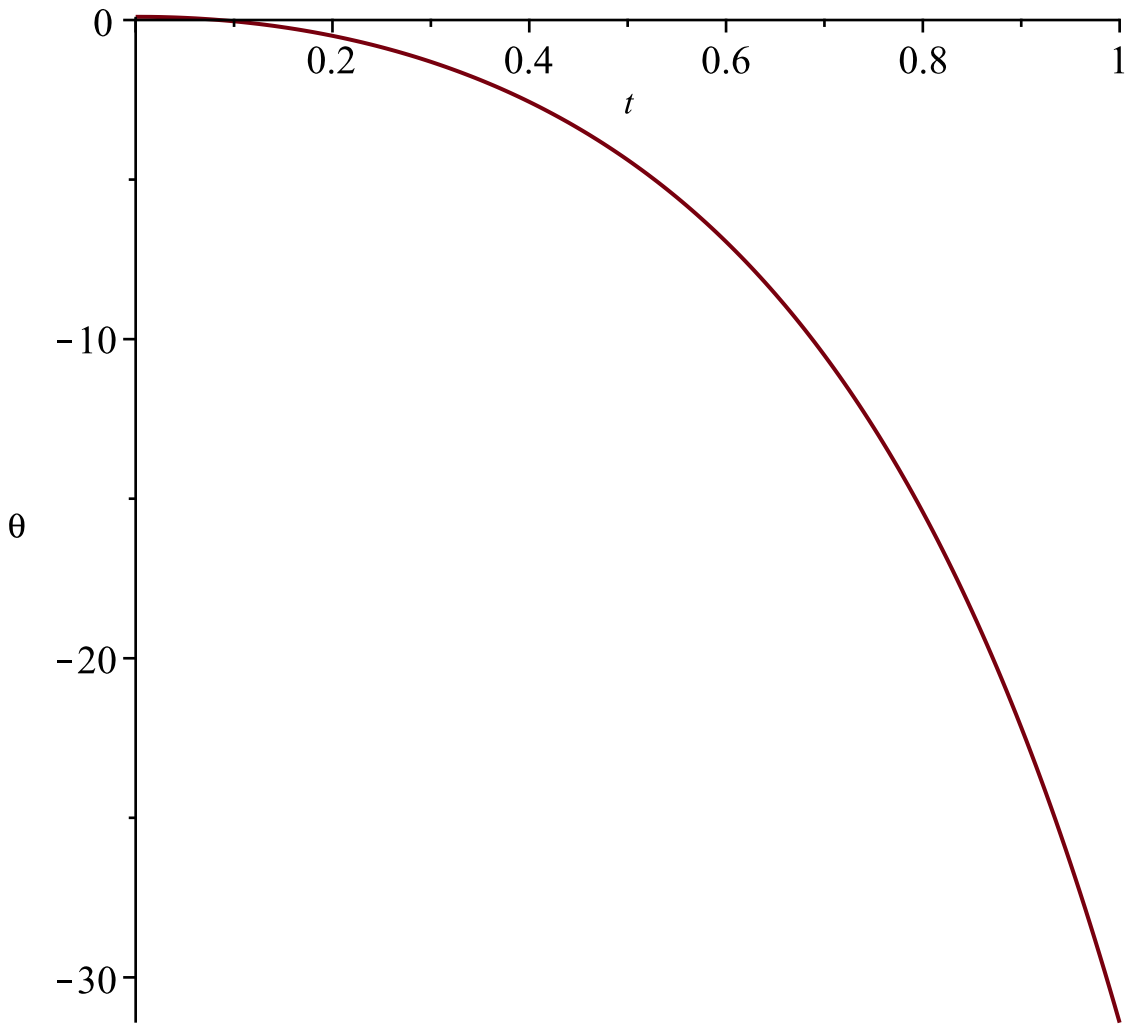
$$\begin{bmatrix} -m\cos(\theta 0)\,\theta 0\,L g + m\sin(\theta 0)\,L g \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} m L^2 \left(\frac{d}{dt}\,\theta_{dot}(t) \right) - m\cos(\theta 0)\,L g\,\theta(t) - m\cos(\theta 0)\,\theta 0\,L g + m\sin(\theta 0)\,L g \\ \frac{d}{dt}\,\theta(t) + \theta_{dot}(t) \end{bmatrix} \qquad \textbf{(4.10.9)}$$

```
> odeplot(sol_real,[t,theta(t)],t=0..1);              # real solution
  odeplot(sol_linearized0,[t,theta(t)],t=0..1);  # linearized around theta=0 with starting condition
  theta=0.1, good results
  odeplot(sol_linearizedPi,[t,theta(t)],t=0..1); # linearized around theta=Pi with starting condition
  theta=0.1, awful results, drift is exteme
```


θ
0.10
0.08
0.06
0.04
0.02
0
−0.02
−0.04
−0.06
−0.08
0.2
0.4
0.6
0.8
1
t


θ
0.10
0.08
0.06
0.04
0.02
0
−0.02
−0.04
−0.06
−0.08
0.2
0.4
0.6
0.8
1
t



> LinearSolve(aa,bb); # Linearsolve to solve for the x' = aa^(-1)*(bb*x + cc)
Eigenvalues(subs([L=1,m=0.1,iz = 0.08,g=9.81],theta0=0,%)); # check if the matrix aa^(-1)*bb has eigenvalues with real positive part, in this case no, stable equilibrium in theta = 0
Eigenvalues(subs([L=1,m=0.1,iz = 0.08,g=9.81],theta0=Pi,%%)); # the first eigenvalue has the real part positive, unstable equilibrium in theta=Pi

$$\begin{bmatrix} 0 & 1 \\ -\frac{\cos(\theta 0)\, g}{L} & 0 \end{bmatrix}$$

$$\begin{bmatrix} 3.132091953\,I \\ -3.132091953\,I \end{bmatrix}$$

$$\begin{bmatrix} 3.132091953 \\ -3.132091953 \end{bmatrix} \qquad \textbf{(4.10.10)}$$

So summing all that we have said about linearization, when we want to linearize a system:
1. Find the equation of motion (in the first order)
2. Find the equilibrium point (with both velocity =0 and with velocity constant (stationary motion))
3. Linearize the system around a generic equilibrium points with linearize() (using as variables the

newvars generated by the first order command)

4. Substutite the chosen equilibrium point in the system (at this point you could also substitute the x0 - x0(t) with deltax0(t), but is optional)
5. Put the system in the form Ax'+Bx+C=0 (x' is diff(newvars , t) with newvars generated by the first order command, containing the qvars and their derivatives)
6. Do LinearSolve(A,B)
7. Look at the real part of the eigenvalues of the previous matrix, if they are all negative you have a stable equilibrium.