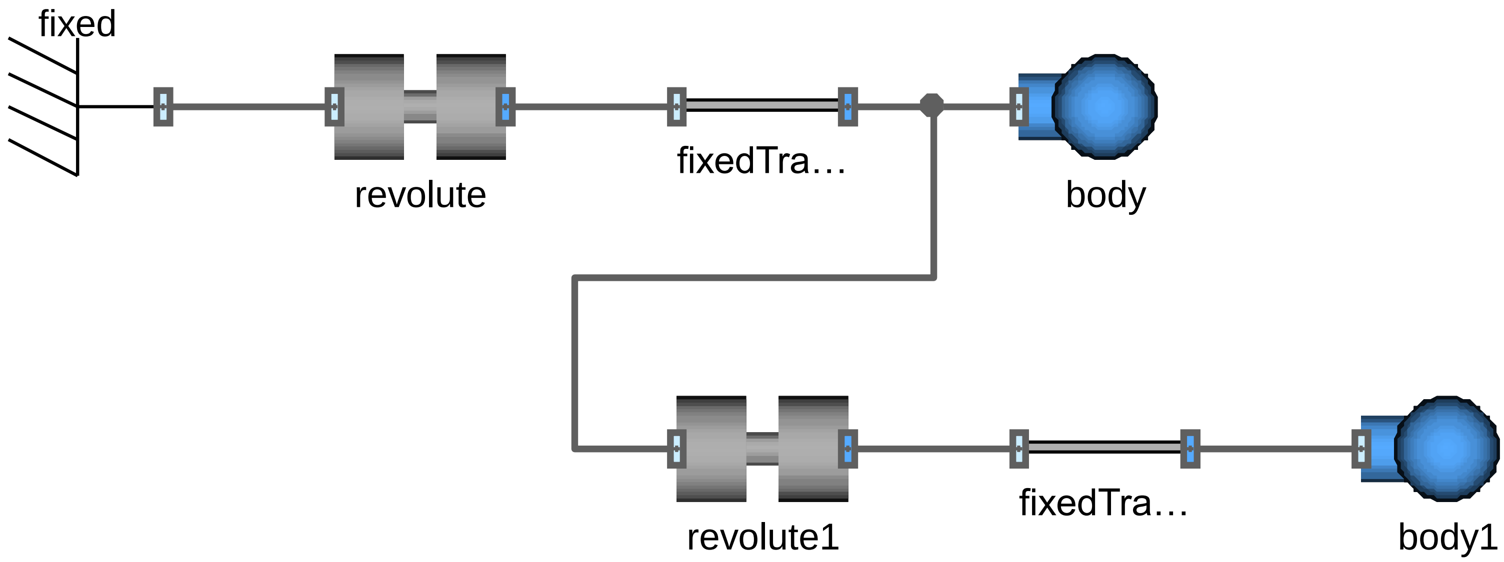
Virtual Physics
Equation-Based Modeling

TUM, November 22, 2022

2D-Mechanical Systems

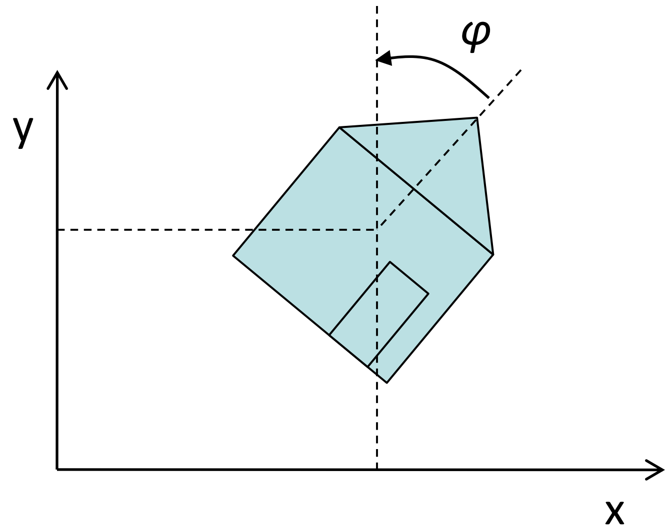


Dr. Dirk Zimmer

German Aerospace Center (DLR), Robotics and Mechatronics Centre

Planar Mechanics

- In planar mechanics, we describe the physics of a multi body system in a two-dimensional plane.



- In the planar world, all motions and positions can be described by two translational positions and an angular orientation
- By convention we denote the horizontal position with x, the vertical position with y and the orientation by the angle φ (phi).

The Task

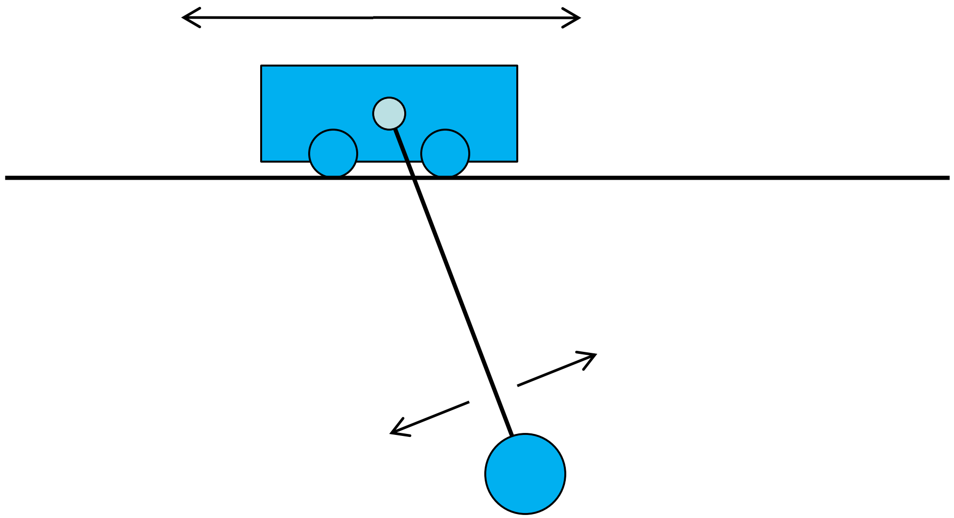
- In this lecture, we want to start modeling our own library for planar mechanics.

- The design of a library is a very multifaceted task. We have to concern:
  - the structure of the library
  - the design of the connector
  - usability of the components
  - effective code reuse
  - solutions for initialization
  - and many things more

Decomposition into components

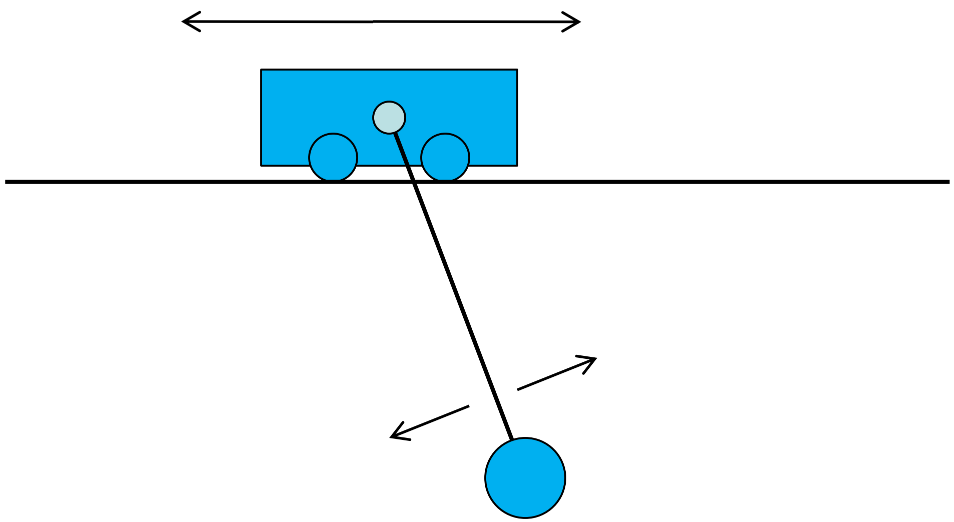
- The first question that we have to address concerns how we want to decompose a planar mechanical system into ideal components.

- Let us investigate an example: The crane crab.



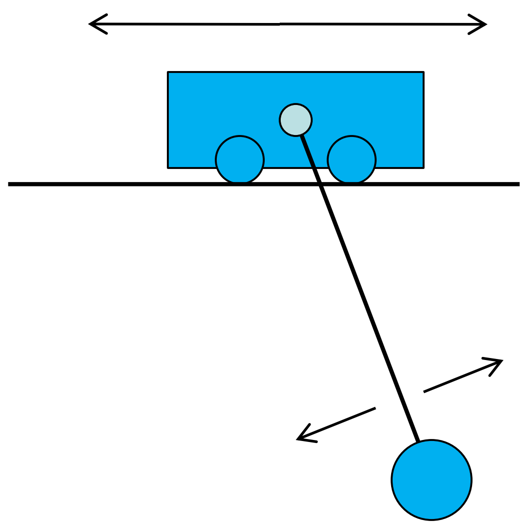
Decomposition into components

- The crane-crab has two degrees of freedom: The horizontal movement of the carriage wagon and the load revolting like a pendulum.
- The carriage and the load possess mass and an inertia
- The cable has given length.



Decomposition into components

- There shall be one ideal component that represents mass and inertia.
- All other components shall be weightless.
- Some parts represent geometric objects, like a rod of finite length.
- The degrees of freedom in motion can be expressed by special joints.
- Furthermore, there are “force” components like springs and dampers.

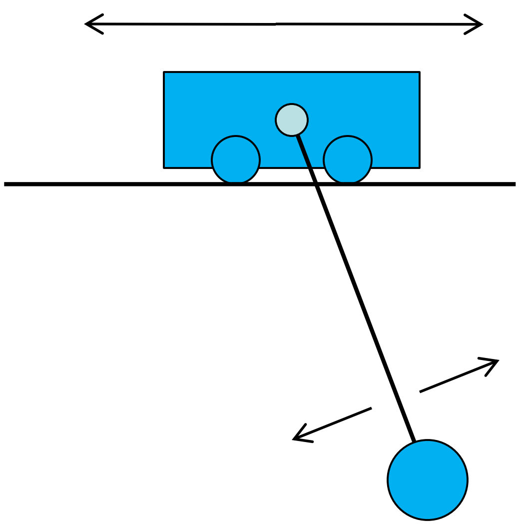


Decomposition into components

Robotics and Mechatronics Centre

- Here is a quick layout of the library…

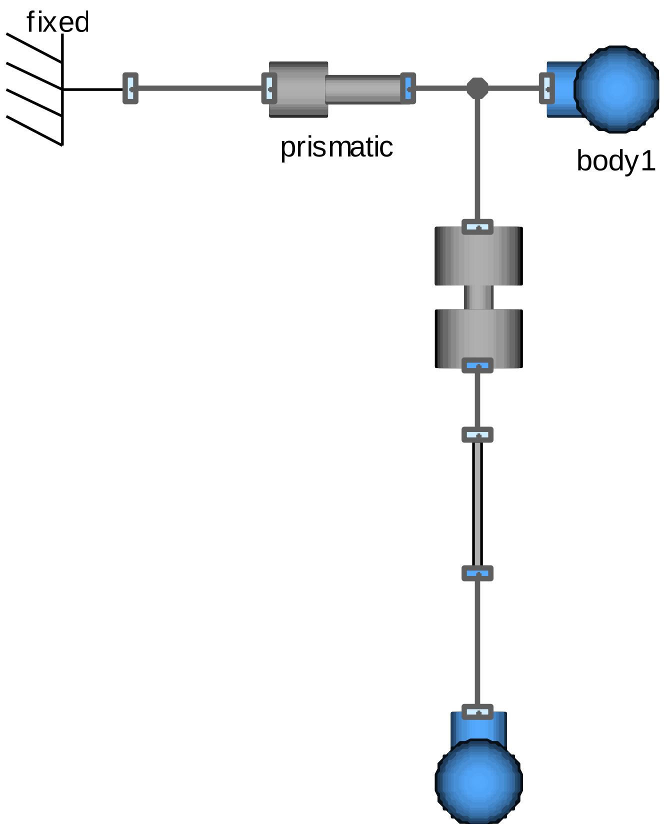
- Parts
  - Body (Mass and Inertia)
  - FixedTranslation
  - FixedRotation
- Joints
  - Revolute Joint
  - Prismatic Joint
- Forces
  - Spring
  - Damper



Decomposition into components

Robotics and Mechatronics Centre

- …and the corresponding decomposition of the crane crab.



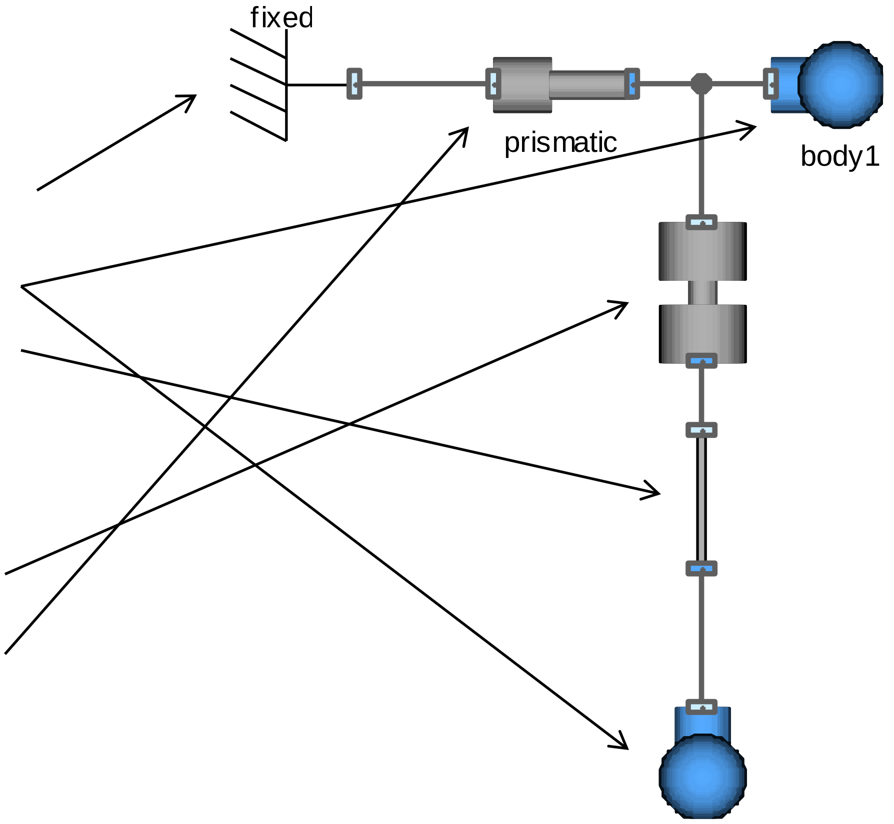
- Parts
  - Wall
  - Body (Mass and Inertia)
  - FixedTranslation
  - FixedRotation
- Joints
  - Revolute Joint
  - Prismatic Joint
- Forces
  - Spring
  - Damper

Decomposition into components

Robotics and Mechatronics Centre

- …and the corresponding decomposition of the crane crab.

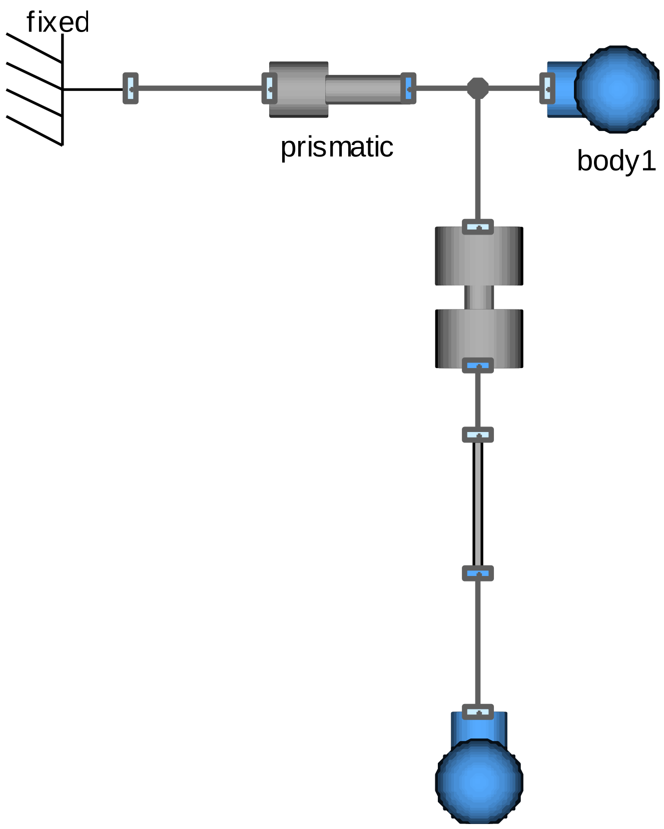
- Parts
  - Wall
  - Body (Mass and Inertia)
  - FixedTranslation
  - FixedRotation
- Joints
  - Revolute Joint
  - Prismatic Joint
- Forces
  - Spring
  - Damper



Decomposition into components

Robotics and Mechatronics Centre

- All components of this library shall use one common connector.



- Parts
  - Wall
  - Body (Mass and Inertia)
  - FixedTranslation
  - FixedRotation
- Joints
  - Revolute Joint
  - Prismatic Joint
- Forces
  - Spring
  - Damper

Connector Variables

- From 1D-mechanics, we learned that the we should choose force and torque as flow-variables and position and angle as potential variables.
- Planar mechanics combine three 1D-subsytems. Hence the following connector design seems natural.

Potential variables

x (horizontal position)

y (vertical position)

φ (orientation angle)

Flow variables

f_x (horizontal force)

f_y (vertical force)

τ (torque)

Connector Variables: Modelica

Robotics and Mechatronics Centre

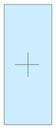
- Here, the corresponding Modelica-Code:

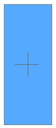
```
connector Frame "General Connector for planar mechanical components„

  SI.Position x          "x-position";
  SI.Position y          "y-position";
  SI.Angle phi           "angle (counter-clockwise)";
  flow SI.Force fx       "force in x-direction";
  flow SI.Force fy       "force in y-direction";
  flow SI.Torque t       "torque (counter-clockwise)";

end Frame;
```

Connectors

- It is common style to extend two connectors with different icons from the general connector.
- Some components contain characteristics that are directed. Hence it is helpful to see, if your connecting to side A or side B.

```
connector Frame_a
  extends Frame;
end Frame_a;


connector Frame_b
  extends Frame;
end Frame_b;
```



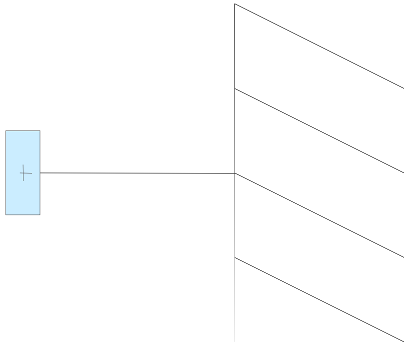


- All of these connectors are collected in an interface package.

Fixed Component

Robotics and Mechatronics Centre

We can already model the first basic components. Let us start with the wall component that represents a fixation point.

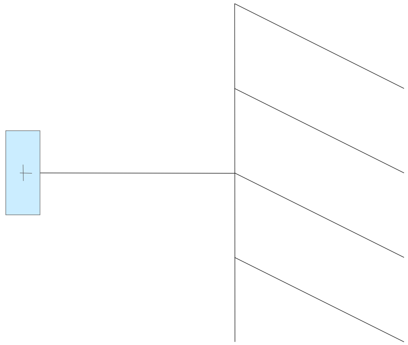


```
model Fixed "FixedPosition"

  Interfaces.Frame_a frame_a;
  parameter SI.Position x = 0
    "fixed x-position";
  parameter SI.Position y = 0
    "fixed y-position";
  parameter SI.Angle phi = 0
    "fixed angle";

equation
  frame_a.x = x;
  frame_a.y = y;
  frame_a.phi = phi;

end Fixed;
```

Fixed Component

Robotics and Mechatronics Centre

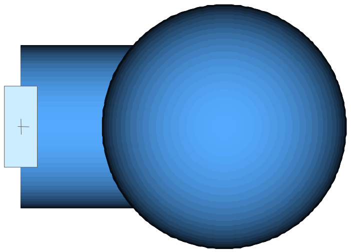
There is an alternative way to formulate this model. Using the vector notation of Modelica, we can unite the x- and y-positions to a 2-dimensional vector.



```
model Fixed "FixedPosition"

  Interfaces.Frame_a frame_a;
  parameter SI.Position r[2] = {0,0};
   "fixed x-position";
  parameter SI.Angle phi = 0
    "fixed angle";

equation
  {frame_a.x, frame_a.y} = r;
  frame_a.phi = phi;

end Fixed;
```

Body Component

- A little more elaborate is the body-component that represents a mass with inertia.

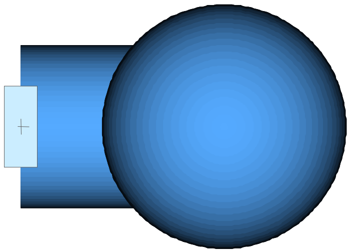


- Essentially, the model formulates Newton' s law.

```
model Body
  Interfaces.Frame_a frame_a;

  parameter SI.Mass m;
  parameter SI.Inertia I;

  SI.Force f[2];
  SI.Position r[2];
  SI.Velocity v[2];
  SI.Acceleration a[2];
  SI.AngularVelocity w;
  SI.AngularAcceleration z;

equation
  r = {frame_a.x, frame_a.y}
  v = der(r);
  w = der(frame_a.phi);

  a = der(v);
  z = der(w);


  f = {frame_a.fx, frame_a.fy};
  f = m*a;
  frame_a.t = I*z;
end Body
```

Body Component

- Since the gravitational force is dependent on the mass (m*g), it makes sense to compute right in the body model.



- A parameter for the gravitational acceleration is added and Newton's law is extended.

```
model Body
  Interfaces.Frame_a frame_a;

  parameter SI.Mass m;
  parameter SI.Inertia I;
  parameter SI.Acceleration[2] g={0,-9.81};
  SI.Force f[2];
  SI Position r[2];
  SI.Velocity v[2];
  SI.Acceleration a[2];
  SI.AngularVelocity w;
  SI.AngularAcceleration z;

equation
  r = {frame_a.x, frame_a.y}
  v = der(r);
  w = der(frame_a.phi);

  a = der(v);
  z = der(w);

  f = {frame_a.fx, frame_a.fy};
  f + m*g = m*a;
  frame_a.t = I*z;
end Body
```

Modelica's Vector Notation

- Vectors of variables, parameters or components can be declared by using rectangular brackets:

  `SI.Position r[2];` or `SI.Position[2] r;`

- A vector can be composed out of scalars by using curly braces:

  ```
  r = {0.2,13.4}
  ```

- Vectors can be added and subtracted and be multiplied by scalars:

  ```
  f + m*g = m*a
  ```

- Vectors can be multiplied with each other. This is the scalar product.

  ```
  v*e
  ```

Modelica's Matrix Notation

- Matrices of variables, parameters or components can be declared by rectangular brackets

```
Real R[2,2];
```
or
```
Real[2,2] R;
```

- A matrix can be expressed row-wise…

```
R = {{1,2},{3,4}}
```
or
```
R=[1,2;3,4]
```

- …or column-wise

```
R = [{1,3},{2,4}]
```

1	2
3	4

- Like with vectors, arithmetic operations can be performed on matrices. For instance, the matrix vector multiplication:

```
y = R*x;
```

Simulating the body model

- The body model contains 16 scalar variables and 13 scalar equation. There are 3 equation missing from connecting to the interface.

- Nevertheless, we can simulate the body model as a total system.

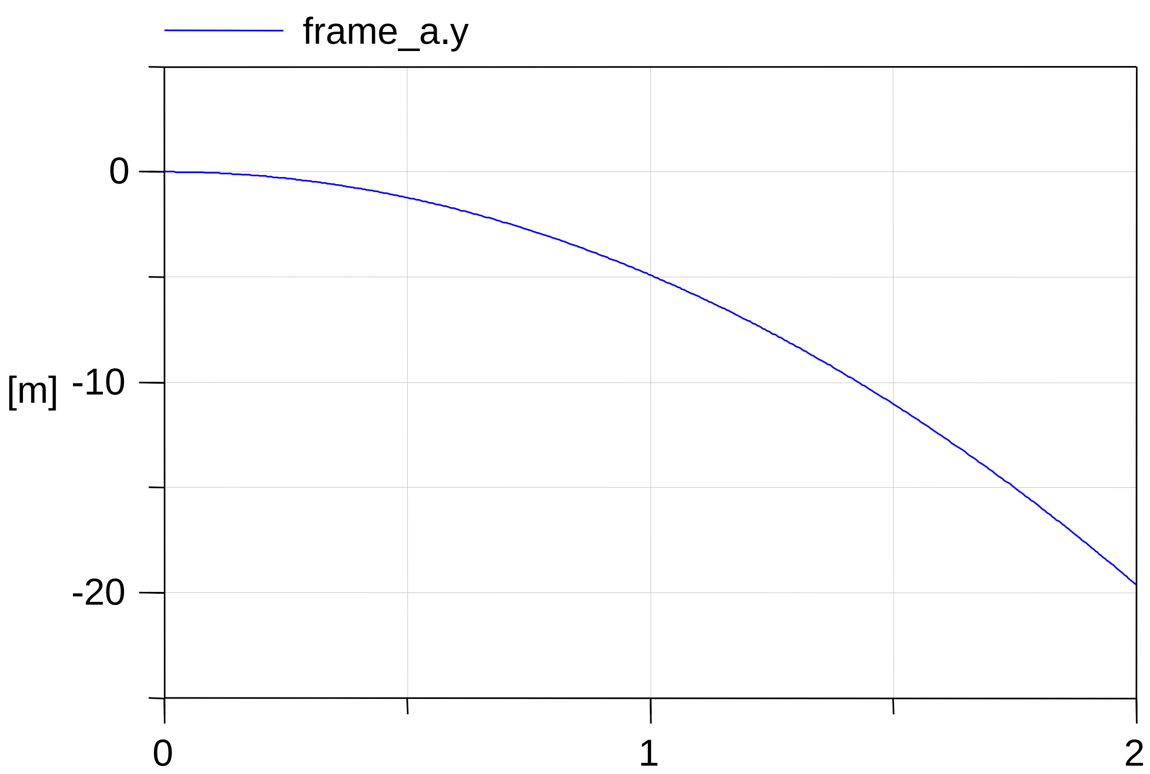
- This is possible, since for each connector that remains unconnected in the total system, all its flow variables are assumed to be zero.

- This means if we simulate the body model as total system, the following 3 equations are added to the system:

```
frame_a.fx = 0;
frame_a.fy = 0;
frame_a.t = 0;
```

Simulating the body model

- Here is the simulation result:



- It shows the parabolic descent of a body due to gravity acceleration.

Components with two Flanges

Robotics and Mechatronics Centre

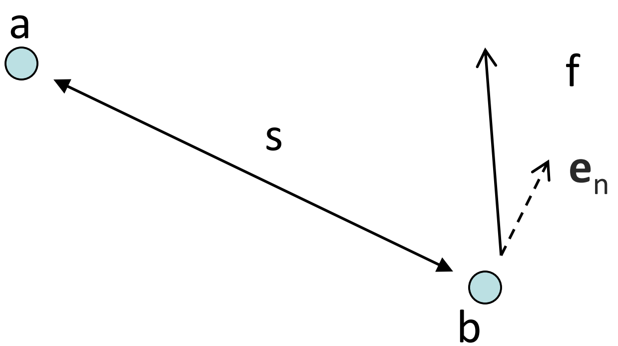
- Components that have two frames are little more difficult.
- Let us start by modeling a neutral component.
- The model itself is rather meaningless but it represents a good starting point for the design of any new component.

```
model Neutral
  Interfaces.Frame_a frame_a;
  Interfaces.Frame_b frame_b;






equation


  frame_a.fx = 0;
  frame_a.fy = 0;
  frame_a.t = 0;


  frame_a.fx + frame_b.fx = 0;
  frame_a.fy + frame_b.fy = 0;
  frame_a.t
  + frame_b.t
  - (frame_b.x - frame_a.x)*frame_b.fy
  + (frame_b.y - frame_a.y)*frame_b.fx
  = 0;
end Neutral
```

Components with two Flanges

Robotics and Mechatronics Centre

- The model imposes no constraints on the positions.
- This component has two frames, but exhibits no effect.
- The balance equations for the forces contains the lever principle.



$\tau = \mathbf{f} \cdot \mathbf{e}_n \cdot s = \mathbf{f} \cdot (\mathbf{e}_n \cdot s)$

$\tau = (fx, fy) \cdot (-sy, sx)$

```
model Neutral
  Interfaces.Frame_a frame_a;
  Interfaces.Frame_b frame_b;


equation


  frame_a.fx = 0;
  frame_a.fy = 0;
  frame_a.t = 0;


  frame_a.fx + frame_b.fx = 0;
  frame_a.fy + frame_b.fy = 0;
  frame_a.t
  + frame_b.t
  + (frame_b.x - frame_a.x)*frame_b.fy
  - (frame_b.y - frame_a.y)*frame_b.fx
  = 0;
end Neutral
```

Components with two Flanges

Robotics and Mechatronics Centre

Guidelines:

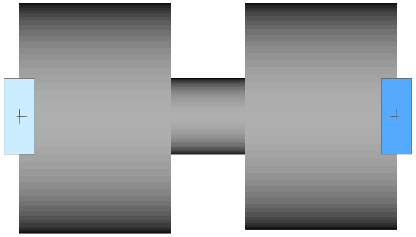
- For each positional constraint we add, we have to remove the corresponding force equation.
- For each variable that we add, we have to add an equation
- Finally, we may be able to simplify the balance equations.

```
model Revolute
  Interfaces.Frame_a frame_a;
  Interfaces.Frame_b frame_b;






equation



  frame_a.fx = 0;
  frame_a.fy = 0;
  frame_a.t = 0;


  frame_a.fx + frame_b.fx = 0;
  frame_a.fy + frame_b.fy = 0;
  frame_a.t
  + frame_b.t
  + (frame_b.x - frame_a.x)*frame_b.fy
  - (frame_b.y - frame_a.y)*frame_b.fx
  = 0;
end Revolute
```

Revolute Joint

Robotics and Mechatronics Centre

Let us start with the revolute joint:

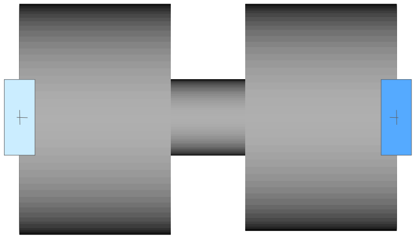


```
model Revolute
  Interfaces.Frame_a frame_a;
  Interfaces.Frame_b frame_b;






equation



  frame_a.fx = 0;
  frame_a.fy = 0;
  frame_a.t = 0;


  frame_a.fx + frame_b.fx = 0;
  frame_a.fy + frame_b.fy = 0;
  frame_a.t
  + frame_b.t
  + (frame_b.x - frame_a.x)*frame_b.fy
  - (frame_b.y - frame_a.y)*frame_b.fx
  = 0;
end Revolute
```

Revolute Joint

Robotics and Mechatronics Centre

Let us start with the revolute joint:

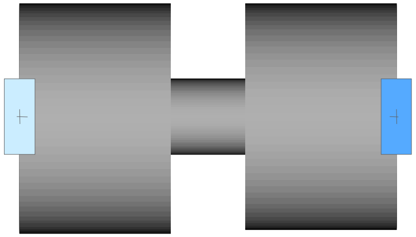


- The translational positions of a and b are equal. (2 constraints)
- No torque can act on the joint.

```
model Revolute
  Interfaces.Frame_a frame_a;
  Interfaces.Frame_b frame_b;

equation

  frame_a.fx = 0 replaced by
  frame_a.x = frame_b.x;
  frame_a.fy = 0 replaced by
  frame_a.y = frame_b.y;
  frame_a.t = 0;

  frame_a.fx + frame_b.fx = 0;
  frame_a.fy + frame_b.fy = 0;
  frame_a.t
  + frame_b.t
  + (frame_b.x - frame_a.x)*frame_b.fy
  - (frame_b.y - frame_a.y)*frame_b.fx
  = 0;

end Revolute
```

Revolute Joint

Let us start with the revolute joint:

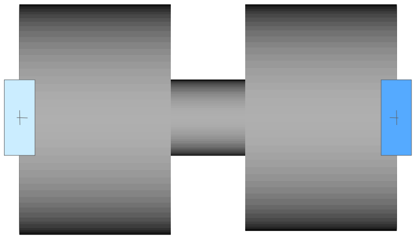


- The translational positions of a and b are equal. (2 constraints)
- No torque can act on the joint.
- The lever principle is redundant here…
- That's it! …actually

```
model Revolute
  Interfaces.Frame_a frame_a;
  Interfaces.Frame_b frame_b;

equation

  frame_a.fx = 0 replaced by
  frame_a.x = frame_b.x;
  frame_a.fy = 0 replaced by
  frame_a.y = frame_b.y;
  frame_a.t = 0;

  frame_a.fx + frame_b.fx = 0;
  frame_a.fy + frame_b.fy = 0;
  frame_a.t  + frame_b.t = 0;

end Revolute
```

Revolute Joint

Let us start with the revolute joint:



- For completeness, we'd like to add two differential equations for the angle, the angular velocity and its acceleration.
- After all, these variables are of interest.
- We can now use the joint in order to express motion.
- It also helps with initialization.

```
model Revolute
  Interfaces.Frame_a frame_a;
  Interfaces.Frame_b frame_b;

  SI.Angle phi
  SI.AngularVelocity w;
  SI.AngularAcceleration z;


equation
  frame_a.phi + phi = frame_b.phi;
  w = der(phi);
  z = der(w);

  frame_a.x = frame_b.x;
  frame_a.y = frame_b.y;
  frame_a.t = 0;


  frame_a.fx + frame_b.fx = 0;
  frame_a.fy + frame_b.fy = 0;
  frame_a.t  + frame_b.t = 0;


end Revolute
```

Fixed Translation

Robotics and Mechatronics Centre

Let us proceed with a rigid rod:

```
model FixedTranslation
  Interfaces.Frame_a frame_a;
  Interfaces.Frame_b frame_b;

equation

  frame_a.fx = 0;
  frame_a.fy = 0;
  frame_a.t = 0;

  frame_a.fx + frame_b.fx = 0;
  frame_a.fy + frame_b.fy = 0;
  frame_a.t
  + frame_b.t
  + (frame_b.x - frame_a.x)*frame_b.fy
  - (frame_b.y - frame_a.y)*frame_b.fx
  = 0;
end FixedTranslation
```

Fixed Translation

Let us proceed with a rigid rod:

- The length and direction of the rod is determined by the parameter vector r
- This vector is resolved w.r.t the body (coordinate) system.

```
model FixedTranslation
  Interfaces.Frame_a frame_a;
  Interfaces.Frame_b frame_b;

  parameter SI.Length r[2];

equation


  frame_a.fx = 0;
  frame_a.fy = 0;
  frame_a.t = 0;


  frame_a.fx + frame_b.fx = 0;
  frame_a.fy + frame_b.fy = 0;
  frame_a.t
  + frame_b.t
  + (frame_b.x - frame_a.x)*frame_b.fy
  - (frame_b.y - frame_a.y)*frame_b.fx
  = 0;

end FixedTranslation
```

Fixed Translation

Let us proceed with a rigid rod:

- We need to transform the vector r into the inertial frame r0 by a 2D rotation:

$$\begin{pmatrix} r0_x \\ r0_y \end{pmatrix} = \begin{pmatrix} \cos(\varphi) & -\sin(\varphi) \\ \sin(\varphi) & \cos(\varphi) \end{pmatrix} \begin{pmatrix} r_x \\ r_y \end{pmatrix}$$

```
model FixedTranslation
  Interfaces.Frame_a frame_a;
  Interfaces.Frame_b frame_b;
  parameter SI.Length r[2];
  SI.Distance r0[2];
  Real R[2,2];

equation

  R = {{cos(frame_a.phi), -sin(frame_a.phi)},
       {sin(frame_a.phi),cos(frame_a.phi)}}

  r0 = R*r;

  frame_a.fx = 0;
  frame_a.fy = 0;
  frame_a.t = 0;


  frame_a.fx + frame_b.fx = 0;
  frame_a.fy + frame_b.fy = 0;
  frame_a.t  + frame_b.t
  + (frame_b.x - frame_a.x)*frame_b.fy
  - (frame_b.y - frame_a.y)*frame_b.fx
  = 0;

end FixedTranslation
```

Fixed Translation

Let us proceed with a rigid rod:

- Finally we can use r0[1], r0[2] to formulate the constraint equations and simplify the lever principle.

```
model FixedTranslation
  Interfaces.Frame_a frame_a;
  Interfaces.Frame_b frame_b;
  parameter SI.Length r[2];
  SI.Distance r0[2];
  Real R[2,2];

equation

  R = {{cos(frame_a.phi), -sin(frame_a.phi)},
       {sin(frame_a.phi),cos(frame_a.phi)}}

  r0 = R*r;

  frame_a.x + r0[1] = frame_b.x;
  frame_a.y + r0[2] = frame_b.y;
  frame_a.phi = frame_b.phi;

  frame_a.fx + frame_b.fx = 0;
  frame_a.fy + frame_b.fy = 0;
  frame_a.t  + frame_b.t
  + r0*{frame_b.fy,-frame_b.fx} = 0;

end FixedTranslation
```

Fixed Translation

Robotics and Mechatronics Centre

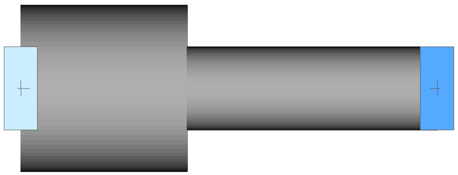
Let us proceed with a rigid rod:

- Finally we can use r0[1], r0[2] to formulate the constraint equations and simplify the lever principle.

```
model FixedTranslation
  Interfaces.Frame_a frame_a;
  Interfaces.Frame_b frame_b;
  parameter SI.Length r[2];
  SI.Distance r0[2];
  Real R[2,2];

equation

  R = {{cos(frame_a.phi), -sin(frame_a.phi)},
       {sin(frame_a.phi),cos(frame_a.phi)}}

  r0 = R*r;

  frame_a.x + r0[1] = frame_b.x;
  frame_b.y + r0[2] = frame_b.y;
  frame_a.phi = frame_b.phi;

  frame_a.fx + frame_b.fx = 0;
  frame_a.fy + frame_b.fy = 0;
  frame_a.t  + frame_b.t
  + r0*{frame_b.fy,-frame_b.fx} = 0;

end FixedTranslation
```

Prismatic Joint

The prismatic joint represents a rod with variable length:

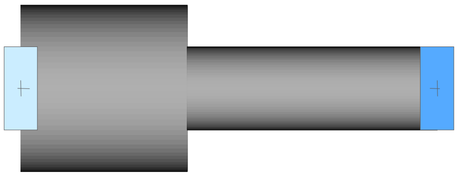


- We can use the FixedTranslation model as a template.
- The final parameter e represents a normalized version of r;
- The variable s shall represent the length of the rod.
- Hence r0 = R*e*s;

```
model Prismatic
  Interfaces.Frame_a frame_a;
  Interfaces.Frame_b frame_b;
  parameter SI.Distance r[2];
  final parameter Real e[2]
    = r/sqrt(r*r);
  SI.Distance s;
  SI.Distance r0[2];
  Real R[2,2];

equation
  R = {{cos(frame_a.phi), -sin(frame_a.phi)},
       {sin(frame_a.phi),cos(frame_a.phi)}}
  r0 = R*e*s;
  frame_a.x + r0[1] = frame_b.x;
  frame_b.y + r0[2] = frame_b.y;
  frame_a.phi = frame_b.phi;

  frame_a.fx + frame_b.fx = 0;
  frame_a.fy + frame_b.fy = 0;
  frame_a.t  + frame_b.t
  + r0*{frame_b.fy,-frame_b.fx} = 0;

end Prismatic
```

Prismatic Joint

The prismatic joint represents a rod with variable length:

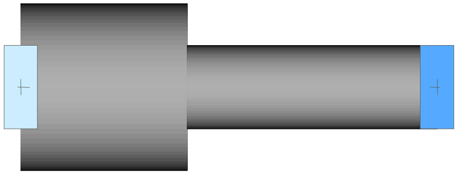


- Since we are relieving one positional constraint by adding the variable s, we have to add one force equation.
- No force can act in direction of the prismatic joint.
- This direction resovled in the inertial system is R*e;

```
model Prismatic
  Interfaces.Frame_a frame_a;
  Interfaces.Frame_b frame_b;
  parameter SI.Distance r[2];
  final parameter Real e[2]
    = r/sqrt(r*r);
  SI.Distance s;
  SI.Distance r0[2];
  Real R[2,2];

equation
  R = {{cos(frame_a.phi), -sin(frame_a.phi)},
       {sin(frame_a.phi),cos(frame_a.phi)}}
  r0 = R*e*s;
  frame_a.x + r0[1] = frame_b.x;
  frame_b.y + r0[2] = frame_b.y;
  frame_a.phi = frame_b.phi;

  {frame_a.fx,frame_a.fy}*(R*e) = 0;
  frame_a.fx + frame_b.fx = 0;
  frame_a.fy + frame_b.fy = 0;
  frame_a.t  + frame_b.t
  + r0*{frame_b.fy,-frame_b.fx} = 0;

end Prismatic
```

Prismatic Joint

The prismatic joint represents a rod with variable length:



- As for the revolute joint, we would like to add the derivatives v and a.

```
model Prismatic
  Interfaces.Frame_a frame_a;
  Interfaces.Frame_b frame_b;
  parameter SI.Distance r[2];
  final parameter Real e[2]
    = r/sqrt(r*r);
  SI.Distance s;
  SI.Distance r0[2];
  Real R[2,2];
  SI.Velocity v;
  SI.Acceleration a;
equation
  v = der(s);
  a = der(v);
  R = {…};
  r0 = R*e*s;
  frame_a.x + r0[1] = frame_b.x;
  frame_b.y + r0[2] = frame_b.y;
  frame_a.phi = frame_b.phi;
  {frame_a.fx,frame_a.fy}*(R*e) = 0;
  frame_a.fx + frame_b.fx = 0;
  frame_a.fy + frame_b.fy = 0;
  frame_a.t  + frame_b.t
  + r0*{frame_b.fy,-frame_b.fx} = 0;
end Prismatic
```

Damper

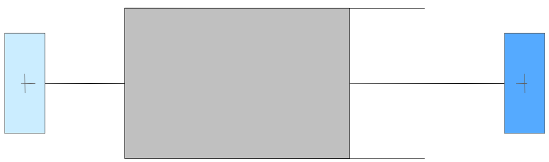
Let us conclude by modeling a damper:

- First of all, the damper does not impose any positional constraints.
- The damping force only acts alongside the damping direction.
- So the lever principle does not apply.

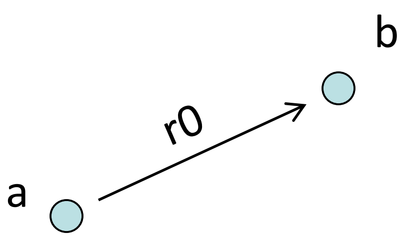
```
model Damper
  Interfaces.Frame_a frame_a;
  Interfaces.Frame_b frame_b;






equation



  frame_a.fx = 0;
  frame_a.fy = 0;
  frame_a.t = 0;




  frame_a.fx + frame_b.fx = 0;
  frame_a.fy + frame_b.fy = 0;
  frame_a.t + frame_b.t = 0;


end Damper
```

Damper

Let us conclude by modeling a damper:



- The direction of the damping force is represented by the variable vector r0.



- We see that Modelica supports also vectors (similar to Matlab)

```
model Damper
  Interfaces.Frame_a frame_a;
  Interfaces.Frame_b frame_b;

  SI.Distance[2] r0;


equation

  frame_a.x + r0[1] = frame_b.x;
  frame_a.y + r0[2] = frame_b.y;

  frame_a.fx = 0;
  frame_a.fy = 0;
  frame_a.t = 0;

  frame_a.fx + frame_b.fx = 0;
  frame_a.fy + frame_b.fy = 0;
  frame_a.t + frame_b.t = 0;


end Damper
```

Damper

Robotics and Mechatronics Centre

Let us conclude by modeling a damper:

- The direction e0 is then the normalized version of r0.
- The built-in function contains work-around for the case that r0 = 0.

```
model Damper
  Interfaces.Frame_a frame_a;
  Interfaces.Frame_b frame_b;

  SI.Length[2] r0;
  Real[2] e0;

equation

  frame_a.x + r0[1] = frame_b.x;
  frame_a.y + r0[2] = frame_b.y;
  e0= Modelica.Math.Vectors.normalize(r0);

  frame_a.fx = 0;
  frame_a.fy = 0;
  frame_a.t = 0;

  frame_a.fx + frame_b.fx = 0;
  frame_a.fy + frame_b.fy = 0;
  frame_a.t + frame_b.t = 0;

end Damper
```

Damper

Let us conclude by modeling a damper:

- v0 represents the relative velocity of the two frames.

- v is then the velocity in direction of the damper e0.

  v = v_d*e0;

```
model Damper
  Interfaces.Frame_a frame_a;
  Interfaces.Frame_b frame_b;
  SI.Length[2] r0;
  Real[2] e0;
  SI.Velocity v0[2];
  SI.Velocity v;

equation
  frame_a.x + r0[1] = frame_b.x;
  frame_a.y + r0[2] = frame_b.y;
  e0= Modelica.Math.Vectors.normalize(r0);
  v0 = der(r0);
  v = v0*e0;

  frame_a.fx = 0;
  frame_a.fy = 0;
  frame_a.t = 0;

  frame_a.fx + frame_b.fx = 0;
  frame_a.fy + frame_b.fy = 0;
  frame_a.t + frame_b.t = 0;

end Damper
```

Damper

Robotics and Mechatronics Centre

Let us conclude by modeling a damper:

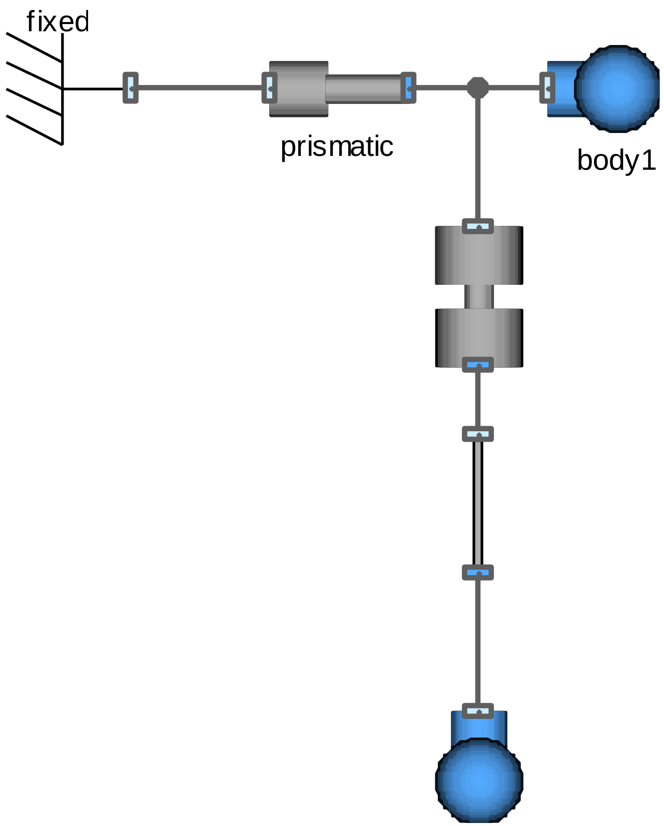
- f is the damping force acting in direction e0.
- It is proportional to the velocity. This is defined by the damping coefficient d:

  f = -d*v;

```
model Damper
  Interfaces.Frame_a frame_a;
  Interfaces.Frame_b frame_b;
  parameter SI.DampingConstant d;
  SI.Length[2] r0;
  Real[2] e0;
  SI.Velocity v0[2];
  SI.Velocity v;
  SI.Force f;
equation
  frame_a.x + r0[1] = frame_b.x;
  frame_a.y + r0[2] = frame_b.y;
  e0= Modelica.Math.Vectors.normalize(r0);
  v0 = der(r0);
  v = v0*e0;
  f = -d*v;
  frame_a.fx = e0[1] * f;
  frame_a.fy = e0[2] * f;
  frame_a.t = 0;
  frame_a.fx + frame_b.fx = 0;
  frame_a.fy + frame_b.fy = 0;
  frame_a.t + frame_b.t = 0;
end Damper;
```

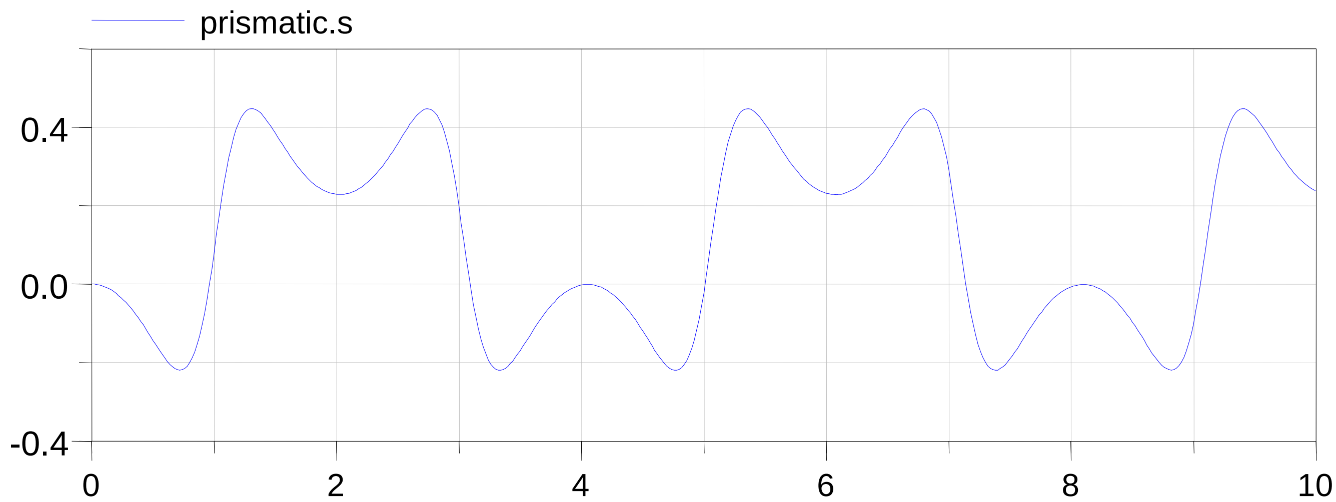
Crane Crab

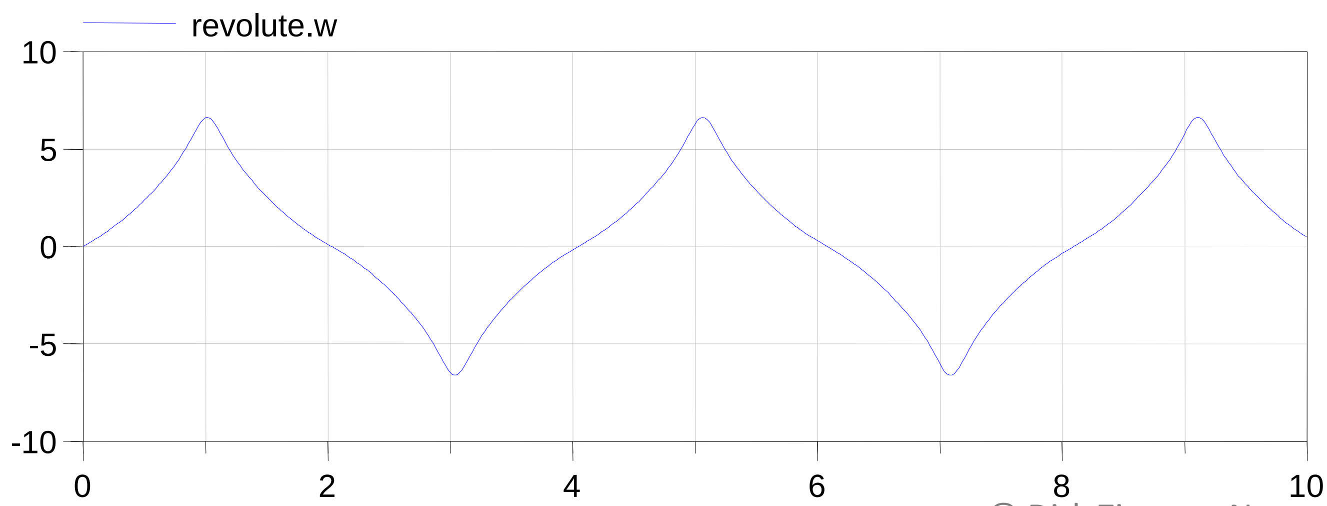
- Finally, we can model the crane crab:



Crane-Crab Results

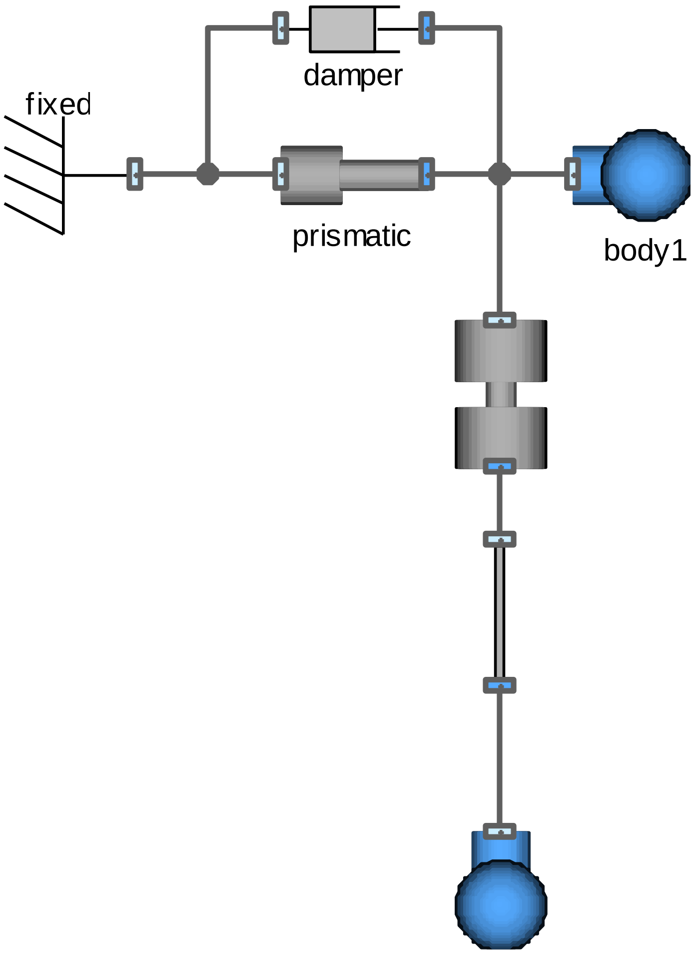
Robotics and Mechatronics Centre





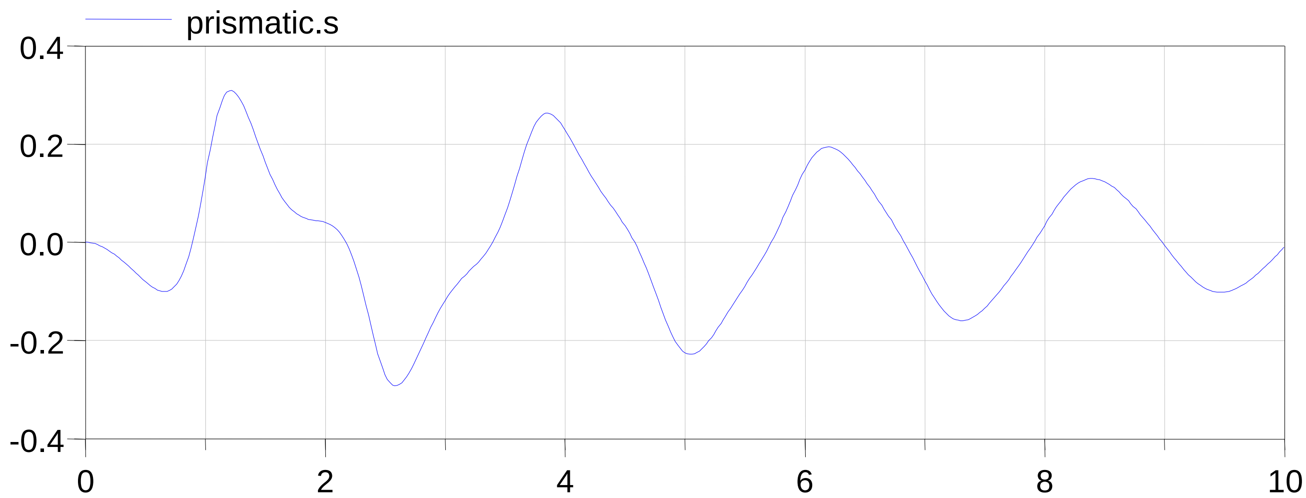
Crane Crab (Damped)

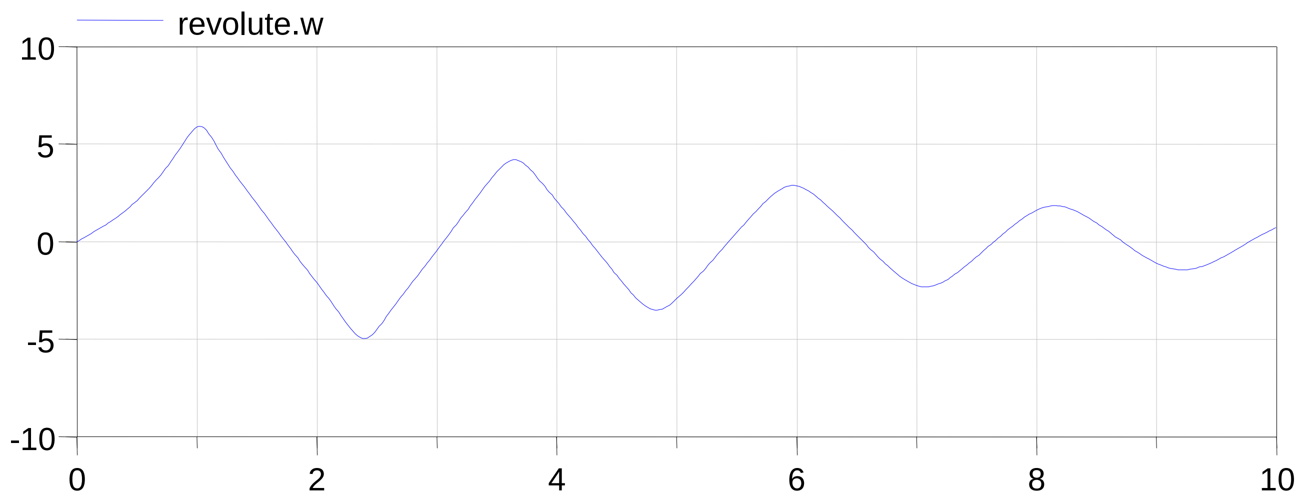
- Here a damped version:



Crane-Crab (Damped) Results

Robotics and Mechatronics Centre





Visualization

Robotics and Mechatronics Centre

- Looking at plots of mechanical systems isn't that exciting.
- We would like to have a 3D animation of our system.
- Fortunately, Dymola provides an internal support for this.
- We can add elements from the MultiBody library in order to visualize our components.

Visualization: Fixed Translation

TUM + DLR

Robotics and Mechatronics Centre

Let us visualize the fixed translation:

- To this end, we have to add the general visualization component: MB.Visualizers.Advanced.Shape

- We have to convert our 2D-data into 3D-vectors.

```
model FixedTranslation
 Interfaces.Frame_a frame_a;
  Interfaces.Frame_b frame_b;
  parameter SI.Length r[2];
  SI.Distance r0[2];
  Real R[2,2];
  final parameter SI.Length l = sqrt(r*r);

  MB.Visualizers.Advanced.Shape cylinder(
    shapeType="cylinder",
    color={128,128,128},
    specularCoefficient=0.5,
    length=l, width=0.1, height=0.1,
    lengthDirection={r0/l,r0/l,0},
    widthDirection={0,0,1},
    r_shape={0,0,0},
    r={frame_a.x,frame_a.y,0},
    R=MB.Frames.nullRotation());

equation
  …
end FixedTranslation
```

Visualization: Fixed Translation

Robotics and Mechatronics Centre

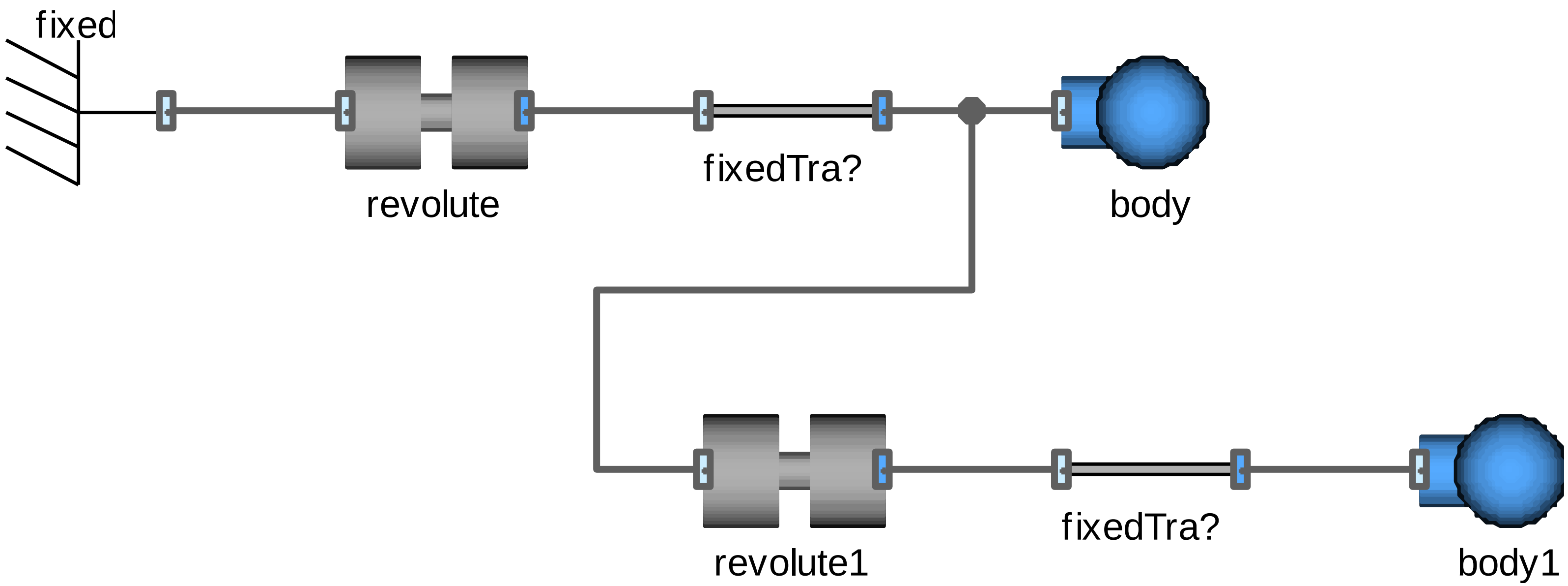
Let us visualize the fixed translation:

- Since the animation shall only be optional, we make this component conditional.
- Conditional components can only be accessed in a very limited way. So use this tool moderately.

```
model FixedTranslation
 Interfaces.Frame_a frame_a;
  Interfaces.Frame_b frame_b;
  parameter SI.Length r[2];
  SI.Distance r0[2];
  Real R[2,2];
  final parameter SI.Length l = sqrt(r*r);
  parameter Boolean animation = true;

  MB.Visualizers.Advanced.Shape cylinder(
    shapeType="cylinder",
    color={128,128,128},
    specularCoefficient=0.5,
    length=l, width=0.1, height=0.1,
    lengthDirection={sx0/l,sy0/l,0},
    widthDirection={0,0,1},
    r_shape={0,0,0},
    r={frame_a.x,frame_a.y,0},
    R=MB.Frames.nullRotation())
     if animation;

equation
  …
end FixedTranslation
```

Double Pendulum

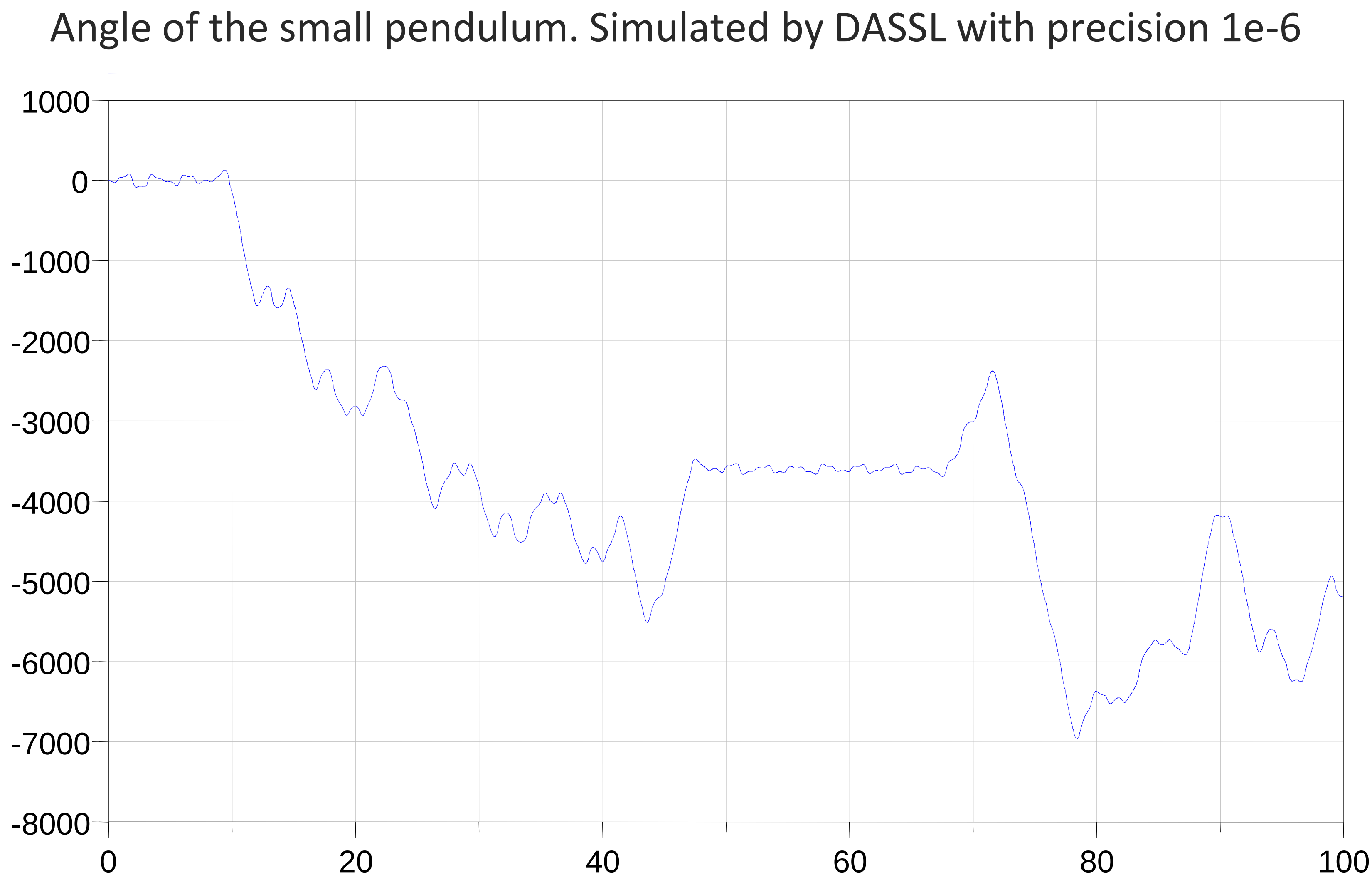
- A seemingly simple system is the double pendulum



- Let us look at the angle of the second revolute joint.

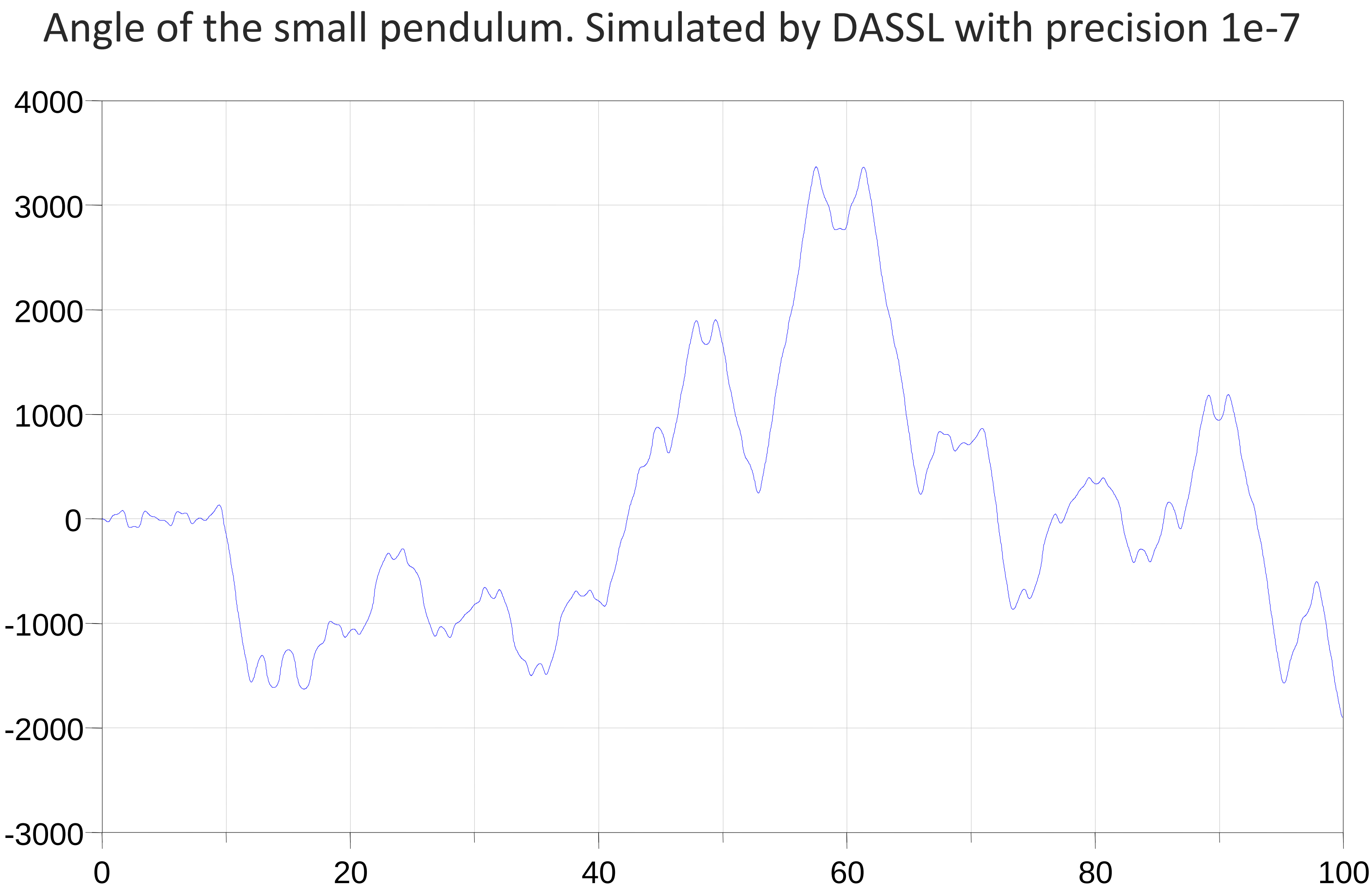
Double Pendulum

Robotics and Mechatronics Centre



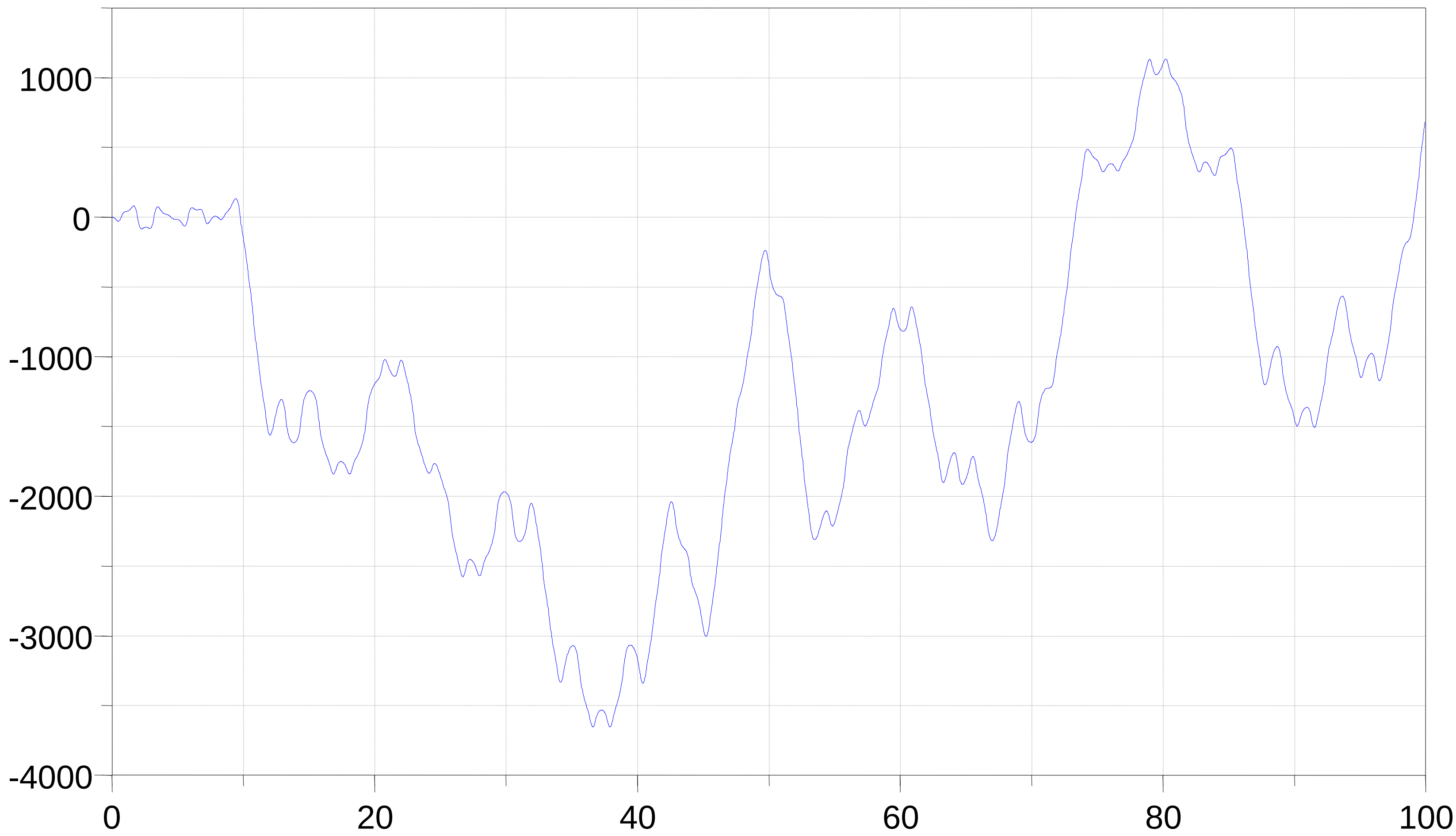
Double Pendulum

Robotics and Mechatronics Centre



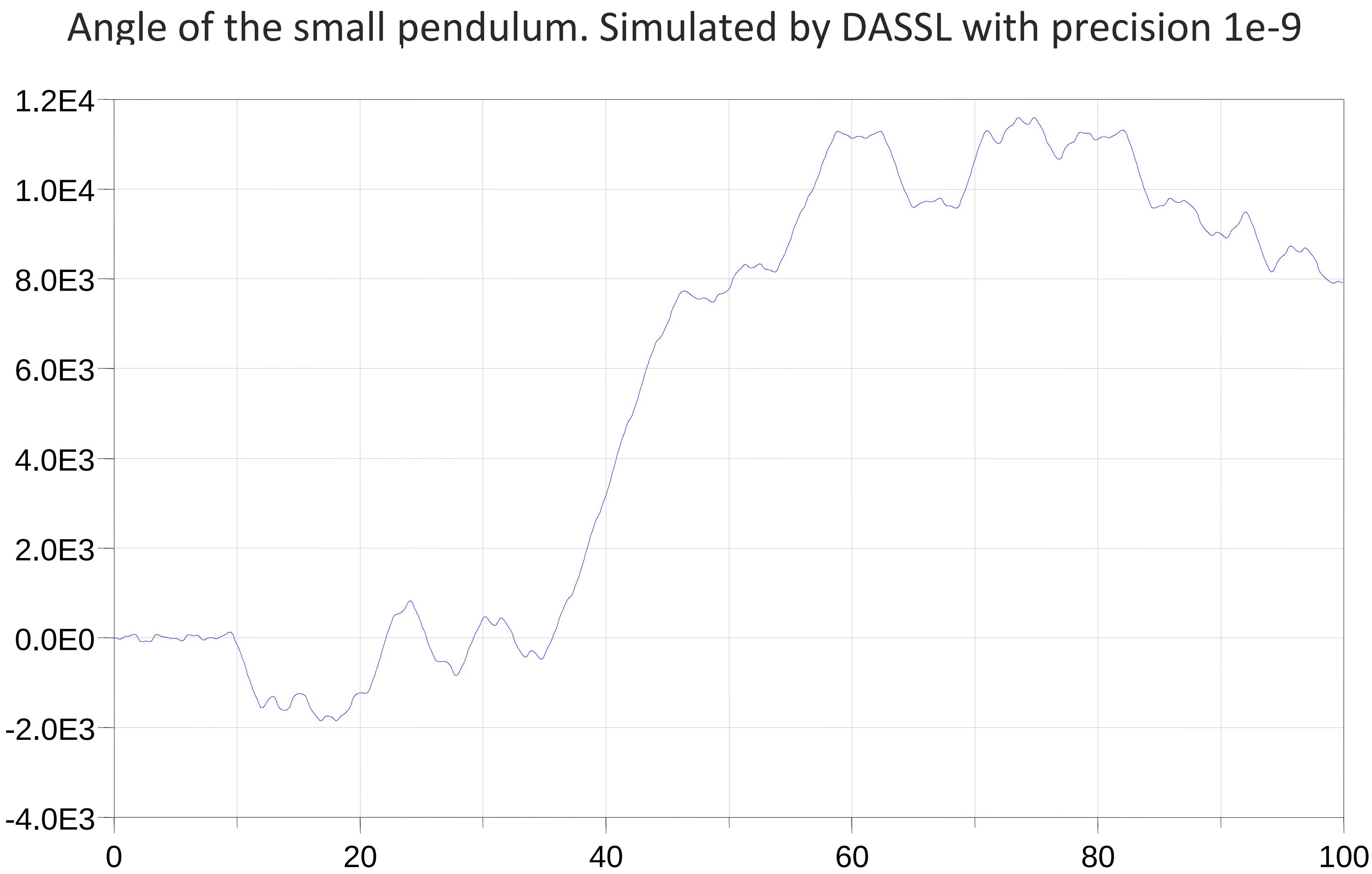
Double Pendulum

Angle of the small pendulum. Simulated by DASSL with precision 1e-8



Double Pendulum

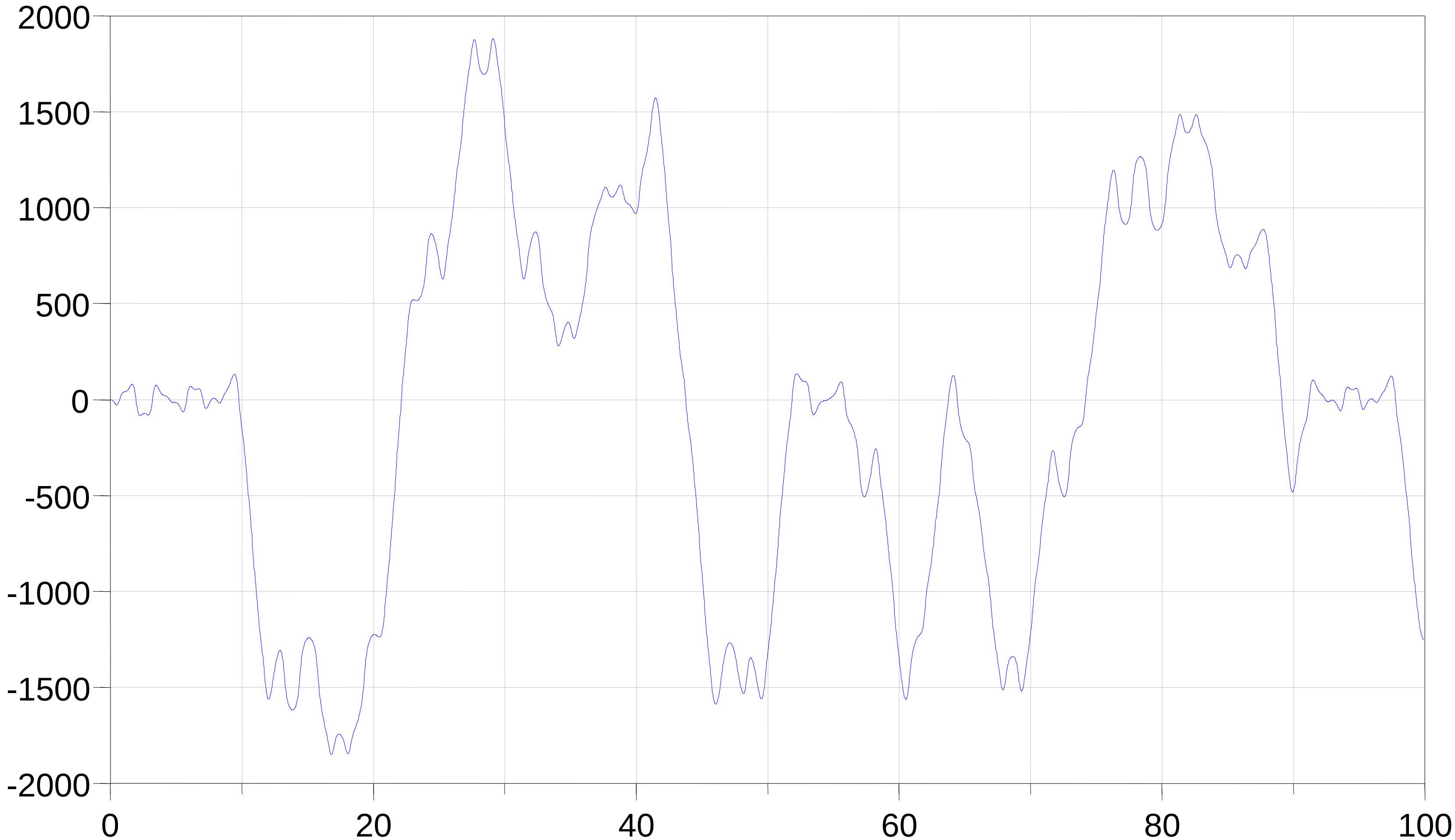
Robotics and Mechatronics Centre



Double Pendulum

Robotics and Mechatronics Centre

Angle of the small pendulum. Simulated by DASSL with precision 1e-10



Double Pendulum

- The simulation does not converge no matter what precision we apply. We have no f*#?ing clue what the state of our system is at t = 100.
- The double pendulum is a chaotic system.
- The upright resting position of the second pendulum represents a bifurcation point.
- During simulation, the system will almost inevitable come close to this bifurcation point. Hence the system is extremely sensitive to its initial state.
- Too sensitive to enable any kind of reliable prediction.

Summary

Robotics and Mechatronics Centre

- We designed the connector of a planar-mechanical library
- We designed the first component.
- We developed component by starting from a neutral pseudo-component.
- We learned about arrays/vectors in Dymola.
- We assembled the crane crab and a double pendulum.
- We were confronted with a chaotic system.

Questions ?