Controle do Manipulador MH12 através do ROS-Industrial

Aluno: Rodolpho Costa Ribeiro

Orientadores: Ramon Costa
Fernando Lizarralde

Dezembro de 2017

Introdução

ROS-Industrial é um projeto *open-source* que estende as aplicações de ROS para a indústria. Consiste de diversos pacotes de ROS cujo objetivo é criar uma interface única entre um computador executando ROS e controladores industriais de diversas fabricantes de manipuladores robóticos utilizados no processo de manufatura industrial.

Embora o número de manipuladores robóticos empregados no ambiente industrial aumente anualmente em todo o mundo, a quantidade de tarefas nas quais eles são empregados permanece estagnada. São tarefas repetitivas e de alto volume, como manipulação de materiais, corte, soldagem, montagem e pintura de componentes de produtos.

Em parte, o motivo dessa baixa diversificação de tarefas pode ser atribuído à arquitetura limitada dos *softwares* que regem o funcionamento dos controladores. Cada fabricante desenvolve suas próprias linguagens de programação e protocolos de comunicação, raramente divulgados, o que dificulta a utilização desses robôs industriais em pesquisas que expandam o espectro de tarefas que eles possam executar.

O ROS, por outro lado, é um projeto *open-source* que contém algoritmos eficientes em áreas mais complexas da robótica, como percepção e planejamento de trajetória. A aplicação desses algoritmos na indústria permitiria que os robôs executassem tarefas de maior grau de dificuldade, mas para integrar esses algoritmos de ROS aos controladores industriais seria necessário escrever *drivers* específicos para cada aplicação e para cada controlador que se desejasse utilizar, o que teria um alto custo.

O ROS-Industrial, portanto, surgiu de uma parceria entre a equipe responsável pelo ROS e algumas das principais fabricantes de manipuladores industriais para criar uma interface comum, composta por pacotes, bibliotecas e *drivers* que seguem as especificações do projeto, que permita utilizar algoritmos de alto-nível existentes no ecossistema do ROS em controladores que possuem arquiteturas completamente distintas. Essa integração reduz custos para as fabricantes, que não precisam desenvolver algoritmos individualmente para cada um de seus controladores, e possibilita à comunidade do ROS testar algoritmos em manipuladores mecanicamente otimizados, robustos e capazes de executar trajetórias com alta precisão e velocidade, além de contar os dispositivos de segurança de baixo-nível dos controladores das fabricantes que ajudam a evitar colisões.

Arquitetura do ROS-Industrial

A Figura 1 mostra a arquitetura de alto-nível do ROS-Industrial, realizando a interface entre o ROS e um controlador da Motoman:

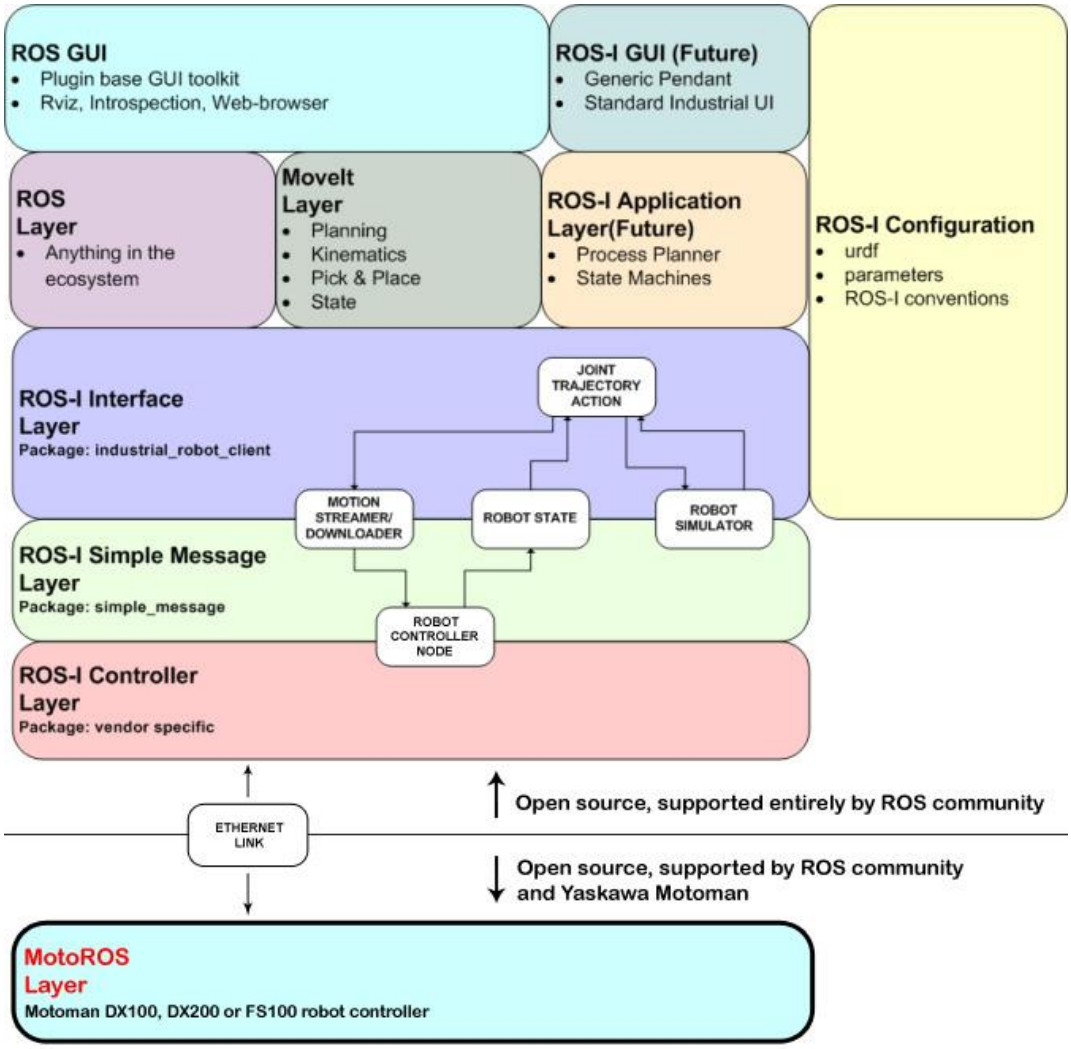


Figura 1: Arquitetura de alto-nível do ROS-Industrial (http://wiki.ros.org/motoman_driver)

Como o objetivo do projeto é justamente criar uma interface que independa do controlador utilizado, essa figura também representa a arquitetura do sistema mesmo que um controlador de outra fabricante seja utilizado. A única diferença é que ao invés da camada MotoROS (nome do *driver* de ROS que deve ser instalado nos controladores da Motoman),

haveria outra camada com o nome do *driver* que deve ser instalado nos controladores das respectivas fabricantes.

Como podemos ver, o ROS-Industrial consiste basicamente de dois pacotes genéricos, industrial_robot_client e simple_message, necessários para realizar a comunicação com qualquer controlador, e alguns pacotes específicos para cada fabricante. No caso da Motoman, por exemplo, temos o pacote motoman_driver.

O pacote industrial_robot_client é uma biblioteca cliente genérica que implementa as especificações do projeto (tais como os tipos de mensagens e nomes dos tópicos que devem ser utilizados por nós de ROS que desejem se comunicar com os controladores) através de classes de C++. O objetivo é que implementações específicas para cada controlador reutilizem os códigos contidos nesse pacote usando o mecanismo de herança de classes. Funcionalidades de baixo-nível que dependam de características particulares de cada controlador (como a forma como os bytes dos encoders dos motores devem ser traduzidos para mensagens de ROS, por exemplo) devem ser implementadas como métodos de classes derivadas das respectivas classes do industrial_robot_client. Essas classes derivadas devem redefinir os métodos que implementam essas funcionalidades e são agrupadas nos pacotes específicos de cada fabricante.

O pacote simple_message implementa o protocolo Simple Message, que define a estrutura das mensagens utilizadas para a comunicação entre os *drivers* de ROS e os controladores.

O *driver* é responsável por traduzir as mensagens de ROS para a linguagem de programação nativa do controlador utilizado. No caso do controlador DX200 da Motoman, que usamos nesse trabalho, o nome da linguagem é INFORM III. O MotoROS utiliza as mensagens do protocolo Simple Message, além de algumas mensagens específicas da Motoman, para enviar as mensagens de ROS do tipo trajectory_msgs/JointTrajectory, que contém a sequência de pontos da trajetória desejada no espaço das juntas. Ao receber as mensagens do *driver*, o controlador faz uma interpolação entre os pontos para gerar uma trajetória contínua suave e envia os comandos de velocidade para os motores das juntas do manipulador de acordo com a lei de controle implementada no controlador. Nem o método de interpolação nem a lei de controle são divulgados pela Motoman e também não há nenhuma funcionalidade no controlador que permita que eles sejam alterados.

O código-fonte do *driver* está disponível no pacote motoman_driver, assim como o arquivo binário resultante da compilação deste (MotoRosDX200.out), que deve ser instalado no controlador. Podemos alterar o código, mas para recompilá-lo e gerar um novo arquivo binário é preciso utilizar o *software* Motoplus SDK, vendido pela Motoman.

Também precisamos instalar o *job* INIT_ROS.JBI no DX200. *Job* é o nome dado aos programas escritos em INFORM III que contém as instruções que o DX200 deve executar. Esse *job* também está no pacote motoman_driver.

Nós do ROS-Industrial

Pelo menos três nós do ROS-Industrial precisam ser executados para que seja possível controlar o manipulador. Os nomes desses nós variam dependendo do pacote específico de fabricante que é utilizado, mas os nomes dos tópicos para os quais eles devem se inscrever, nos quais devem publicar e os tipos de mensagem desses tópicos são sempre os mesmos (a interface independe do fabricante). Para esse trabalho, a versão do ROS que usamos é o Kinetic Kame, rodando em Ubuntu 16.04, e estamos controlando um manipulador MH12 da Motoman através do controlador DX200, portanto utilizamos os nós do pacote motoman_driver.

- joint_state:

Publica as informações dos encoders absolutos das juntas do MH12 como mensagens do tipo sensor_msgs/JointState no tópico /joint_states.

sensor_msgs/JointState:

```
std_msgs/Header header
string[] name
float64[] position
float64[] velocity
float64[] effort
```

O campo *name* contém os nomes das juntas do manipulador. O campo *position* contém os deslocamentos angulares das juntas lidos pelos

encoders, em radianos. Os valores aparecem na mesma ordem que os nomes das juntas no campo *name*. Os campos *velocity* e *effort* não são preenchidos.

std_msgs/Header

uint32 seq
time stamp
string frame_id

Todas as mensagens utilizadas pelo ROS-Industrial possuem um campo com o tipo std_msgs/Header, que contém metadados sobre a mensagem da qual esse campo faz parte. O campo *seq* é o número de sequência da mensagem, para que saibamos a ordem em que as mensagens são enviadas. O campo *stamp* é o instante em que a mensagem foi enviada, de acordo com o relógio do ROS. O campo *frame_id* indica o sistema de coordenadas em que os dados numéricos das mensagens são escritos, mas esse campo não é preenchido nas mensagens do ROS-Industrial.

A taxa de publicação no tópico /joint_states é de 40 mensagens por segundo. Esse valor pode ser aumentado para até 1000 mensagens por segundo alterando uma variável no código do MotoROS, mas como mencionado anteriormente, para isso é preciso compilar o código com o Motoplus SDK e reinstalar o arquivo binário do *driver* no DX200.

- joint_trajectory_action:

Esse nó implementa o servidor da ação JointTrajectoryAction. É responsável por publicar as trajetórias no tópico /joint_path_command, garantindo que cada trajetória seja publicada apenas uma vez nesse tópico e que ela não seja perdida.

Uma ação é uma forma de comunicação do tipo pedido-resposta no ROS, ao contrário do sistema de tópicos. Um nó cliente da ação faz um pedido (chamado de objetivo no ROS) e permanece bloqueado até receber uma resposta (chamada de resultado no ROS) do servidor. Enquanto não atinge o resultado, o servidor pode enviar mensagens de feedback para atualizar o cliente sobre o progresso do objetivo.

As ações são uma forma de comunicação de nível mais alto, construídas sobre o sistema de tópicos. Quando uma ação é implementada,

vários tipos de mensagens e seus respectivos tópicos são gerados para implementar o objetivo, o feedback e o resultado. Esses tópicos são para uso interno do protocolo de comunicação entre o cliente e o servidor da ação. Não é possível se inscrever ou publicar diretamente neles. Ao invés disso, as mensagens são enviadas e recebidas através de métodos de classes clientes associadas a cada ação.

- motion_streaming_interface:

Se inscreve para o tópico /joint_path_command, no qual são publicadas as trajetórias das juntas que o manipulador deve seguir como mensagens do tipo trajectory_msgs/JointTrajectory.

Esse nó é responsável pela validação da trajetória, ou seja, verifica se ela cumpre os requisitos impostos pelo controlador, como por exemplo verificar se os limites de deslocamento, velocidade e aceleração das juntas são respeitados e se o vetor de posição do primeiro ponto da trajetória coincide com a posição atual das juntas, lidas pelos encoders (por esse motivo, esse nó também se inscreve para o tópico /joint_states).

Caso a trajetória não satisfaça todos os requisitos, ela é rejeitada e o nó exibe uma mensagem específica para o erro encontrado. Se a trajetória for validada, ela é enviada para o MotoROS.

As regras para validação das trajetórias são estipuladas pelo controlador, portanto mesmo que o nó não rejeitasse trajetórias inválidas, o próprio controlador as rejeitaria.

trajectory_msgs/JointTrajectory:

```
std_msgs/Header header
string[] joint_names
trajectory_msgs/JointTrajectoryPoint[] points
```

O campo *joint_names* deve conter os mesmos nomes de juntas que o campo *name* das mensagens do tipo sensor_msgs/JointState, caso contrário não seria possível verificar se o primeiro ponto da trajetória desejada coincide com as leituras mais recentes dos encoders. O campo *points* contém a sequência de pontos da trajetória. Não há limite para o número de pontos.

trajectory_msgs/JointTrajectoryPoint:

```
float64[] positions
float64[] velocities
float64[] accelerations
float64[] effort
duration time_from_start
```

Para cada ponto, é obrigatório preencher os campos *positions*, em radianos, *velocities*, em radianos por segundo e *time_from_start*, que representa o instante em que esse ponto deve ser alcançado a partir do instante em que primeiro ponto da trajetória for alcançado. O campo *accelerations* é opcional e o campo *effort* (torques) não é usado.

Para que o controlador esteja habilitado a receber comandos de um computador através do ROS, é preciso passar a chave do *programming pendant* do controlador para a posição REMOTE e chamar o serviço /robot_enable para ligar os motores do manipulador. Para desabilitar os comandos remotos podemos tirar a chave da posição REMOTE ou chamar o serviço /robot_disable. Ambos os serviços são implementados no nó motion_streaming_interface.

Obs: Existe um arquivo launch no pacote motoman_driver para executar esses três nós simultaneamente(robot_interface_streaming_DX200.launch). É preciso entrar com o endereço IP do controlador como argumento. Como não há necessidade de alterar esse endereço com frequência, podemos escrevê-lo como valor default no launch para que não seja preciso escrever o argumento no terminal cada vez que quisermos executá-lo. O endereço utilizado para o DX200 é 192.168.10.77. A comunicação entre o computador remoto executando ROS e o controlador se dá através do cabo Ethernet em uma conexão via socket TCP/IP.

Uso do ROS-Industrial

Para enviar uma trajetória ao controlador, o usuário deve escrever um nó que implemente um cliente da ação JointTrajectoryAction e escrever a trajetória desejada como uma mensagem do tipo control_msgs/FollowJointTrajectoryGoal, que é o tipo de mensagem que o objetivo dessa ação deve ter. Opcionalmente, o nó escrito pelo usuário

também pode se inscrever para o tópico /joint_states para ler as mensagens dos encoders.

control_msgs/FollowJointTrajectoryGoal:

```
trajectory_msgs/JointTrajectory trajectory
control_msgs/JointTolerance[] path_tolerance
control_msgs/JointTolerance[] goal_tolerance
duration goal_time_tolerance
```

O campo *trajectory* tem o mesmo tipo de mensagem que deve ser enviado ao nó motion_streaming_interface. De fato, tudo que o nó joint_trajectory_action faz é extrair o campo trajectory do objetivo e publicá-lo diretamente no tópico /joint_path_command. Os campos de tolerância dessa mensagem não precisam ser preenchidos, pois ainda não são utilizados pelo servidor da ação.

Pacote denvolvido pelo usuário

Desenvolvemos o pacote trajectory. Nesse pacote está o nó trajectory_publisher, que se inscreve para o tópico /joint_states e implementa um cliente da ação JointTrajectoryAction. Esse nó lê arquivos XML contendo todos os pontos da trajetória desejada para as juntas, preenche os campos da mensagem-objetivo da ação e a envia para o servidor.

Os arquivos XML são gerados a partir de um modelo de MATLAB/SIMULINK. Entramos com a referência desejada para posição e orientação do efetuador do MH12 e o modelo calcula a cinemática inversa do manipulador para obter as trajetórias das juntas.

A Figura 2 mostra o diagrama de comunicação entre os nós necessários para controlar o manipulador. O diagrama foi obtido através do pacote rqt_graph.

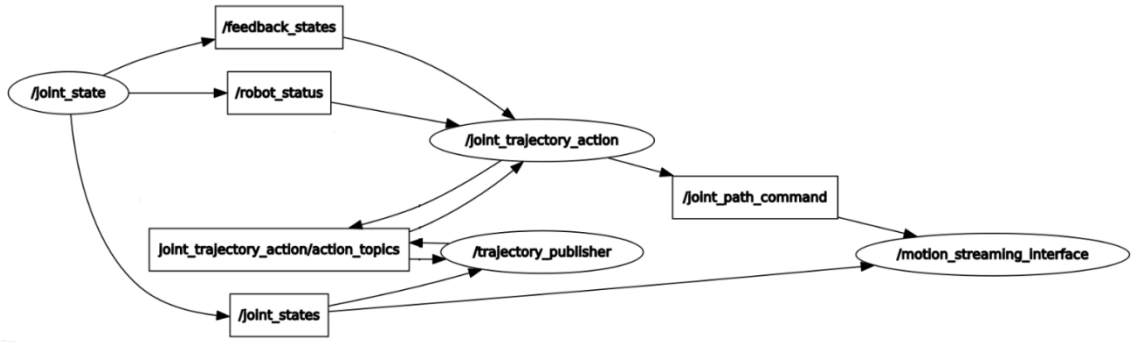


Figura 2: Diagrama de conexão entre os nós de ROS para envio de uma trajetória completa.

Controle On-the-Fly

Oficialmente, ainda não há uma interface no ROS-Industrial que permita o envio de pontos isolados para o controlador, o que é necessário para executar o controle on-the-fly. Contudo, como se trata de um projeto *open source*, qualquer um pode alterar livremente os códigos contidos no repositório do projeto, criar novas funcionalidades e disponibilizá-las.

Encontramos no github do ROS-Industrial uma *pull-request* de um desenvolvedor que alterou o nó motion_streaming_interface para que ele aceite o envio de pontos isolados de trajetória. Para isso, esse nó passa a se inscrever para um novo tópico, /joint_command, no qual também são publicadas mensagens do tipo trajectory_msgs/JointTrajectory, assim como no tópico joint_path_command. A diferença é que as mensagens publicadas nesse novo tópico devem conter apenas um ponto e a ação JointTrajectoryAction não é utilizada. O usuário deve escrever um nó que publique mensagens diretamente nesse tópico. O link para a versão do pacote motoman_driver que contém essa alteração é: https://github.com/ros-industrial/motoman/tree/boneil_point_streaming

Nó desenvolvido pelo usuário

Para testar essa funcionalidade, escrevemos o nó joint_control no pacote trajectory. Esse nó se inscreve para o tópico /joint_states e publica no tópico /joint_command.

A Figura 3 mostra a conexão entre os nós necessários para executar o controle on-the-fly.

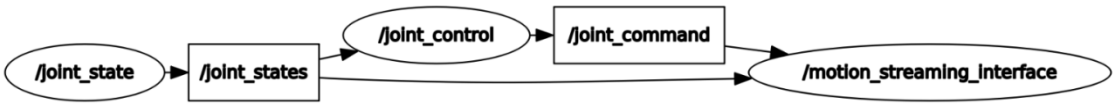


Figura 3: Diagrama de conexão entre os nós de ROS para controle on-the-fly.

A cada instante em que uma mensagem dos encoders é recebida, o nó calcula o valor da referência e a cinemática direta para então executar a seguinte lei de controle cinemático para posição e orientação do efetuador do manipulador MH12:

$$u = J^{-1}(v_d + Ke) \tag{1}$$

em que $v_d = [\dot{p}_d \quad \omega_d]^T$, sendo $\dot{p}_d$ a derivada de p_d, a referência de posição do efetuador, ω_d é a velocidade angular de referência do efetuador, J é o jacobiano geométrico do manipulador, $K = \begin{bmatrix} K_p & 0 \\ 0 & K_o \end{bmatrix}$ é a matriz de ganho do controle, $e = [p_e \quad q_e]^T$ é o vetor de erro, sendo $p_e = p_d - p$ e $q_e = (q_d q^{-1})_v$, em que p é a posição atual do efetuador, q é o quaternion unitário que representa a orientação atual do efetuador e q_d é o quaternion unitário que representa a referência de orientação.

Com essa lei de controle, a origem do erro é assintoticamente estável para todo $K > 0$.

Embora utilizemos o quaternion unitário como representação de orientação na lei de controle, é difícil visualizar uma referência de orientação variante no tempo nessa forma. Por esse motivo, escrevemos a referência como a matriz de rotação $R_d = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix}$ ou na representação eixo-ângulo dada pelo vetor unitário $\vec{u}_d$ e pelo deslocamento angular θ_d, e então convertemos para q_d usando as relações [1]:

$$q_d = \frac{1}{2\sqrt{tr(R_d)+1}} \begin{bmatrix} tr(R_d)+1 \\ r_{32} - r_{23} \\ r_{13} - r_{31} \\ r_{21} - r_{12} \end{bmatrix} \quad \text{ou} \quad q_d = \begin{bmatrix} \cos\left(\frac{\theta_d}{2}\right) \\ \vec{u}_d \sin\left(\frac{\theta_d}{2}\right) \end{bmatrix} \tag{2}$$

ω_d é obtida pela equação:

$$\omega_d = 2E(q)^T \dot{q}_d \tag{3}$$

em que $E(q) = \begin{bmatrix} -(q_d)_v^T \\ (q_d)_s I - S((q_d)_v) \end{bmatrix}$, sendo $(q_d)_s$ a parte escalar de q_d, $(q_d)_v$ a parte vetorial de q_d e $S(x) = \begin{bmatrix} 0 & -x_3 & x_2 \\ x_3 & 0 & -x_1 \\ -x_2 & x_1 & 0 \end{bmatrix}$.

Para o cálculo da cinemática direta do manipulador MH12, usamos a convenção Denavit-Hartenberg Standard, mostrada na Figura 4, para obter os sistemas de coordenadas dos elos, cujos parâmetros são exibidos na Tabela 1.

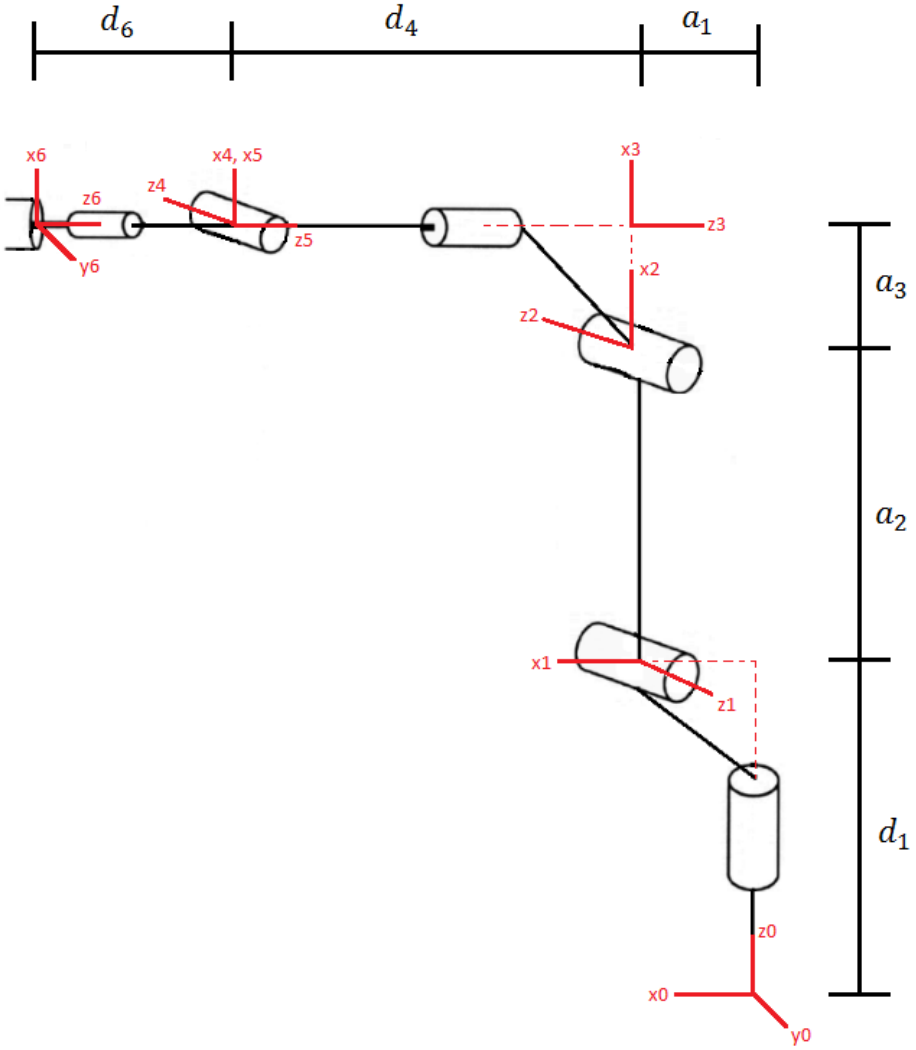


Figura 4: Convenção Denavit-Hartenberg Standard para o MH12.

Elo i	θ_i	d_i	a_i	α_i	offset
1	θ_1	d_1	a_1	$-\pi/2$	0
2	θ_2	0	a_2	π	$-\pi/2$
3	θ_3	0	a_3	$-\pi/2$	0
4	θ_4	$-d_4$	0	$\pi/2$	0
5	θ_5	0	0	$-\pi/2$	0
6	θ_6	$-d_6$	0	0	0

Medidas (cm)
$a_1 = 15.5$
$a_2 = 61.4$
$a_3 = 20$
$d_1 = 45$
$d_4 = 64$
$d_6 = 10$

Tabela 1: Parâmetros de Denavit-Hartenberg para o MH12.

Para obter p e q, respectivamente a posição e a orientação de E_6 em relação a E_0 (sistema de coordenadas inercial), usamos o algoritmo a seguir [2]:

Para o sistema de coordenadas $E_i = \{x_1, y_i, z_i\}$, solidário ao elo i, $i \in \{1, 2, 3, 4, 5, 6\}$, calculamos iterativamente:

$$q_i = \begin{bmatrix} \cos\left(\frac{\theta_i}{2}\right)\cos\left(\frac{\alpha_i}{2}\right) \\ \cos\left(\frac{\theta_i}{2}\right)\sin\left(\frac{\alpha_i}{2}\right) \\ \sin\left(\frac{\theta_i}{2}\right)\sin\left(\frac{\alpha_i}{2}\right) \\ \sin\left(\frac{\theta_i}{2}\right)\cos\left(\frac{\alpha_i}{2}\right) \end{bmatrix} \qquad p_i = \begin{bmatrix} a_i\cos(\theta_i) \\ a_i sin(\theta_i) \\ d_i \end{bmatrix} \rightarrow \bar{p_i} = \begin{bmatrix} 0 \\ p_i \end{bmatrix}$$

em que q_i é o quaternion unitário que representa a orientação de E_i em relação a E_{i-1} e p_i é a posição da origem de E_i nas coordenadas E_{i-1}.

Com isso, temos:

$$q = q_1 * q_2 * q_3 * q_4 * q_5 * q_6 \tag{4}$$

$\bar{p}_{01} = \bar{p}_1$
$\bar{p}_{12} = q_1 * \bar{p}_2 * q_1^{-1}$
$\bar{p}_{23} = q_1 * q_2 * \bar{p}_3 * q_2^{-1} * q_1^{-1}$

$$\bar{p}_{34} = q_1 * q_2 * q_3 * \bar{p}_4 * q_3^{-1} * q_2^{-1} * q_1^{-1}$$
$$\bar{p}_{45} = q_1 * q_2 * q_3 * q_4 * \bar{p}_5 * q_4^{-1} * q_3^{-1} * q_2^{-1} * q_1^{-1}$$
$$\bar{p}_{56} = q_1 * q_2 * q_3 * q_4 * q_5 * \bar{p}_5 * q_5^{-1} * q_4^{-1} * q_3^{-1} * q_2^{-1} * q_1^{-1}$$

$$\bar{p} = \bar{p}_{01} + \bar{p}_{12} + \bar{p}_{23} + \bar{p}_{34} + \bar{p}_{45} + \bar{p}_{56}$$

e p é a parte vetorial de $\bar{p}$:

$$p = (\bar{p})_v \tag{5}$$

O jacobiano geométrico também pode ser calculado iterativamente, coluna por coluna, a partir dos vetores obtidos durante o cálculo da cinemática direta.

Implementação da lei de controle

Idealmente, gostaríamos de enviar ao controlador a velocidade u das juntas, obtida pela lei de controle (1), nos instantes em que recebemos as mensagens dos encoders. Pela forma como o ROS-Industrial foi desenvolvido, porém, isso não é possível.

As mensagens do tipo trajectory_msgs/JointTrajectory publicadas no tópico /joint_command devem conter as posições, velocidades e instantes do ponto seguinte a ser alcançado pelo manipulador. O *loop* executado pelo nó joint_control é:

1. A mensagem mais recente enviada pelos encoders é recebida contendo os valores das posições das juntas $\theta(t)$, no instante atual t (calculado a partir da timestamp do cabeçalho da primeira mensagem recebida, que consideramos como $t = 0$).
2. Cálculo dos valores das referências de posição $p_d(t)$ e orientação ($R_d(t)$ ou $\vec{u}_d(t)$ e $\theta_d(t)$) do efetuador do manipulador no instante t.
3. Conversão de $R_d(t)$ ou $\vec{u}_d(t)$ e $\theta_d(t)$ para o quaternion $q_d(t)$, pela equação (2) e cálculo de $\dot{p}_d(t)$ e $\omega_d(t)$, a referência de velocidade angular do efetuador, pela equação (3).
4. Cálculo da cinemática direta, a partir de $\theta(t)$, para obter a posição $p(t)$ e o quaternion $q(t)$ do efetuador, pelas equações (4) e (5), e o jacobiano geométrico $J\big(\theta(t)\big)$.
5. Cálculo do sinal de controle $u(t)$ pela equação (1).

6. Estimativa do instante t' em que a próxima mensagem dos encoders chegará: $t' = t + dt$, em que $dt \cong 25ms$.
7. Estimativa dos valores das posições das juntas em t', pelo método de Euler: $\theta(t') = \theta(t) + u(t)dt$.
8. Repetição dos passos 2-5 no instante t'.
9. Preenchimento e envio da mensagem $goal$, do tipo trajectory_msgs/JointTrajectory, contendo apenas um ponto.
    - $goal.points[0].positions = \theta(t')$
    - $goal.points[0].velocities = u(t')$
    - $goal.points[0].time\_from\_start = t'$

Experimentos

Para testar o controle on-the-fly, usamos trajetórias circulares como referência de posição (em centímetros) e referência constante de orientação (punho para baixo).

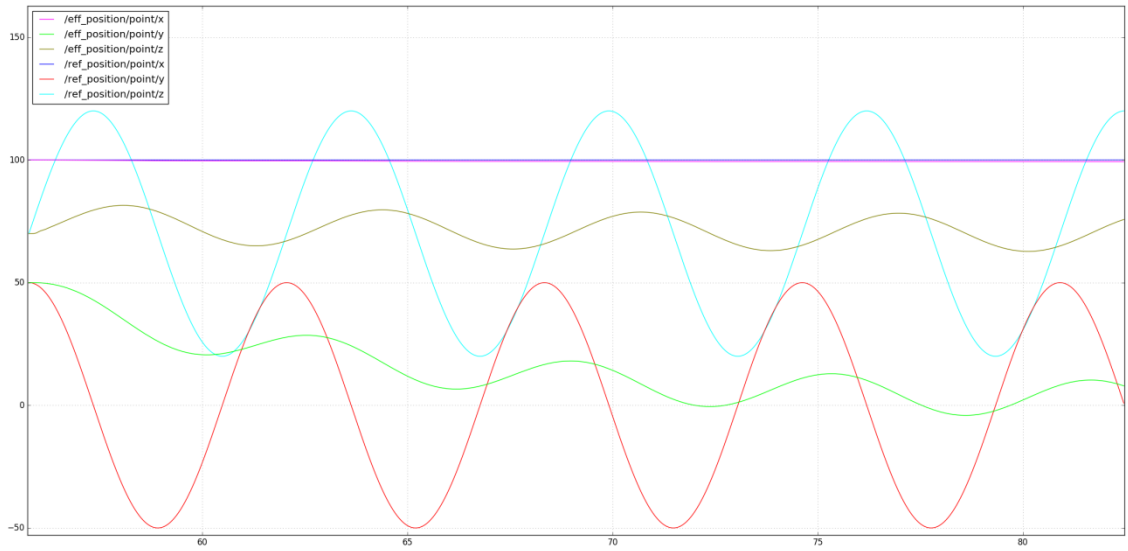
$$p_d(t) = \begin{bmatrix} 100 \\ 50.\cos(wt) \\ 50.\sin(wt) + 70 \end{bmatrix} \qquad R_d = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \rightarrow q_d = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

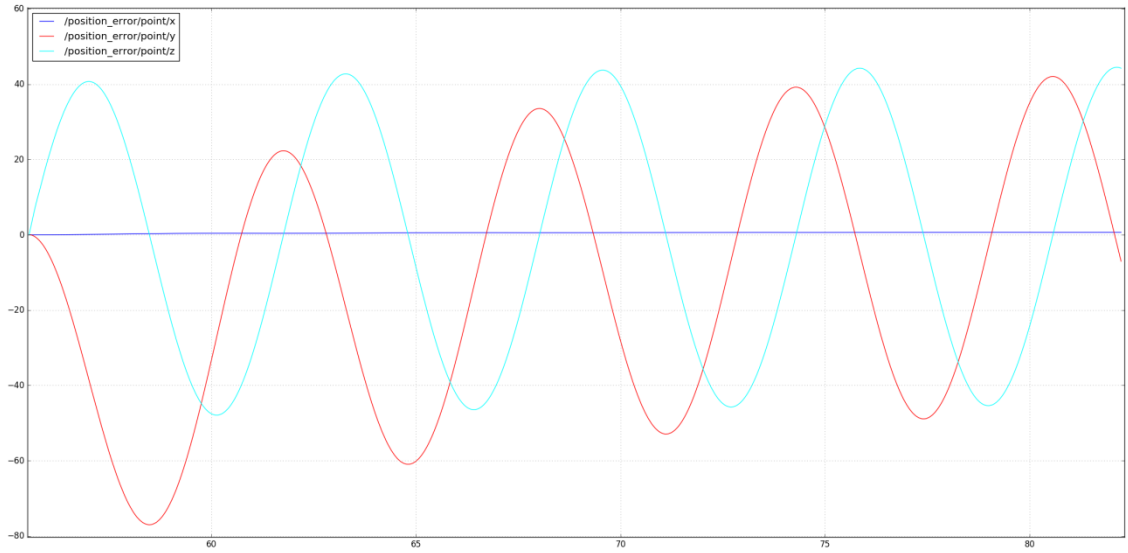
Consideramos que a matriz de ganho K da lei de controle é da forma $K = kI, k > 0$ e variamos o valor de k e da frequência angular w nos experimentos.

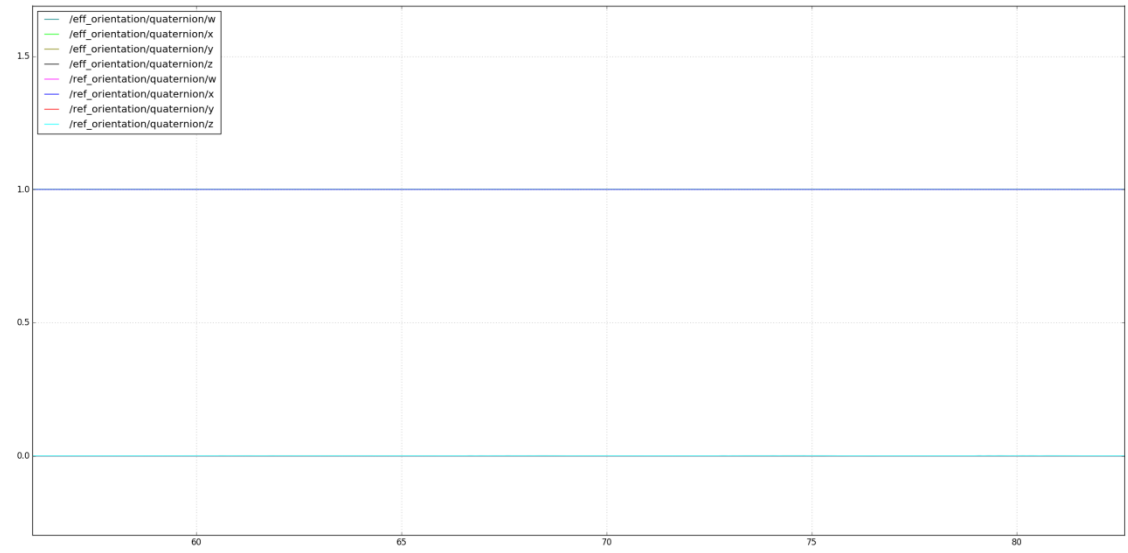
Verificamos que sempre há um atraso de 0,2-0,3s entre o instante em que publicamos uma mensagem de trajetória e o instante em que as mensagens dos encoders deixam de ser constantes, o que indica que o manipulador finalmente começou a se mover. Além disso, a velocidade real das juntas, que estimamos através das mensagens dos encoders, é sempre aproximadamente 8,5 vezes menor do que o sinal de controle que enviamos. Ainda não identificamos o motivo desses problemas.

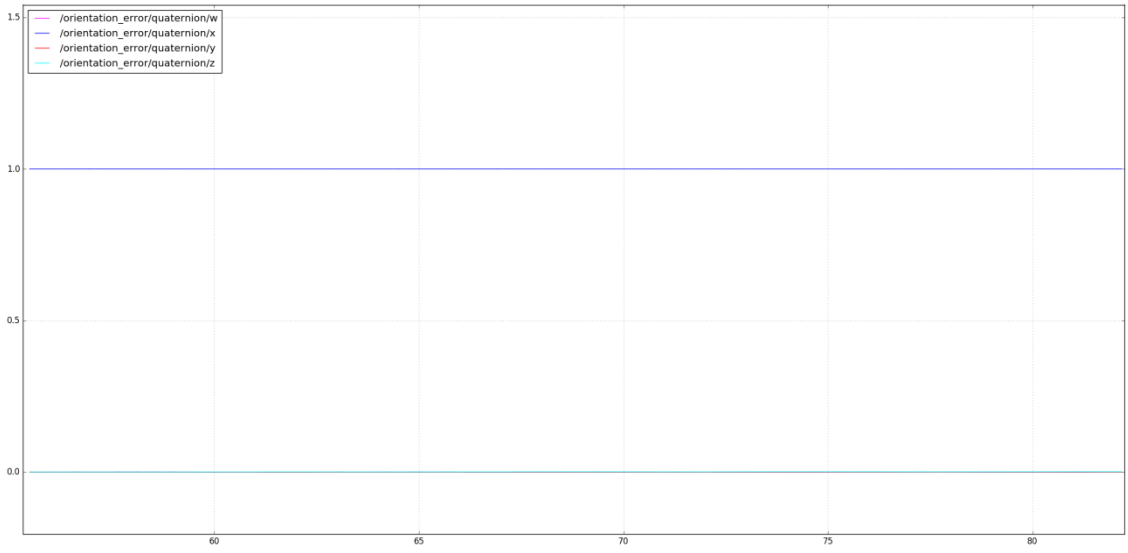
Quando testamos uma trajetória de referência variante no tempo, como os círculos, o erro estacionário não é nulo como deveria ser para a lei de controle utilizada. Quanto mais lenta a trajetória de referência e quanto maior o ganho do controlador, menor o erro estacionário.

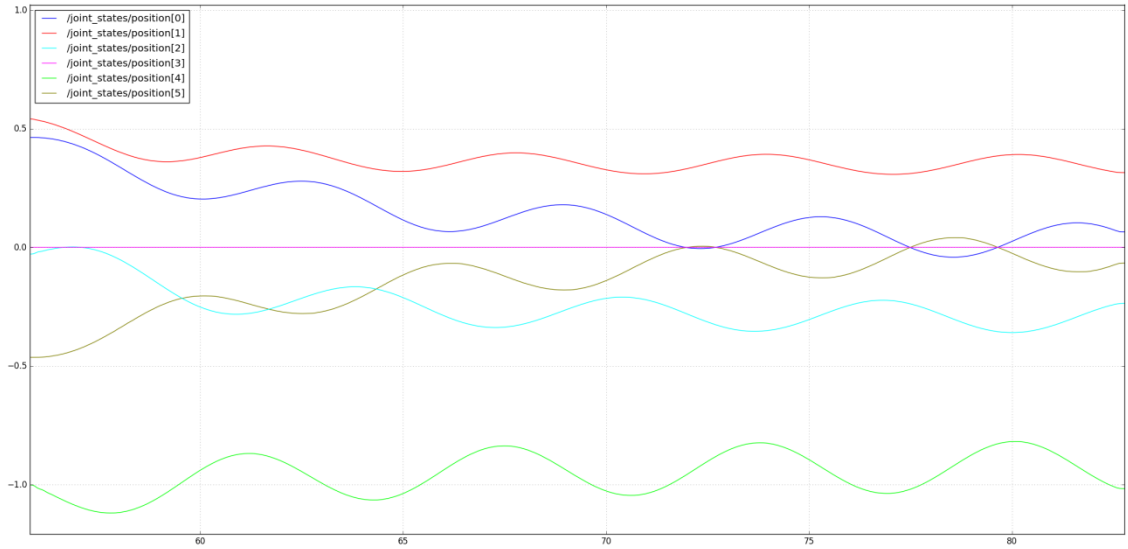
Para $w = 1, k = 1$

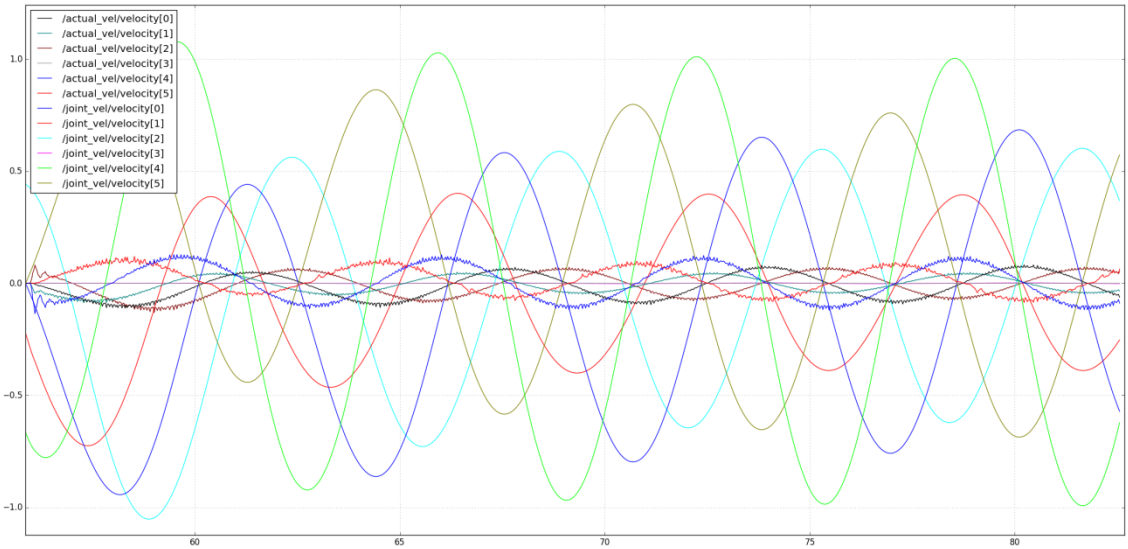

/eff_position/point/x
/eff_position/point/y
/eff_position/point/z
/ref_position/point/x
/ref_position/point/y
/ref_position/point/z
150
100
50
0
−50
60
65
70
75
80


/position_error/point/x
/position_error/point/y
/position_error/point/z
60
40
20
0
−20
−40
−60
−80
60
65
70
75
80

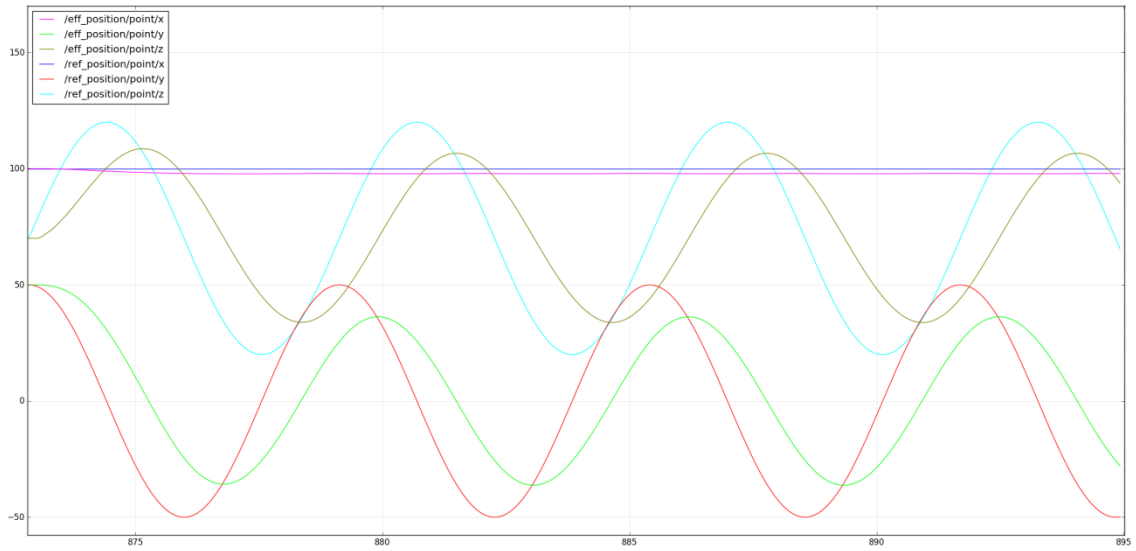

/eff_orientation/quaternion/w
/eff_orientation/quaternion/x
/eff_orientation/quaternion/y
/eff_orientation/quaternion/z
/ref_orientation/quaternion/w
/ref_orientation/quaternion/x
/ref_orientation/quaternion/y
/ref_orientation/quaternion/z
1.5
1.0
0.5
0.0
60
65
70
75
80

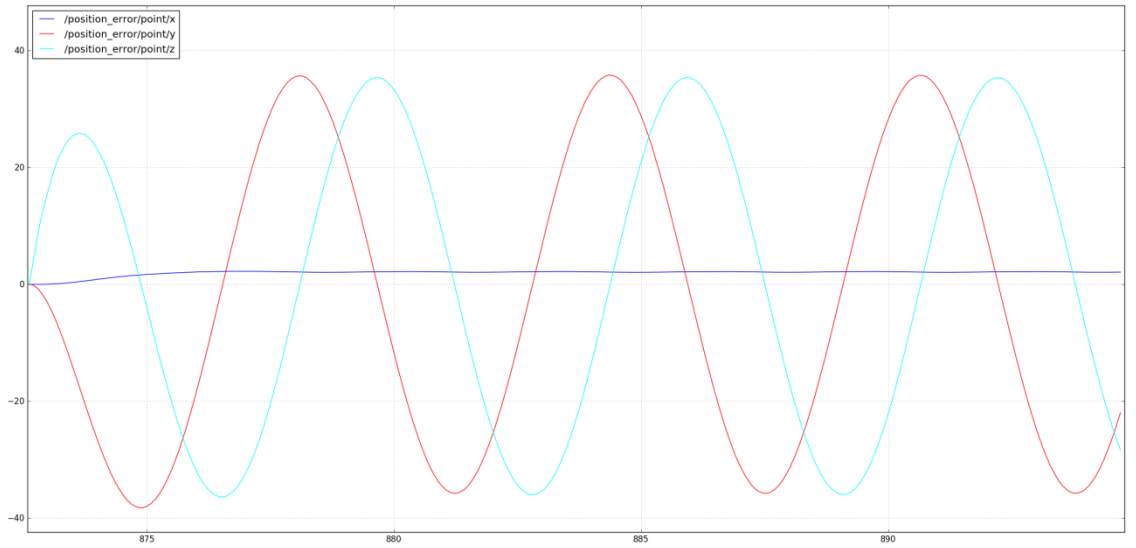

/orientation_error/quaternion/w
/orientation_error/quaternion/x
/orientation_error/quaternion/y
/orientation_error/quaternion/z
1.5
1.0
0.5
0.0
60
65
70
75
80

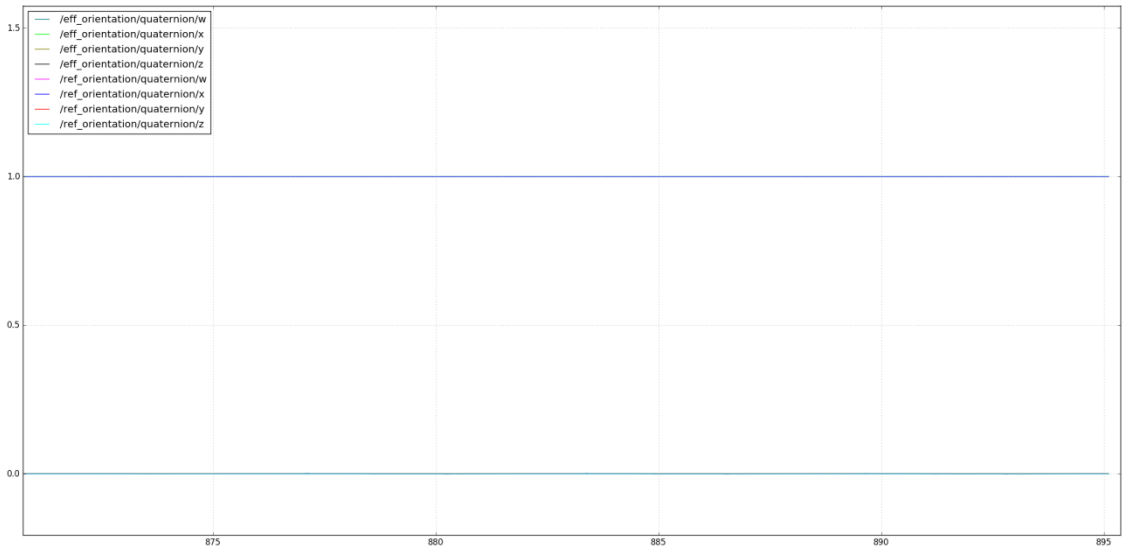

/joint_states/position[0]
/joint_states/position[1]
/joint_states/position[2]
/joint_states/position[3]
/joint_states/position[4]
/joint_states/position[5]
1.0
0.5
0.0
−0.5
−1.0
60
65
70
75
80

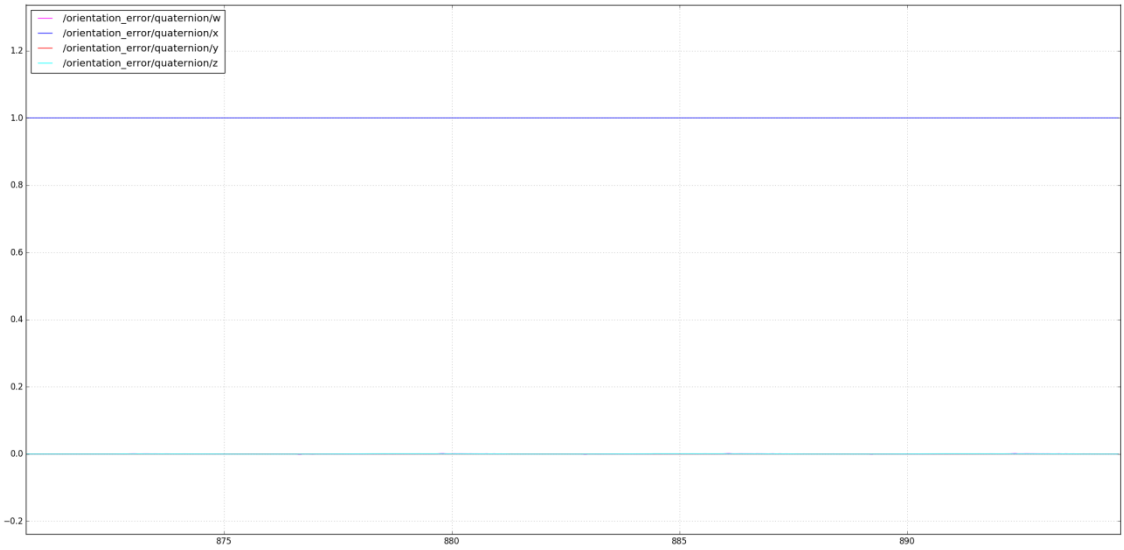

/actual_vel/velocity[0]
/actual_vel/velocity[1]
/actual_vel/velocity[2]
/actual_vel/velocity[3]
/actual_vel/velocity[4]
/actual_vel/velocity[5]
/joint_vel/velocity[0]
/joint_vel/velocity[1]
/joint_vel/velocity[2]
/joint_vel/velocity[3]
/joint_vel/velocity[4]
/joint_vel/velocity[5]
1.0
0.5
0.0
−0.5
−1.0
60
65
70
75
80

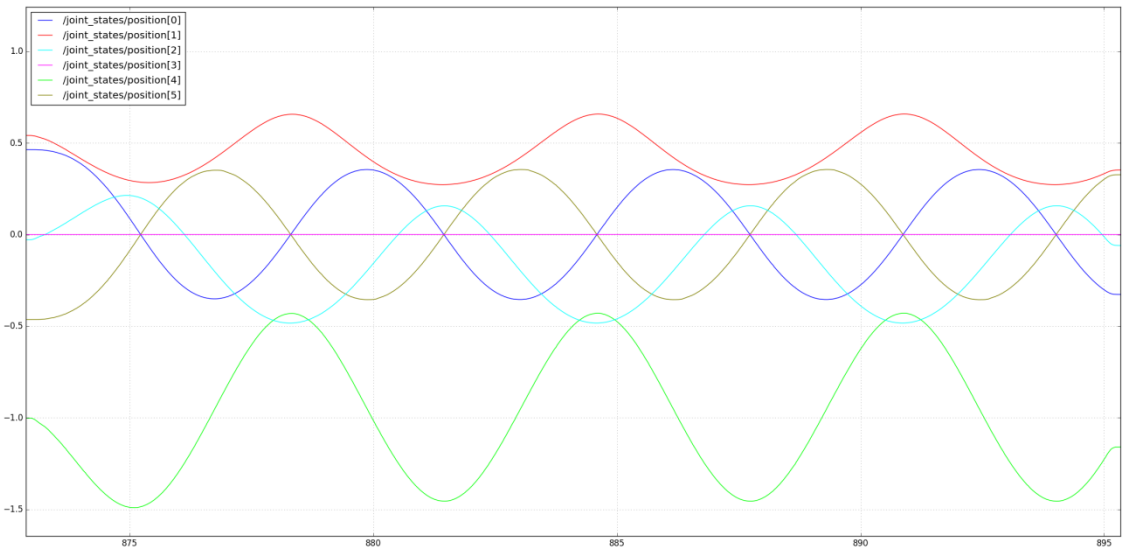
Para $w = 1, k = 10$

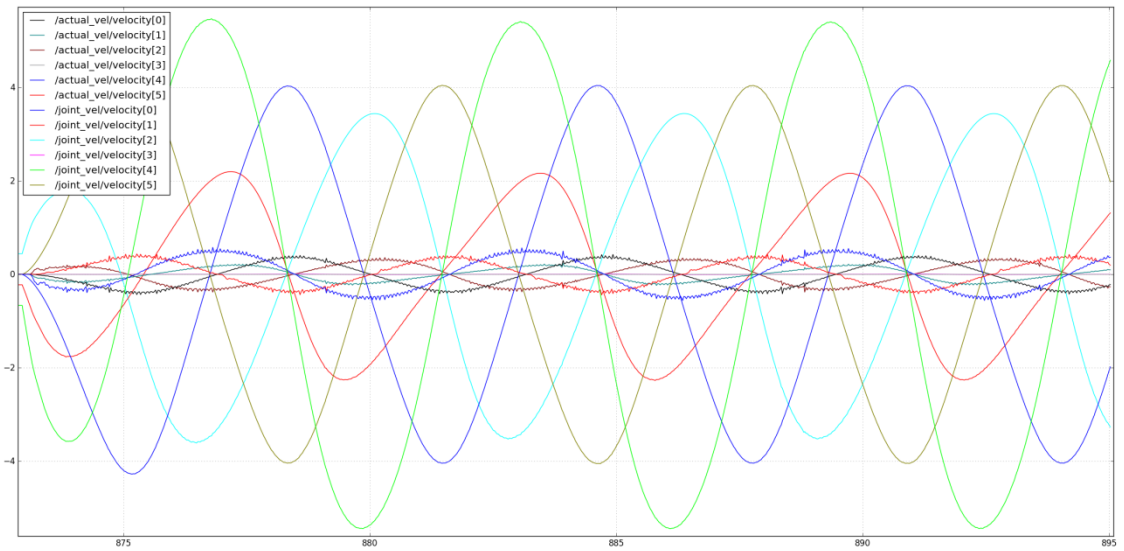

/eff_position/point/x
/eff_position/point/y
/eff_position/point/z
/ref_position/point/x
/ref_position/point/y
/ref_position/point/z
150
100
50
0
−50
875
880
885
890
895


/position_error/point/x
/position_error/point/y
/position_error/point/z
40
20
0
−20
−40
875
880
885
890

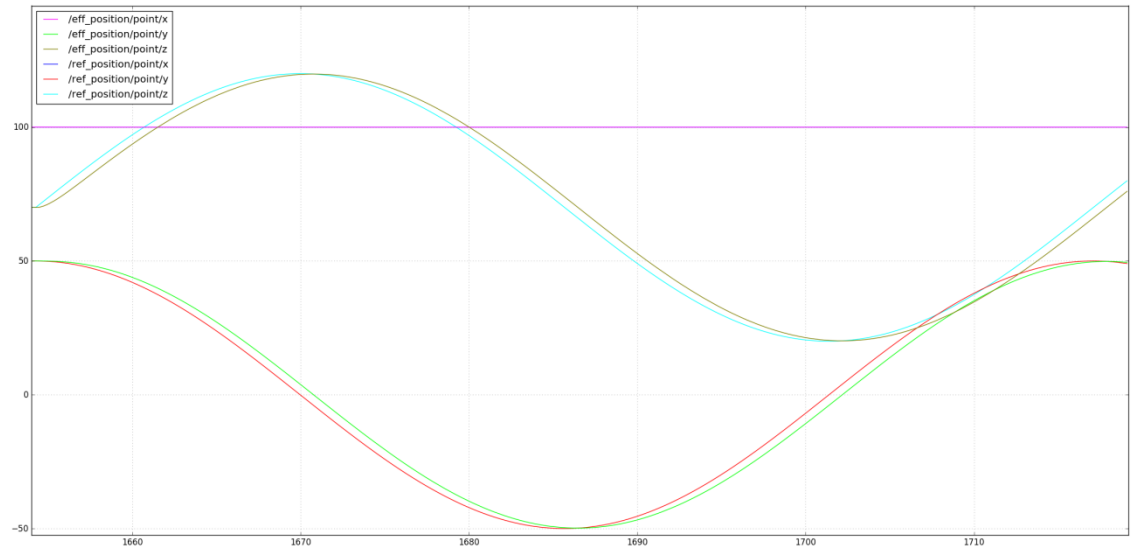

/eff_orientation/quaternion/w
/eff_orientation/quaternion/x
/eff_orientation/quaternion/y
/eff_orientation/quaternion/z
/ref_orientation/quaternion/w
/ref_orientation/quaternion/x
/ref_orientation/quaternion/y
/ref_orientation/quaternion/z
1.5
1.0
0.5
0.0
875
880
885
890
895

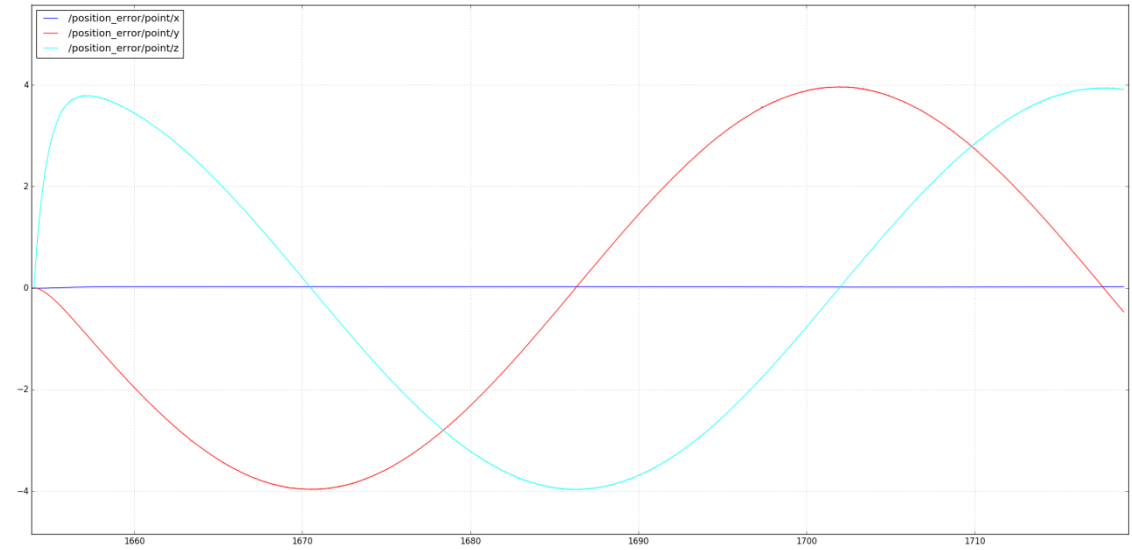

/orientation_error/quaternion/w
/orientation_error/quaternion/x
/orientation_error/quaternion/y
/orientation_error/quaternion/z
1.2
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
875
880
885
890

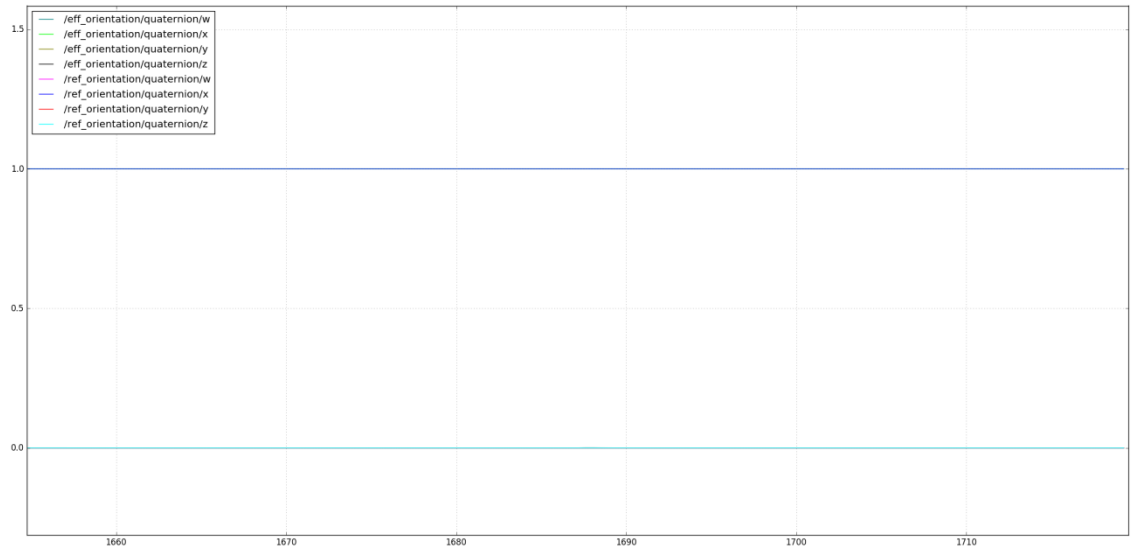

/joint_states/position[0]
/joint_states/position[1]
/joint_states/position[2]
/joint_states/position[3]
/joint_states/position[4]
/joint_states/position[5]
1.0
0.5
0.0
−0.5
−1.0
−1.5
875
880
885
890
895

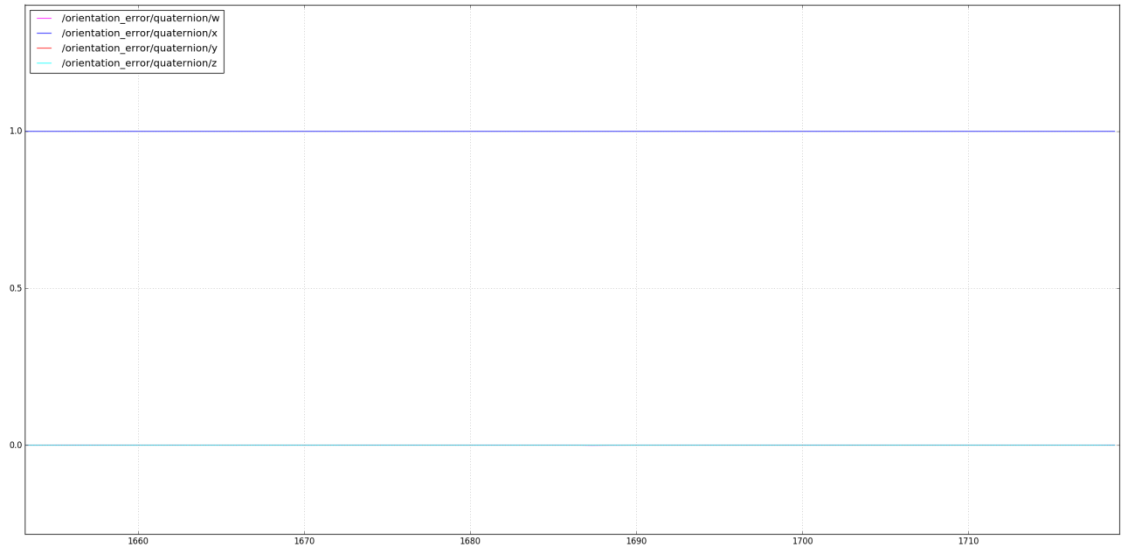

/actual_vel/velocity[0]
/actual_vel/velocity[1]
/actual_vel/velocity[2]
/actual_vel/velocity[3]
/actual_vel/velocity[4]
/actual_vel/velocity[5]
/joint_vel/velocity[0]
/joint_vel/velocity[1]
/joint_vel/velocity[2]
/joint_vel/velocity[3]
/joint_vel/velocity[4]
/joint_vel/velocity[5]
4
2
0
−2
−4
875
880
885
890
895

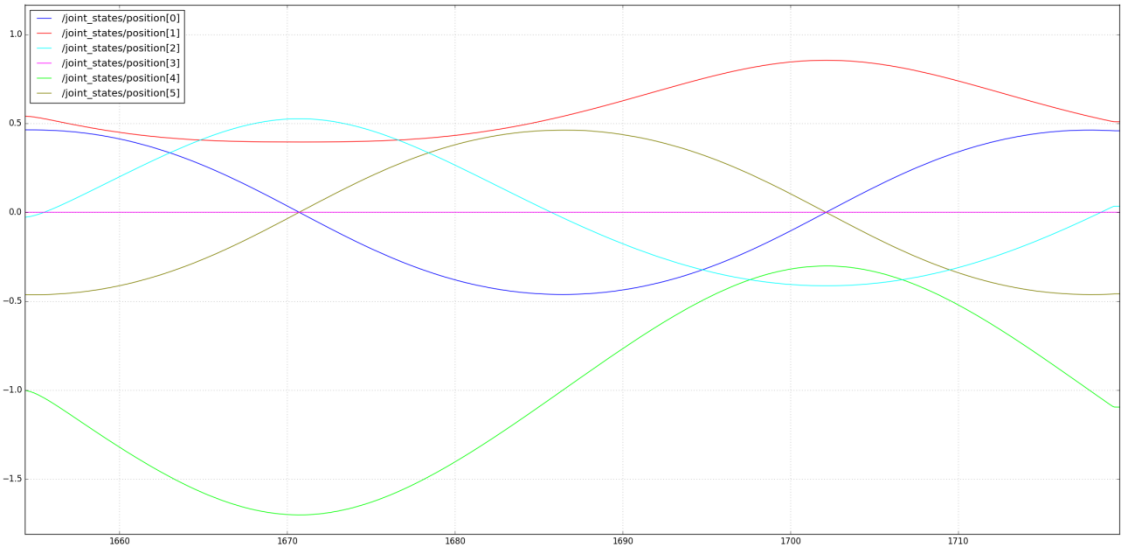
Para $w = 0.1, k = 10$

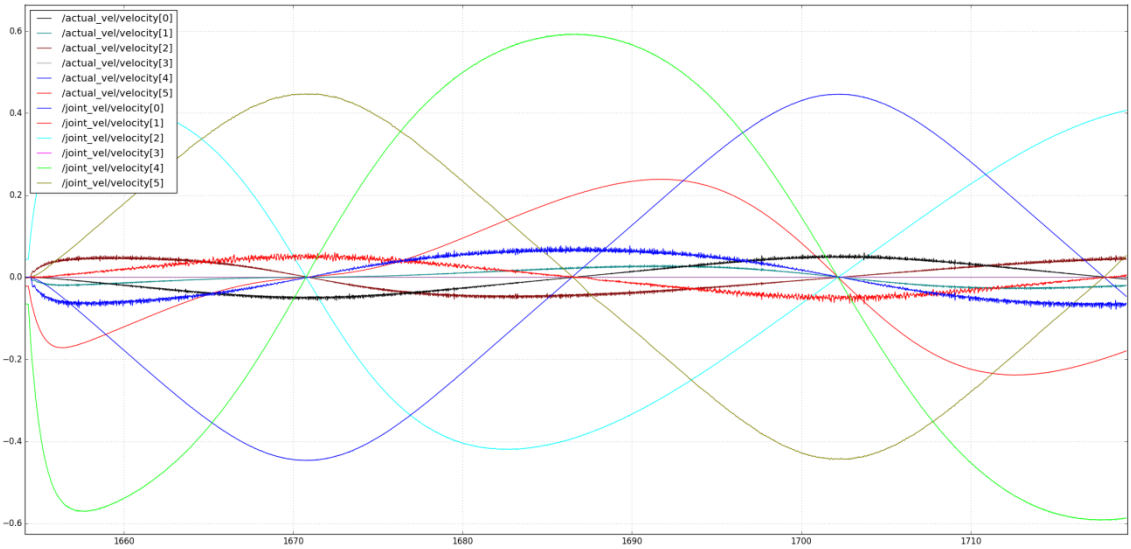

/eff_position/point/x
/eff_position/point/y
/eff_position/point/z
/ref_position/point/x
/ref_position/point/y
/ref_position/point/z
100
50
0
−50
1660
1670
1680
1690
1700
1710


/position_error/point/x
/position_error/point/y
/position_error/point/z
4
2
0
−2
−4
1660
1670
1680
1690
1700
1710


/eff_orientation/quaternion/w
/eff_orientation/quaternion/x
/eff_orientation/quaternion/y
/eff_orientation/quaternion/z
/ref_orientation/quaternion/w
/ref_orientation/quaternion/x
/ref_orientation/quaternion/y
/ref_orientation/quaternion/z
1.5
1.0
0.5
0.0
1660
1670
1680
1690
1700
1710


/orientation_error/quaternion/w
/orientation_error/quaternion/x
/orientation_error/quaternion/y
/orientation_error/quaternion/z
1.0
0.5
0.0
1660
1670
1680
1690
1700
1710


/joint_states/position[0]
/joint_states/position[1]
/joint_states/position[2]
/joint_states/position[3]
/joint_states/position[4]
/joint_states/position[5]
1.0
0.5
0.0
−0.5
−1.0
−1.5
1660
1670
1680
1690
1700
1710

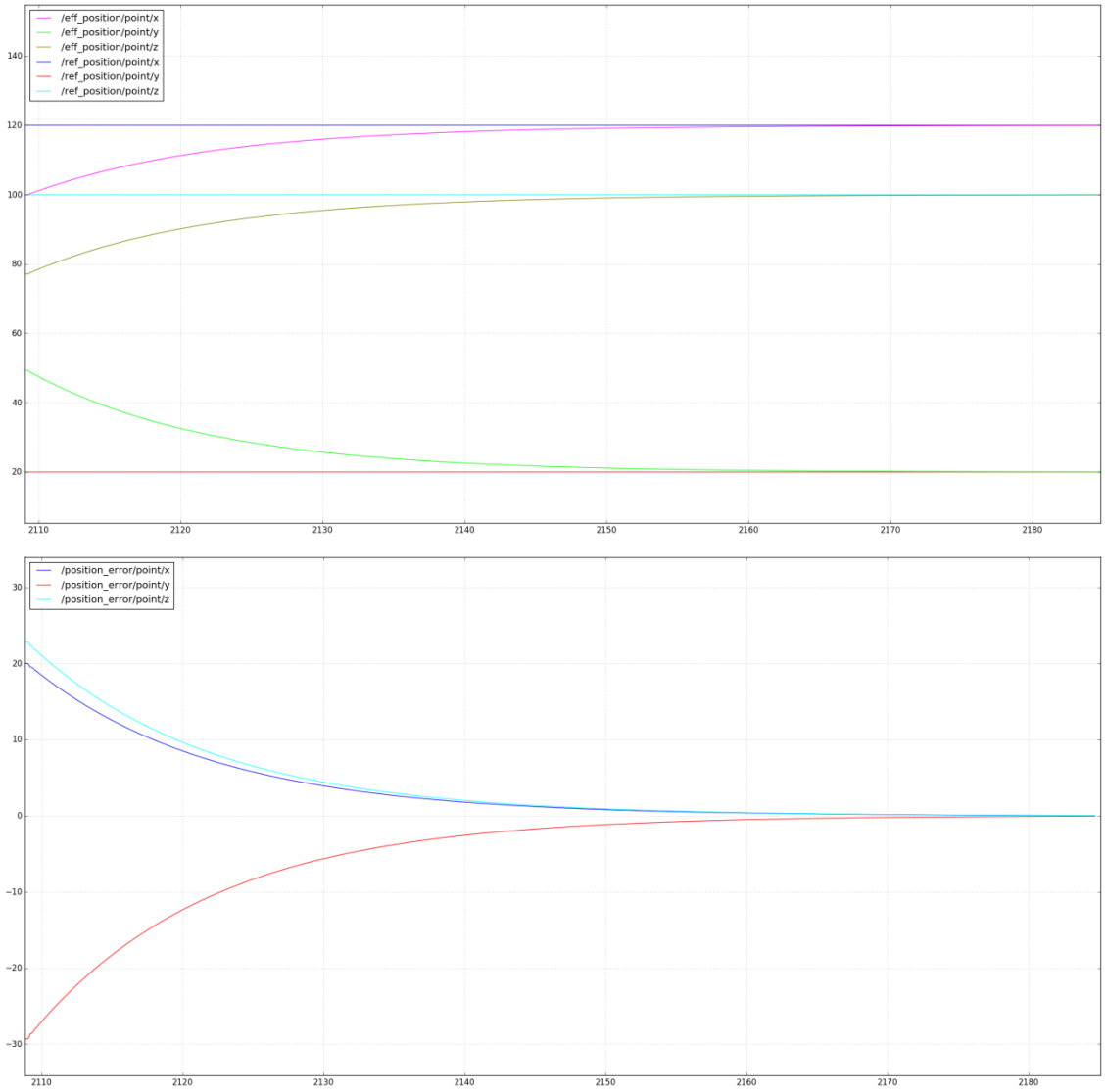

/actual_vel/velocity[0]
/actual_vel/velocity[1]
/actual_vel/velocity[2]
/actual_vel/velocity[3]
/actual_vel/velocity[4]
/actual_vel/velocity[5]
/joint_vel/velocity[0]
/joint_vel/velocity[1]
/joint_vel/velocity[2]
/joint_vel/velocity[3]
/joint_vel/velocity[4]
/joint_vel/velocity[5]
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
1660
1670
1680
1690
1700
1710

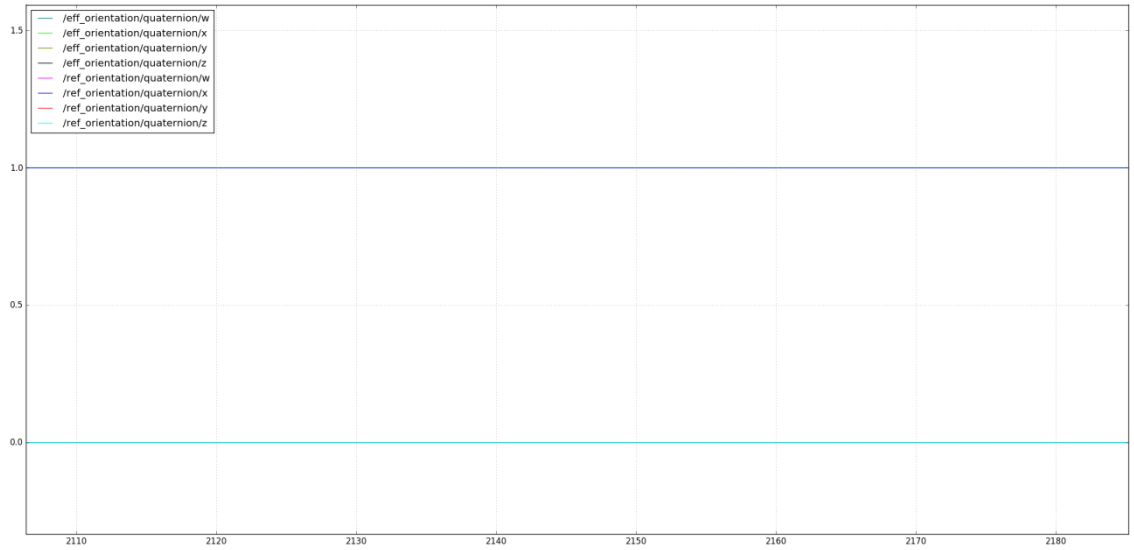
Testamos então uma trajetória constante:

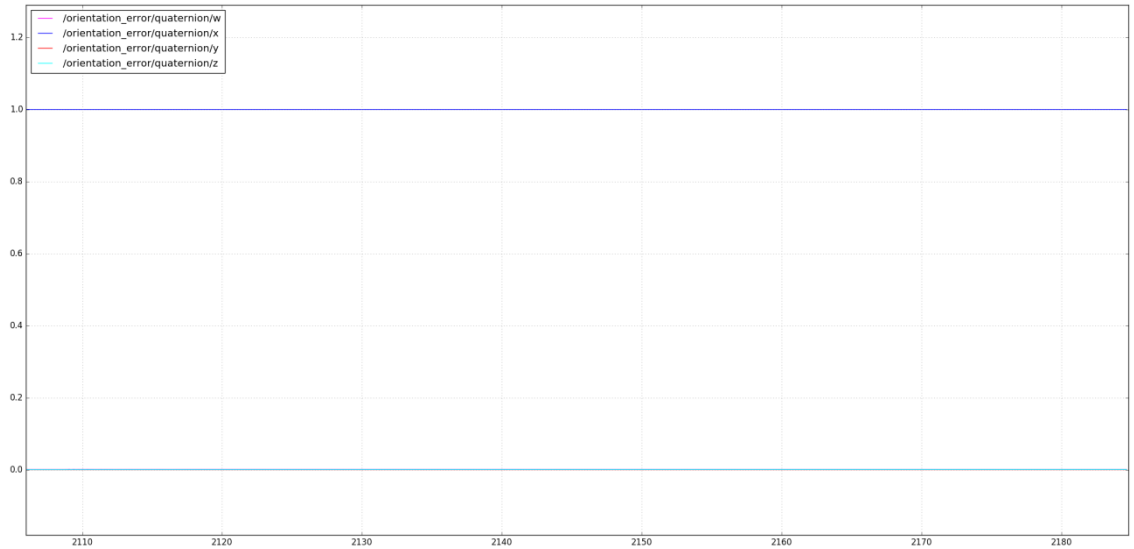
$$p_d = \begin{bmatrix} 120 \\ 20 \\ 100 \end{bmatrix} \qquad R_d = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \rightarrow q_d = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

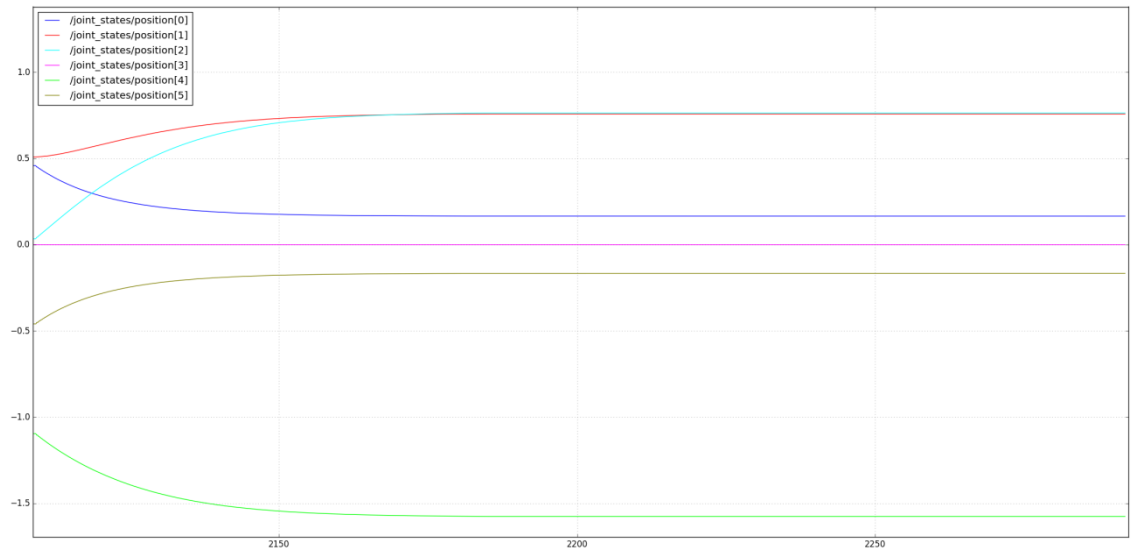
Verificamos que, nesse caso, o erro estacionário é nulo para todo $k > 0$.

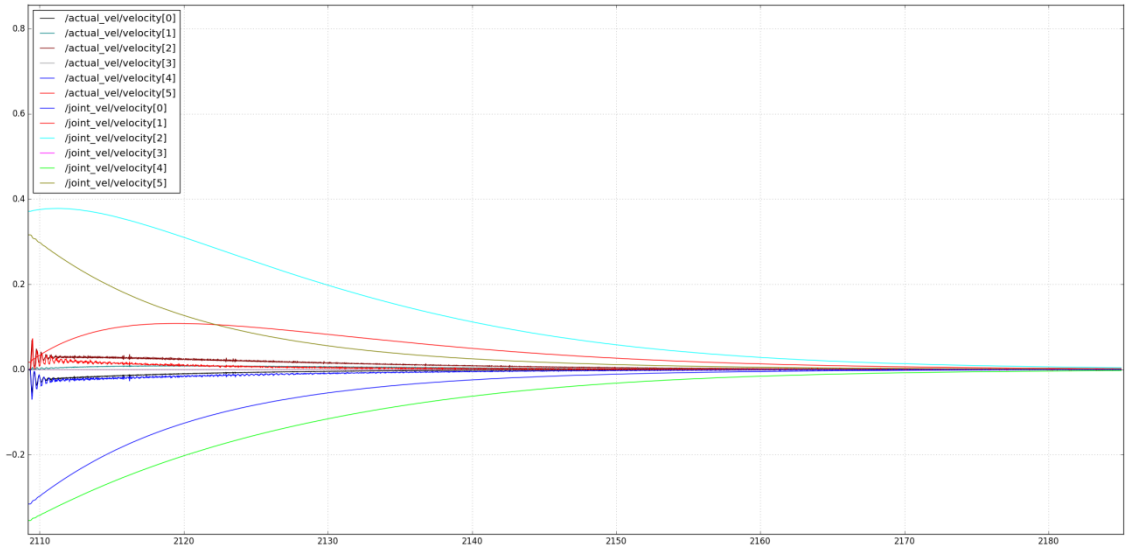
Para $k = 1$




/eff_orientation/quaternion/w
/eff_orientation/quaternion/x
/eff_orientation/quaternion/y
/eff_orientation/quaternion/z
/ref_orientation/quaternion/w
/ref_orientation/quaternion/x
/ref_orientation/quaternion/y
/ref_orientation/quaternion/z
1.5
1.0
0.5
0.0
2110
2120
2130
2140
2150
2160
2170
2180


/orientation_error/quaternion/w
/orientation_error/quaternion/x
/orientation_error/quaternion/y
/orientation_error/quaternion/z
1.2
1.0
0.8
0.6
0.4
0.2
0.0
2110
2120
2130
2140
2150
2160
2170
2180


/joint_states/position[0]
/joint_states/position[1]
/joint_states/position[2]
/joint_states/position[3]
/joint_states/position[4]
/joint_states/position[5]
1.0
0.5
0.0
−0.5
−1.0
−1.5
2150
2200
2250



Referências Bibliográficas:

[1] J. J. Craig, *Introduction to Robotics: Mechanics and Control*, 3ed, Pearson Prentice Hall, USA, 1989.

[2] R. Campa, K. Camarillo, and L. Arias, "Kinematic modeling and control of robot manipulators via unit quaternions: application to a spherical wrist", Proc. IEEE Conference on Decision and Control, San Diego, CA, December 2006.