Eje temático: Análisis de señales, modelados y simulación
Modalidad: Trabajo en desarrollo - Estudiante de grado
Facultad Regional: Reconquista

Programación en lenguaje Python de un MPC utilizando Casadi

MPC programming in Python language using Casadi

Presentación: 30/10/2025

Brian Bournissent
UTN, Facultad Regional Reconquista, Grupo de Investigación en Programación, Electrónica y Control, Reconquista, Santa Fe, Arg.
Email de contacto: brianbournissent@gmail.com
Enzo Corgniali
UTN, Facultad Regional Reconquista, Grupo de Investigación en Programación, Electrónica y Control, Reconquista, Santa Fe, Arg.
E-mail de contacto: enzo.ivan01@gmail.com
Erik Goldberger
UTN, Facultad Regional Reconquista, Grupo de Investigación en Programación, Electrónica y Control, Reconquista, Santa Fe, Arg.
E-mail de contacto: erikarielgoldberger@gmail.com
Malena Ribatto
UTN, Facultad Regional Reconquista, Grupo de Investigación en Programación, Electrónica y Control, Reconquista, Santa Fe, Arg.
Email de contacto: malenaribatto@gmail.com
Marcos Rossi
UTN, Facultad Regional Reconquista, Grupo de Investigación en Programación, Electrónica y Control, Reconquista, Santa Fe, Arg.
Email de contacto: rossimarcos2002@gmail.com
Ramiro Ayala
UTN, Facultad Regional Reconquista, Grupo de Investigación en Programación, Electrónica y Control, Reconquista, Santa Fe, Arg.
E-mail de contacto: ramiayala2013@gmail.com

Resumen

El objetivo de este trabajo es desarrollar un tutorial para programar en el lenguaje Python un MPC utilizando el marco de programación simbólica CasADi. La metodología permite formular problemas de control óptimo en tiempo discreto y aprovechar la diferenciación automática de CasADi para lograr alto rendimiento computacional. El enfoque se aplicó a un sistema masa–resorte–amortiguador; los resultados de simulación validan la robustez del controlador, guiando los estados x_1 y x_2 desde una condición inicial desplazada hasta el equilibrio xe = [0, 0] en 50 pasos, mientras la variable manipulada u se mantiene dentro de los límites de [-2, 2]. El tutorial obtenido es adaptable para futuros desarrollos de MPC más complejos.
Palabras clave: Control Predictivo Basado en Modelo (MPC), Optimización, Python, CasADi, Sistema masa-resorte

Abstract

The objective of this work is to develop a tutorial for implementing a Model Predictive Control (MPC) strategy in Python using the CasADi symbolic programming framework. The proposed methodology enables the formulation of discrete-time optimal control problems and leverages CasADi's automatic differentiation capabilities to achieve high computational performance. The approach was applied to a mass–spring–damper system; simulation results confirm the controller's robustness by driving the states x_1 and x_2 from an initial offset to the equilibrium point xe = [0, 0] within 50 time steps, while keeping the manipulated variable u within the predefined limits of [–2, 2], providing an adaptable tutorial for future developments of more complex MPC formulations.
Keywords: Model Predictive Control (MPC), Optimization, Python, CasADi, Mass-spring system

1. Introducción

En este trabajo se estudia el Control Predictivo Basado en Modelo (MPC) (Ferramosca et al., 2014) aplicado a un sistema masa-resorte. El MPC es una estrategia de control avanzado que utiliza un modelo matemático del sistema para predecir estados futuros y optimizar las acciones de control, teniendo en cuenta las restricciones del sistema.

A diferencia de los controladores clásicos, que solo reaccionan al estado presente, el MPC incorpora un horizonte de predicción. El problema de control se formula como una optimización matemática que busca determinar la secuencia óptima de acciones de control. Dicha secuencia debe minimizar una función de coste predefinida, como la reducción del error de posición, la minimización del consumo energético o la disminución del desgaste del actuador. De este modo, el MPC ofrece una alta eficacia y precisión, y también permite incluir restricciones sobre los estados y las entradas, lo cual es fundamental en aplicaciones reales. Se utilizó Python para la implementación, porque tiene bibliotecas variadas y es un lenguaje versátil. Se empleó la librería CasADi (Anderson et al., 2019) por su capacidad de cálculo automático de derivadas y su potencia en optimización numérica. Esta herramienta ofrece la flexibilidad necesaria para simular diferentes lazos de control y facilita la implementación de métodos de control numérico óptimo. Además, presenta una ventaja en la comprensión del lazo de control frente a herramientas como do-mpc (Felix et al., 2023) o hilo-mpc (Pohlodek et al., 2025), orientadas a formulaciones más robustas y estandarizadas. La herramienta se puede aplicar en escenarios fuera de línea y en aplicaciones de control predictivo no lineal (NMPC), lo que demuestra la aplicabilidad del MPC en sistemas dinámicos complejos.
El resto del trabajo se organiza de la siguiente manera: en la Sección 2: Control predictivo basado en modelo, mientras en la Sección 3: Programación en lenguaje Python, continuando con la Sección 4: Ejemplo de simulación y en Sección 5: Conclusiones.

2. Objetivos

El objetivo general de este trabajo es desarrollar un tutorial en Python para implementar un controlador predictivo basado en modelo (MPC) con CasADi, como guía adaptable a formulaciones más complejas. Considerando como objetivos específicos formular el modelo discreto del sistema masa–resorte–amortiguador; implementar el MPC en Python usando CasADi; y, simular y analizar el desempeño del controlador.

3. Control predictivo basado en modelo

El Control Predictivo basado en Modelo (MPC), es la técnica de control avanzado que mayor aceptación e implementación tuvo a nivel industrial que se basa en un enfoque tratable para poder aplicar a procesos reales la teoría de Control Óptimo, donde la acciones para lograr el objetivo de control son elegidas dentro de un conjunto factible, en un horizonte de tiempo finito.
La formulación de un lazo de control mediante MPC se basa en varios elementos fundamentales: el modelo de predicción, que es una representación matemática —lineal o no lineal— utilizada para anticipar la evolución temporal del sistema bajo determinadas acciones de control; el horizonte de predicción y control, que define los pasos futuros en los que se predice la evolución del sistema y cuyo valor depende del objetivo de control y del tipo de sistema, valores elevados de este tipo de horizonte nos facilita una mejor resultado pero a costa de un mayor procesamiento de cómputo, la función objetivo, que constituye la expresión matemática del criterio que se busca minimizar dentro del problema de optimización, reflejando el desempeño deseado del lazo de control; y finalmente las restricciones, que establecen los límites dentro de los cuales debe evolucionar el sistema, tanto en sus estados como en sus salidas y acciones de control, asegurando que la dinámica permanezca dentro de márgenes preestablecidos a lo largo del horizonte de predicción. Una forma general de escribir el problema de optimización asociado a una estrategia de MPC es el siguiente:

$$\min_{u} \quad \sum_{i=0}^{N-1} V_N\left(x_{i|k}; \boldsymbol{u}\right) = \sum_{i=0}^{N-1} \ell\left(x_{i|k}, u_{i|k}\right) + V_f\left(x_{N|k}\right) \tag{1}$$

sujeto a: $x_{0|k} = x_k$; $x_{i+1|k} = Ax_{i|k} + Bu_{i|k} + Ew_{i|k}$; $y_{i|k} = Cx_{i|k} + Du_{i|k}$; $x_{i|k} \in \mathbb{X}, u_{i|k} \in \mathbb{U}$; $i \in \mathbb{I}_{0:N-1}$; $x_{N|k} \in \Omega$.

En donde, x_k es el comportamiento dinámico asociado al estado, mientras que y_k se asocia a la salida del sistema; u_k son las acciones de control; V_N es la función objetivo; $\mathbb{X}$ es un conjunto convexo, y $\mathbb{U}$ es convexo y acotado. Además, $x_{i|k}$y $u_{i|k}$ representan las predicciones y planificaciones para los estados y entradas de control para el tiempo k hacia i pasos en el futuro.

4. Programación en lenguaje Python

La programación del modelo matemático, presentado en el diagrama de flujo de la Figura 1, se divide en los siguientes tres bloques: (3.1) Modelo matemático del sistema; (3.2) Lazo de control; (3.3) Simulación.

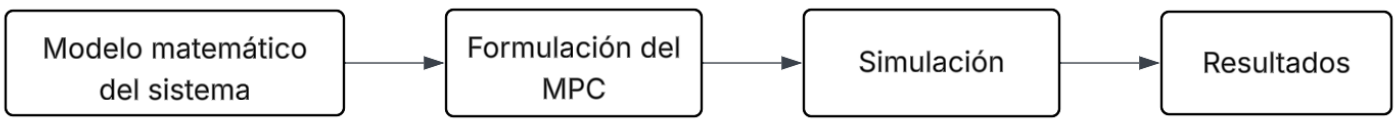

Figura 1. Diagrama de flujo de la metodología de implementación del MPC.

4.1 Modelo matemático del sistema

En CasADi las variables se declaran de forma simbólica, para que el software pueda realizar las operaciones de optimización, por ejemplo, para definir las variables de estado del sistema se utiliza la siguiente sintaxis:

```
x = SX.sym('x', 2)  ;  nx=2  # Estados del sistema
```

donde nx indica el número estados del sistema.

Se continúa definiendo la variable de control u, que representa la entrada aplicada al sistema.

```
u = SX.sym('u', 1)  ;  nu=1  # Acciones de control
```

Para cargar las matrices de un modelo lineal de tiempo continuo en espacio de estados, también se utiliza el lenguaje simbólico de Casadi:

```
# Matrices del sistema en tiempo continuo (SX)
A = SX([[0, 1],[-k/m, -b/m]])  ;  B = SX([[0],[1/m]])
```

La matriz A describe como los estados del sistema evolucionan naturalmente sin control externo. Por otro lado, la matriz B describe como la acción del control afecta a los estados del sistema.

Como el control predictivo de modelo opera en tiempo discreto, se debe discretizar el modelo en tiempo continuo:

```
def f(x, u):
    return A @ x + B @ u
# Integración RK4 simbólica (RungeKutta Orden 4)
k1 = f(x, u)  ;  k2 = f(x + Ts/2 * k1, u)  ;  k3 = f(x + Ts/2 * k2, u)
k4 = f(x + Ts * k3, u)
x_next = x + Ts/6 * (k1 + 2*k2 + 2*k3 + k4)
# Función discreta del modelo
F = Function('F', [x, u], [x_next], ['x', 'u'], ['x_next'])
```

Se utiliza la discretización por método Runge – Kutta 4 (RK4) en lugar de Euler, porque es un método de integración numérica más preciso para predecir la trayectoria futura del sistema; lo que permite tomar decisiones de control más efectivas y fiables.

4.2 Formulación del MPC – Lazo de control

Esta sección define el problema de optimización que el controlador debe resolver en cada paso de tiempo. El código utiliza la clase Opti de CasADi, que simplifica la formulación de problemas de optimización:

```
MPC = casadi.Opti()  ;  N=5 # Horizonte de control  ; Tsim=50 # Tiempo de simulación
```

Crea un objeto de optimización llamado MPC. Este objeto contiene todas las variables, restricciones y el objetivo de nuestro problema de optimización.

N define el horizonte de predicción, es decir, cuántos pasos de tiempo futuros debe mirar el controlar para tomar una decisión. Mientras que Tsim es el número total de pasos que durará la simulación.

En MPC el problema de control se formula como una optimización matemática, para resolverlo es necesario definir explícitamente las variables de decisión, que son las incógnitas que el optimizador ajustará:

```
xr = MPC.variable(nx, N+1)  # estados
ur = MPC.variable(nu, N)    # entrada manipulada
```

donde xr y ur son esas variables en forma de matriz que representan las incógnitas del problema. xr representa los estados futuros que el controlador predice, mientras que ur son las acciones de control.

Se definen los parámetros del problema, es decir, valores que se actualizan en cada instante de tiempo pero que no son optimizados:

```
# Condicion inicial (feedback de estado)
xk = MPC.parameter(nx, 1)  ;  xe = MPC.parameter(nx, 1)  ; ue = MPC.parameter(nu, 1)
```

donde xk es la condición inicial del sistema para cada momento (condición variable), mientras que xe y ue son los puntos de equilibrio que se quieren alcanzar (condición fija o referencia del sistema).

Continuando con la definición de variables, se definen las restricciones en entradas y estados como un arreglo de matrices:

```
umin = np.array([-2])  ;  umax = np.array([2])       #Restricciones para los estados
xmin = np.array([-5, -5])  ;  xmax = np.array([5, 5])
```

Estas variables definen los límites del sistema, tanto para las acciones de control (u) como para la de estado (x). En el ejemplo mostrado se tiene una acción de control y dos estados:

```
Q = np.array([[ 2 , 0 ],[ 0 , 2 ]])  ;  R = np.array([[ 0.2 ]])
T = np.array([[ 1 , 0 ],[ 0 , 1 ]])
```

Se establecen las matrices de ponderación utilizadas en la función de costo cuadrática, las cuales asignan la importancia relativa a los errores de estado, al esfuerzo de control y al estado terminal en la función de costo cuadrática del MPC.
Q, R y T son variables en forma de matrices, donde Q y T buscan mantener valores cercanos al de equilibrio, R intenta encontrar una solución más eficiente energéticamente, ponderando las acciones de control.
Se construye el funcional de costo cuadrático ponderado estándar del MPC, que suma las penalizaciones a lo largo del horizonte:

```
V = 0
for j in range(0, N-1):
    costo = (
        (ur[:, j] - ue).T @ (R @ (ur[:, j] - ue)) +
        (xr[:, j] - xe).T @ (Q @ (xr[:, j] - xe)) +
        (xr[:, N-1] - xe).T @ (T @ (xr[:, N-1] - xe)) )
    V += costo
VN = V
MPC.minimize(VN)
```

Se define la función de costo del MPC, donde V es la suma de los costos en cada paso del horizonte, y MPC.minimize(VN) indica al solver que minimice ese valor total.
Continuando, se definen las restricciones:

```
for j in range(0, N-1):
    MPC.subject_to(xr[:, j+1] == F(xr[:, j], ur[:, j])) # Modelo del sistema
  # Restricciones en los estados y las variables manipuladas
    MPC.subject_to(xmin <= xr[:,j])  ;  MPC.subject_to(xr[:,j] <= xmax)
    MPC.subject_to(umin <= ur[:,j])  ;  MPC.subject_to(ur[:,j] <= umax)
MPC.subject_to(xr[:, 0] == xk) # condición inicial
p_opts = {'expand': True}
s_opts = {
    'max_iter': 1500, 'print_level': 5 # Detalle de impresión en consola,
    'file_print_level': 5 # Detalle de impresión en archivo,
    'output_file': 'ipopt_log.txt'  # Nombre del archivo de salida
}
```

Donde primero se restringe el modelo, donde se le dice al optimizador que su predicción de estados futuros debe representar la física del sistema. Posteriormente, se aplican restricciones físicas, con la finalidad de asegurarse que los estados y las acciones de control se mantengan dentro de los límites físicos del sistema. Por último, se tienen en cuenta las restricciones que son propias de las condiciones iniciales, fijando el punto de partida de la predicción al estado actual del sistema
Y, por último, adjuntan líneas para la solución y configuración del Solver para poder llamarlo al MPC al momento de la simulación:

```
MPC.solver('ipopt', p_opts, s_opts)  # Asignar solver IPOPT con opciones
MPC = MPC.to_function(
    'MPC',
    [xk, xe, ue]  ;  [ur, xr]  ;  ['xk', 'xe', 'ue']  ;  ['uf_opt', 'xf_opt'] )
```

En la primera línea de esta imagen selecciona el motor o el solucionador que hará el trabajo, mientras que en la siguiente línea se asigna una función de compilación reutilizable, es decir que para cada paso de la simulación solo tiene que llamar a la función con el estado actual (xk).

4.3 Simulación

En este apartado comienza la sección donde se realiza la simulación del sistema con el controlador MPC. En primer lugar, se crean listas vacías para ir guardando valores en cada iteración:

```
vx = []  ;  vu = []
```

En vx almacena los estados y vu almacena las acciones de control durante la simulación.
A continuación, se definen los puntos de equilibrio:

```
xe = np.array([[ 0 ],[ 0 ]]) ; ue = np.array([[ 0 ]])
```

Esto establece la **condición inicial** del sistema:

```
xk = np.array([5, 5])  # Condición inicial en formato de array
```

Este es el bucle principal de la simulación del MPC, a continuación, se detalla brevemente como es su forma de operación:

```
for j in range(Tsim):
    uf_sol, xf_sol = MPC(xk, xe, ue) ; uf = uf_sol.full()  ;  xf = xf_sol.full()
```

```
    uf = uf_sol.full()  ;   xf = xf_sol.full()
    uk = uf[:, 0]  # Se aplica solamente la primera acción de Control
    vu.append(uk.flatten())  ;  vx.append(xk.flatten())
    xk = F(xk, uk).full()  # Calcula el estado x(k+1)
vu = np.array(vu) ; vx = np.array(vx)
```

El **MPC** opera en un ciclo continuo donde, primero, resuelve un problema de optimización para predecir la secuencia óptima de acciones y estados futuros a partir del estado actual del sistema, luego aplica solo la primera acción de esta secuencia (el concepto de **horizonte deslizante**), simula el siguiente estado basándose en la acción aplicada y el modelo del sistema, y finalmente almacena los resultados para su posterior análisis, permitiendo así que el controlador corrija errores y se adapte en cada iteración.

5. Ejemplo de simulación

Se tiene un sistema Masa-Resorte amortiguado como el mostrado en la Figura 2.

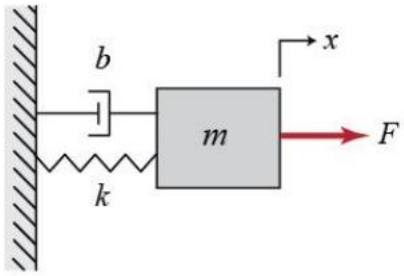

Figura 2. Sistema Masa-Resorte. m: masa, x: posición de la masa, F: fuerza externa (variable manipulada), b: coeficiente de amortiguamiento y k: constante del resorte.

El sistema esquematizado describe mediante la siguiente ecuación diferencial de segundo orden:

$$m\ddot{x}(t) + b\dot{x}(t) + kx(t) = F(t) \tag{2}$$

donde m es la masa, b el coeficiente de amortiguamiento, k constante del resorte, $F(t)$ la fuerza externa $x(t)$ es posición de la masa, $\dot{x}(t)$ la velocidad y $\ddot{x}(t)$ la aceleración.
Definiendo el vector de estados como $\boldsymbol{x}(t) = [\, x(t)\dot{x}(t)]^{\mathrm{T}}$, se tiene la siguiente ecuación de estado de tiempo continuo:

$$\dot{\boldsymbol{x}}(t) = A\boldsymbol{x}(t) + Bu(t) \tag{3}$$

donde $u(t) = F(t)$ es la variable manipulada. Las matrices de estados (A) y entrada (B) se definen la siguiente manera:

$$A = \begin{bmatrix} 0 & 1 \\ -\frac{k}{m} & -\frac{b}{m} \end{bmatrix}, B = \begin{bmatrix} 0 \\ \frac{1}{m} \end{bmatrix} \tag{4}$$

Se considerarán además restricciones para el estado y la entrada: $x \in \mathbb{X}: \{x \in \mathbb{R}^2: -5 \leq x \leq 5\}$; $u \in \mathbb{U}: \{x \in \mathbb{R}: -2 \leq x \leq 2\}$

El valor de la masa y la constante del resorte son $m = 1$ y $k = 1$, respectivamente. Dependiendo de los valores que asuma la constante b se tendrán distintos comportamientos del sistema.
Se propone un controlador MPC estándar con función de costo cuadráticas ponderadas para regular el sistema a un punto de equilibrio (el origen). El problema de optimización asociado a la estrategia de MPC es el siguiente:

$$\underset{\boldsymbol{u}}{mín} \quad V_N(x_k; \boldsymbol{u})$$

subject to: $x_0 = x;\ \ x_{j+1} = Ax_j + Bu_j; x_j \in \mathbb{X}, u_j \in \mathbb{U};\ j \in \mathbb{I}_{0:N-1}$, donde el funcional de costo es el indicado debajo:

$$V_N(x_k; \boldsymbol{u}) \coloneqq \left\|x_N - x_e\right\|_T^2 + \sum_{j=0}^{N-1} \left\|x_j - x_e\right\|_Q^2 + \left\|u_j - u_e\right\|_R^2 \tag{5}$$

Los resultados de la simulación que describen la dinámica del sistema se muestran en la Figura 3, considerando tiempo de muestreo de 0,5, un valor de b = 0,4 para la figura 2 (a), y un valor de b = 0,8 para la figura 2 (b).

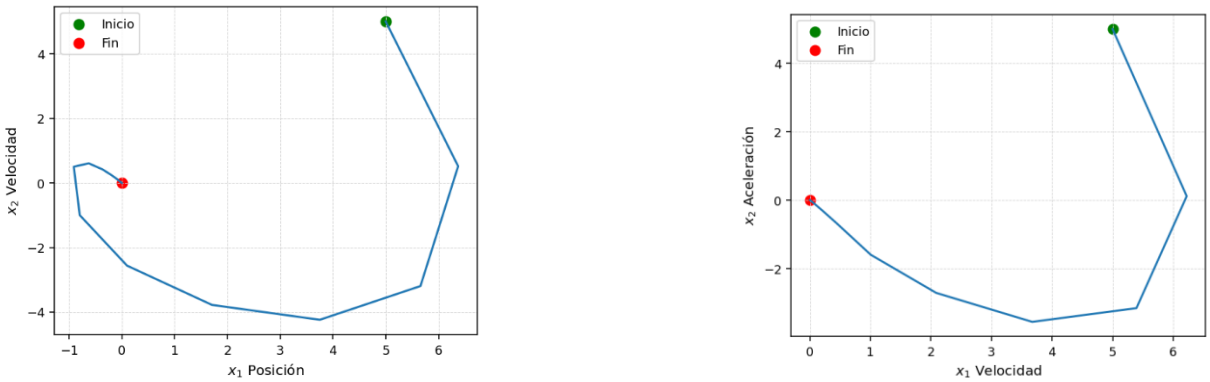

Figura 3 (a) y (b). Diagrama de fases: representación gráfica de la velocidad en función de la posición.

En la Figura 3 se aprecia la evolución de la dinámica del sistema desde la condición inicial hasta alcanzar el equilibrio. El objetivo principal de desarrollar un **tutorial de MPC en Python con CasADi se ha cumplido totalmente**. Se proporciona una **guía didáctica integral** que detalla el modelo discreto RK4, la implementación del MPC con código funcional y los resultados de simulación, sentando una base sólida para futuras extensiones a problemas más complejos.

6. Conclusiones

El trabajo cumple su objetivo al entregar un tutorial de MPC en Python con CasADi de alto valor didáctico. La metodología presentada permite formular problemas de control óptimo en tiempo discreto, aprovechando la capacidad de CasADi para la diferenciación automática, lo que se traduce en un notable rendimiento computacional.

El enfoque de control desarrollado se aplicó con éxito a un sistema de masa-resorte-amortiguador. Los resultados de la simulación validan la robustez y la eficacia del controlador MPC. Se observa que el controlador es capaz de guiar los estados del sistema, x_1 y x_2, desde una condición inicial desplazada hasta el punto de equilibrio deseado, xe = [0, 0], en un horizonte de tiempo de simulación de 50 pasos, manteniendo la variable manipulada, y dentro de los límites predefinidos de [-2, 2]. Además, la implementación es una plantilla clara que desmitifica la formulación del lazo de control óptimo. En esencia, este trabajo sienta una base adaptable inmediata para que el usuario explore y desarrolle formulaciones de control predictivo más complejas.

Agradecimientos

Se agradece el apoyo financiero provisto por la Universidad Tecnológica Nacional – Facultad Regional Reconquista.

Referencias bibliográficas

Felix et al. (2023). Towards FAIR nonlinear and robust model predictive control. *Control Engineering Practice.*

Ferramosca et al. (2014). Economic mpc for a changing economic criterion for linear systems. *IEEE Transactions on Automatic Control*, 2657–2667.

Pohlodek et al. (2025). Flexible development and evaluation of machine-learning-supported optimal control and estimation methods via HILO-MPC. *International Journal of Robust and Nonlinear Control*, 2835-2859.