DynaCoM Manual

Nahuel Villa

December 15, 2022

Contents

1 The Contact Classes

The contact classes are based on the virtual class `ContactBase` and can be instantiated under the form of a `ContactPoint` which interacts with the environment by a 3D force, or a `Contact6D` that can produce a full 6D wrench.

The contacts contain information about the feasibility of forces that they can produce and the relative prioritization of the magnitude of such forces.

In c++ it can be included as

```
#include <dynacom/contact_base.hpp>
#include <dynacom/contact_point.hpp>
#include <dynacom/contact6d.hpp>
```

and in python imported it as

```
from dynacom import ContactBase, ContactPoint, Contact6D
```

1.1 Instantiation

Functions:

```
ContactPoint::ContactPoint()
ContactPoint::initialize()
Contact6D::Contact6D()
Contact6D::initialize()
```

Instantiating a `ContactPoint` and `Contact6D` require the following structs of settings respectively:

```
ContactPointSettings.mu           |  double
                    .weights      |  Eigen::Vector3d
                    .frame_name   |  std::string

   Contact6DSettings.mu           |  double
                    .gu           |  double
                    .half_length  |  double
                    .half_width   |  double
                    .weights      |  Eigen::Vector6d
                    .frame_name   |  std::string
```

which can be used as an input in the constructor, or afterwards in the function `initialize()`.

1.2 Modifying Contact Settings

Functions:

`ContactPoint::setMu()`
`ContactPoint::setForceWeights()`
`Contact6D::setMu()`
`Contact6D::setGu()`
`Contact6D::setForceWeights()`
`Contact6D::setTorqueWeights()`
`Contact6D::setSurfaceHalfWidth()`
`Contact6D::setSurfaceHalfLength()`

Most of the contact settings can be modified after the initialization by using the specific setters.

1.3 Contact Matrices

Functions:

`ContactBase::uni_A()`	`ContactBase::uni_b()`
`ContactBase::fri_A()`	`ContactBase::fri_b()`
`ContactBase::reg_A()`	`ContactBase::reg_b()`
`ContactBase::ne_A()`	

Each contact contains the matrices related to its own feasibility and prioritization. Such matrices are attributes of the `ContactBase` and adopt different forms according to the specific kind of contact used, either `ContactPoint` or `Contact6D`.

All the matrices `_b` are currently set to zero, and in both cases `reg_A` is a vector containing the 3 or 6 weights accordingly.

In the **`ContactPoint`** the unilaterality and friction cone matrices are:

$$\texttt{uni\_A} = \begin{bmatrix} 0 & 0 & -1 \end{bmatrix}, \qquad \texttt{fri\_A} = \begin{bmatrix} 1 & 0 & -\mu \\ 0 & 1 & -\mu \\ -1 & 0 & -\mu \\ 0 & -1 & -\mu \end{bmatrix}.$$

And the `ne_A` are the three first rows of the adjoint matrix transforming the force from the local contact frame to the frame of the CoM.

In the **`Contact6D`**, the unilaterality and friction cone matrices are:

$$\texttt{uni\_A} = \begin{bmatrix} 0 & 0 & -1 & 0 & 0 & 0 \\ & & -d_x & 0 & -1 & \\ \vdots & \vdots & -d_y & 1 & 0 & \vdots \\ & & -d_x & 0 & 1 & \\ 0 & 0 & -d_y & -1 & 0 & 0 \end{bmatrix}, \quad \texttt{fri\_A} = \begin{bmatrix} 1 & 0 & -\mu & 0 & 0 & 0 \\ 0 & 1 & -\mu & & & \vdots \\ -1 & 0 & -\mu & \vdots & \vdots & \\ 0 & -1 & -\mu & & & 0 \\ 0 & 0 & -\gamma & & & 1 \\ 0 & 0 & -\gamma & 0 & 0 & -1 \end{bmatrix}.$$

And `ne_A` is the adjoint matrix transforming the wrench from the local contact frame to the frame of the CoM.

2 The DynaCoM Class

This class is in charged to compute the centroidal wrench required to perform certain motion of the robot and to distribute such wrench optimally among the active contacts of the robot.

In c++ it can be included as

```
#include <dynacom/dyna_com.hpp>
```

and in python import it as

```
from dynacom import DynaCoM
```

2.1 Instantiation

Functions:

```
DynaCoM::DynaCoM()
DynaCoM::initialize()
```

The instantiation of the `DynaCoM class` requires a `struct` called `DynaCoMSettings` containing the address of the `URDF` file that describes the model of the robot.

We can instantiate the `DynaCoM` using the `DynaCoMSettings` as a parameter or by default without parameters, it incorporates the settings later with the method `initialize()`.

2.2 Computation of the Centroidal Wrench

Function:

`DynaCoM::computeDynamics()`

This computation is based on the Newton and Euler equations:

$$\sum_k f_k = m\ddot{c} - mg - f_e \tag{1}$$

$$\sum_k r_k \times f_k = \dot{L} - \tau_e, \tag{2}$$

where the robot weight mg is obtained from the `pinnochio::model`, the known or expected external wrench f_e, τ_e (not supporting wrench, in CoM frame) are provided by the user and the variation of the linear $m\ddot{c}$ and angular $\dot{L}$ momentum are obtained from the function

`pinocchio::computeCentroidalMomentumTimeVariation`$(q, \dot{q}, \ddot{q})$,

based on the inputs `position` (q), `velocity` $(\dot{q})$ and `acceleration` $(\ddot{q})$.

The supporting wrench $\sum_k f_k$ and $\sum_k r_k \times f_k$, expressed in the frame of the CoM, can be accessed by the getter methods:

$$\sum_k f_k = \texttt{getGroundCoMForce()}; \tag{3}$$

$$\sum_k r_k \times f_k = \texttt{getGroundCoMTorque()}; \tag{4}$$

2.2.1 Computing the Center of Pressure

We compute the Center of Pressure (CoP) assuming that the ground is flat and horizontal (`flatHorizontalGround = true`) or without assumptions (`flatHorizontalGround = false`). In all cases, the CoP is always computed on a plane ^{x,y} perpendicular to the gravity.

case `flatHorizontalGround = true`: In this case the CoP p in x and y coordinates can be computed directly from the centroidal wrench:

$$p^{x,y} = c^{x,y} + \frac{(S(\sum_k r_k \times f_k)^{x,y} - (\sum_k f_k)^{x,y} CoM^z)}{(\sum_k f_k)^z} \tag{5}$$

where $S = \left[\begin{smallmatrix} 0 & -1 \\ 1 & 0 \end{smallmatrix}\right]$ is a $\frac{\pi}{2}$ rotation matrix.

case `flatHorizontalGround` = `false`: Without assumptions, we need to specify where are the robot supports as contact placements, and we must distribute the centroidal wrench among the contacts (*learn more about the force distribution in the section 2.3.2*).

$$p^{x,y} = \frac{S\,(\sum_k \tau_k^o)^{x,y}}{(\sum_k f_k)^z}, \tag{6}$$

where τ_k^o is the world frame torque produced by the k-th contact.

In a future release, we plan to remove the assumption of Horizontal ground, by replacing the flag `flatHorizontalGround` by `flatGround`. Such computation would require additionally a vector normal to the ground plane.

2.2.2 Computing the non-linearity

Function:

`DynaCoM::computeNL()`

Once the CoP is computed, we can obtain the "non-linearity" n, defined as the difference between the CoP and the Virtual Repellent Point VRP v defined as the base of an equivalent Linear Inverted Pendulum (LIP) with the same CoM motion shown by the robot. This requires the time constant of the LIP ω.

$$v^{x,y} = c^{x,y} - \ddot{c}^{x,y}/\omega^2 \tag{7}$$

$$n^{x,y} = p^{x,y} - v^{x,y}. \tag{8}$$

2.3 Computation of Contact Forces

The `Contact class` is described in the next section. Here we focus on how the `DynaCoM` deals with the contacts.

2.3.1 Contact Management

Functions:

`DynaCoM::addContact6d()`
`DynaCoM::removeContact6d()`
`DynaCoM::activateContact6d()`
`DynaCoM::deactivateContact6d()`.

The `DynaCoM` gathers all the known contacts in a map, called `known_contacts`, relating the assigned name of each contact with a `shared_ptr` to the contact. Moreover, it has a `vector<string>` with the names of all active contacts.

Contacts are incorporated or removed from the `known_contacts` with the methods `DynaCoM::addContact6d()` and `DynaCoM::removeContact6d()`. When a contact is added to the map of known contacts, the frame where it is defined is associated to one of the model frames according to its name.

Moreover, the known contacts can be activated or deactivated with the methods `DynaCoM::activateContact6d()` and `DynaCoM::deactivateContact6d()`.

2.3.2 Force Distribution

Function:

`DynaCoM::distributeForce()`.

One given centroidal wrench can be reproduced by infinite combinations of contact wrenches when we consider several contact surfaces. We manage this redundancy by a numerical optimization based on Quadratic Programming (QP).

In this optimization problem, we make sure that the combined action of all contact forces reproduces our **desired centroidal wrench** `cWrench`, while maintaining the forces of each contact **unilateral**, within its corresponding **friction cone**. On each contact, we choose the wrench with the **minimum force and torque** components according to user provided weights.

The optimization problem can be written as follows:

$$\begin{aligned} \underset{\boldsymbol{\lambda}}{\textbf{minimum}} \quad & \texttt{Regularization} \\ \textbf{subject to} \quad & \texttt{Unilaterality}, \\ & \texttt{FrictionCone}, \\ & \texttt{NewtonEuler}, \end{aligned} \tag{9}$$

where

$$\texttt{Regularization} \quad = \quad \boldsymbol{\lambda}^T Q \, \boldsymbol{\lambda}, \tag{10}$$

$$\texttt{Unilaterality} \quad : \quad U \boldsymbol{\lambda} < 0, \tag{11}$$

$$\texttt{FrictionCone} \quad : \quad C \boldsymbol{\lambda} < 0, \tag{12}$$

$$\texttt{NewtonEuler} \quad : \quad NE \, \boldsymbol{\lambda} = \texttt{cWrech}, \tag{13}$$

the optimization variable $\boldsymbol{\lambda}$ is a concatenation of all active contact wrenches expressed locally on each contact frame, and the matrices Q, U, C and NE are concatenations of all active contact matrices:

$$Q = \begin{bmatrix} Q_1 & & \\ & Q_2 & \\ & & \ddots \end{bmatrix}, \qquad U = \begin{bmatrix} U_1 & & \\ & U_2 & \\ & & \ddots \end{bmatrix}, \tag{14}$$

$$C = \begin{bmatrix} C_1 & & \\ & C_2 & \\ & & \ddots \end{bmatrix}, \qquad NE = \begin{bmatrix} NE_1 & NE_2 & \cdots \end{bmatrix}. \tag{15}$$

The numerical subscripts belong to an enumeration of the active contacts.