PythonとCasADiで学ぶモデル予測制御

株式会社 Proxima Technology

目次

第3章　CasADi 入門

CasADi [1] は数理最適化、特に最適制御を行うためのオープンソースソフトウェアで、Joel Andersson 氏と Joris Gillis 氏によって開発が開始されました。C++、Python、Matlab などから利用可能ですが、本書では Python 版の CasADi を使用します。

3.1　環境構築

Python 版の CasADi は Windows、Mac、Linux すべての OS での動作をサポートしておりますが、本書では Windows 10/11 を前提に話を進めていきます。

3.1.1　Python のインストール

Windows 版の Python は公式サイト（`https://www.python.org/downloads/windows/`）から Stable Releases の 3.8 以上のバージョンをインストールして下さい。本書では Python 3.10.12 を使っています。

3.1.2　Anaconda による Python のインストール

また、Anaconda を使って Python 環境を構築することも可能で、こちらの場合は公式サイト（https://www.anaconda.com/download）からインストーラをダウンロードし、インストールを実行してください。本書でのコーディングはインタラクティブな環境で行えるよう Jupyter Notebook（https://jupyter.org/）を使います。その他の可視化のツールなども最初からインストールしてあるので Anaconda を使うと各種ライブラリを個別にインストールせずに済むのでとっかかりとしてはこちらの方が楽であると思われます。

3.1.3　CasADi のインストール

CasADi のインストールは非常に簡単で、以下の一行をコマンドプロンプトで実行するだけです。

```
pip install casadi
```

インストールが出来たかどうかは、Jupyter Notebook を立ち上げて

```
import numpy as np
import casadi
```

と記載したセルを実行してチェックしてください。エラーメッセージが出なければインストール成功です。なお、本書では執筆時点での最新版の CasADi v3.6.3 を使用しています。

3.2　シンボリック

CasADi では**すべての変数が行列の形をしたシンボリック**として扱われます。すなわち、スカラーは 1×1 の行列として、ベクトルは n

× 1 行列としてみなされます。

3.2.1　SX シンボリック

SX 型は各要素が式からなる行列を扱います。

```
x = casadi.SX.sym('x')
x
```

```
SX(x)
```

これはスカラー（1 × 1 行列）のシンボリックに対応します。行列のシンボリックも同様に

```
z = casadi.SX.sym('Z',4,2)
print(z)
```

```
SX(
[[Z_0, Z_4],
 [Z_1, Z_5],
 [Z_2, Z_6],
 [Z_3, Z_7]])
```

として与えられます。SX を使うと式は直観的に定義出来て、例えば $f(x) = \sqrt{x^2 + 10}$ は

```
f = casadi.sqrt(x**2 + 10)
print(f)
```

```
SX(sqrt((sq(x)+10)))
```

として与えることが出来ます。

また、SX の引数に直接リストや np.ndarray インスタンスを渡すことで SX シンボリックを生成することも可能です。

```
casadi.SX(np.array([[1,2]]))
```

```
SX([[1, 2]])
```

なお、CasADi には SX 型の他にも DM 型、MX 型という SX 型と非常に似たシンボリックが存在しますが、ここでは説明は割愛いたします。

3.2.2 値の代入

作成した行列への値の代入は numpy のようにスライス表現を用いて行うことが出来ます。

```
M = casadi.SX(casadi.diag([2,3,4,5]))
print(M)
```

```
[[2, 00, 00, 00],
 [00, 3, 00, 00],
 [00, 00, 4, 00],
 [00, 00, 00, 5]]
```

（※ここで 00 は structural zero と言ってスパース表現を表す値であり、実数の 0 とは異なります。）

```
print(M[:2,:3])
```

```
[[2, 00, 00],
 [00, 3, 00]]
```

```
M[0,:] = 2
print(M)
```

```
@1=2,
[[@1, @1, @1, @1],
 [00, 3, 00, 00],
 [00, 00, 4, 00],
 [00, 00, 00, 5]]
```

@1 が指定された要素に代入されていて、その値は最初の行に書かれている「@1=2」より確認できます。

3.2.3 算術

CasADi は基本的な算術である加減乗除と、多項式や三角関数などの初等関数をサポートしています。行列に対する操作は numpy と同様に各要素ごとの演算に自然に拡張されます。

```
x = casadi.SX.sym('x')
y = casadi.SX.sym('y',2,2)
print(casadi.sin(y)-x)
```

```
[[(sin(y_0)-x), (sin(y_2)-x)],
 [(sin(y_1)-x), (sin(y_3)-x)]]
```

行列同士の要素積は*で、行列積は@で計算できます。

```
print(y*y)
print(y@y)
```

```
[[sq(y_0), sq(y_2)],
 [sq(y_1), sq(y_3)]]
```

```
[[(sq(y_0)+(y_2*y_1)), ((y_0*y_2)+(y_2*y_3))],
 [((y_1*y_0)+(y_3*y_1)), ((y_1*y_2)+sq(y_3))]]
```

行列の転置は.T で計算できます。

```
print(y)
print(y.T)
```

```
[[y_0, y_2],
 [y_1, y_3]]

[[y_0, y_1],
 [y_2, y_3]]
```

また、内積（dot）は次のように定義されます。

$$\langle A, B\rangle := \mathrm{Tr}(AB) = \sum_{i,j} A_{ij}B_{ij} \tag{3.1}$$

```
x = casadi.SX.sym('x',2,2)
print(casadi.dot(x,y))
```

```
((((x_0*y_0)+(x_1*y_1))+(x_2*y_2))+(x_3*y_3))
```

そのほかにも、reshape や concatenate などの numpy にある関数に対応する操作が可能です。

3.2.4　自動微分

CasADi の中核機能の一つは自動微分（AD: Automatic (or Algorithmic) Differentiation）です。自動微分は TensorFlow や PyTorch などの深層学習向けのライブラリにも用いられている手法で、合成関数の微分がチェインルールで計算できることを利用しています。関数の構造をグラフ化して微分をシステマチックに解くことが出来るということは非常に面白い内容ですが、本書の範囲を超えるものですので今回は CasADi の自動微分機能の使い方のみを扱うこととします。

下の例では $f(x) = x^2$ とし、その微分 $f'(x) = 2x$ を計算しています。

```
x = casadi.SX.sym('x',1)
df = casadi.jacobian(x**2, x)
print(df)
print(casadi.simplify(df))
```

```
(x+x)
(2*x)
```

simplify を行う前は $f'(x) = x + x$ となっていますが、これは計算グラフの仕組み上出てくるもので値を実際に計算する上では問題にはなりません。

次に多変量の微分を計算してみましょう。行列 A とベクトル x の積の微分は

$$\frac{\partial(Ax)}{\partial x} = A \tag{3.2}$$

となりますが、実際に CasADi で同じ計算を実行してみると同様の結果が得られます。

```
A = casadi.SX.sym('A',3,2)
x = casadi.SX.sym('x',2)
print(A)
print(casadi.jacobian(A@x,x))
```

```
[[A_0, A_3],
 [A_1, A_4],
 [A_2, A_5]]

[[A_0, A_3],
 [A_1, A_4],
 [A_2, A_5]]
```

スカラー関数に対して hessian() メソッドを使うとヘッセ行列と勾配が同時に手に入ります。

```
[H,g] = casadi.hessian(casadi.dot(x,x),x)
print('H:', H)
```

```
H: @1=2,
[[@1, 00],
 [00, @1]]
```

3.2.5　関数オブジェクト

関数オブジェクトは、次の構文で定義されます。

```
f = casadi.Function( 関数名, 引数, ..., [options])
```

実際に $f(x,y)=(x,x\sin y)^T$ を定義してみると

```
x = casadi.SX.sym('x')
y = casadi.SX.sym('y')
f = casadi.Function('f',[x,y],\
            [x,casadi.sin(y)*x])
print(f)
```

```
f:(i0,i1)->(o0,o1) SXFunction
```

となります。定義した関数オブジェクトの呼び出しは代入するだけで行えて、DM 型の行列が返ってきます。DM 型は numpy.ndarray にそのままキャストできます。

```
print(f(1.1,2.1))
print(np.array(f(1.1,2.1)))
```

```
(DM(1.1), DM(0.94953))
[[[1.1       ]]

 [[0.9495303]]]
```

3.3　非線形最適化

CasADi の NLP（非線形計画）ソルバーは次の形式の最適化問題をサポートしています。

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & f(x,p) \\ \text{subject to} \quad & x_{lb} \leq x \qquad \leq x_{ub} \\ & g_{lb} \leq g(x,p) \quad \leq g_{ub} \end{aligned} \tag{3.3}$$

（ただし、$x \in \mathbb{R}^{N_x}$ は決定変数、$p \in \mathbb{R}^{N_p}$ は既知のパラメータです。）

CasADi は様々な NLP ソルバーをサポートしていますが、本書では基本的に IPOPT という主双対内点法というアルゴリズムを用いたオープンソースのソルバーを使用します。

それでは実際に、以下の問題を NLP ソルバーを使って解いてみましょう。

$$\begin{aligned} \underset{x,y}{\text{minimize}} \quad & x^3 + x^2 + 8x + 4y^2 + 3xy \\ \text{subject to} \quad & x^2 + y^2 = 1 \end{aligned} \tag{3.4}$$

これは CasADi では次のように定式化されます。

```
x = casadi.SX.sym('x')
y = casadi.SX.sym('y')
nlp = {'x':casadi.vertcat(x,y), \
       'f':x**3 + x**2 + 8*x + 4*y**2 + 3*x*y,\
       'g':x**2 + y**2 - 1}
S = casadi.nlpsol('S', 'ipopt', nlp)
print(S)
```

```
S:(x0[2],p[],lbx[2],ubx[2],lbg,ubg,lam_x0[2],lam_g0)->
(x[2],f,g,lam_x[2],lam_g,lam_p[]) IpoptInterface
```

定式化が出来たので、初期値を $(x, y) = (0, 1)$ として次のように最適化問題を解くことが出来ます。

```
r = S(x0=[0, 1],\
      lbg=0, ubg=0)
x_opt = np.array(r['x'])
print('x_opt: ', x_opt)
```

```
/////////////////////
// IPOPT の出力は省略 //
/////////////////////

x_opt:  [[-0.98575744]
 [ 0.16817333]]
```

この結果を matplotlib を使って可視化すると，確かに最適化問題が解けていることが確認できました。

```
import matplotlib.pyplot as plt
from matplotlib.colors import LogNorm

X_,Y_ = np.arange(-2,2.01,0.01),np.arange(-2,2.01,0.01)
X, Y = np.meshgrid(X_, Y_)
Z = X**3 + X**2 + 8*X + 4*Y**2 + 3*X*Y

levs = np.linspace(-20,50,20)

fig, ax = plt.subplots(figsize=(8,6))
ax.scatter(x_opt[0], x_opt[1], c="red")
cs = ax.contour(X,Y,Z,levels=levs)
ax.add_patch(plt.Circle(xy=(0,0),radius=1,fill=False))
fig.colorbar(cs)

plt.savefig("chap3_NLP_2D.png")
plt.show()
```

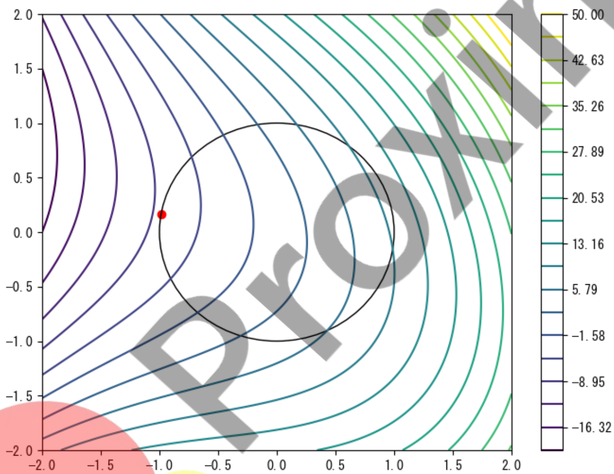

図 3.1: IPOPT による最適化の結果

3.4　常微分方程式 †

CasADI には常微分方程式（ODE: Ordinary Differential Equation）や微分代数方程式（DAE: Differential Algebraic Equation）の初期値問題を解くためのツールがそろっています。ここでは簡単な微分方程式を例に casadi.integrator() の使い方を扱います。

次にような微分方程式と初期条件を考えます。

$$\begin{aligned} \dot{x} &= -2x \\ x(0) &= 1 \end{aligned} \tag{3.5}$$

この方程式の解析解は $x(t) = e^{-2t}$ ですが、casadi.integrator() を用いれば次にように数値解を計算できます。

```
dt = 0.1
times = np.arange(0, 2+dt, dt)
X_t = [1] # 初期値


options = {'t0':0, 'tf':dt} # 積分範囲（時刻）
ode = {'x': x, 'ode': -2*x} # 常微分方程式

F = casadi.integrator('F', 'idas', ode, options)

for t in times[:-1]:
    res = F(x0=X_t[-1])
    X_t += res['xf'].toarray().tolist()[0]

Y_t = np.exp(-2*times)  # 解析解

plt.plot(times, X_t, label="数値解", color="blue")
plt.plot(times, Y_t, label="解析解", color="red")
plt.legend()
plt.savefig("chap3_integ.png")
plt.show()
```

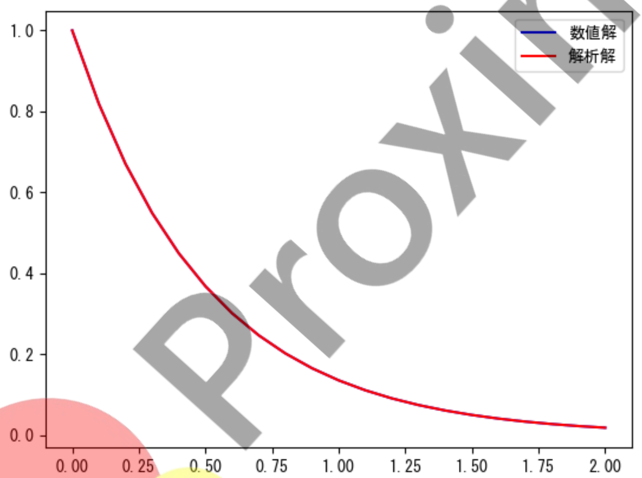

図 3.2: 解析解と数値解の比較

第 4 章　離散時間のモデル予測制御

4.1　非線形振動子モデル

この章では次のような減衰振動をする 3 次の補正項が入った 1 次元非線形振動子のモデルを考えてみましょう。

$$z_{k+2} = 1.750 z_{k+1} - 0.902 z_k - 0.01 z_k^3 + u_k \tag{4.1}$$

```
coef_0 = 1.750
coef_1 = -0.902
coef_2 = -0.01

def oscillator_eq(z_kp1, z_k, u):
    z_kp2 = coef_0*z_kp1 + coef_1*z_k + coef_2*z_k**3
    return z_kp2 + u
```

初期値を $z_0 = 1, z_1 = 1$ とし、制御入力を 0（$u_t \equiv 0$）とすると下図のようなパターンで減衰振動をします。

```
Z_k = []
z_kp1 = 1
z_k = 1

for t in range(100):
    z_kp2 = oscillator_eq(z_kp1, z_k, 0)
    Z_k.append(z_kp2)

    z_k = z_kp1
    z_kp1 = z_kp2


plt.title("減衰：制御なし")
plt.plot(Z_k)
plt.savefig("chap4_mpc_no_control.png")
plt.show()
```

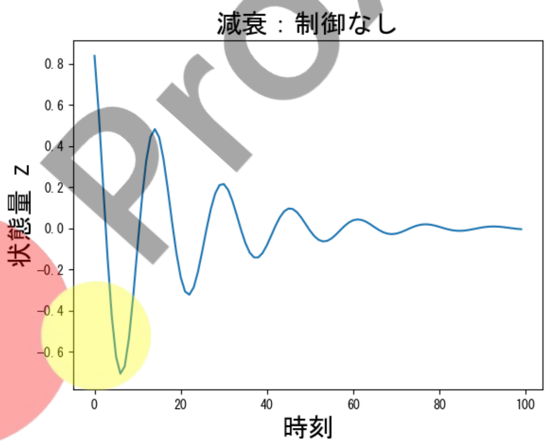

図 4.1: 制御入力 0 の時の振動子の様子

この振動子をモデル予測制御で $z = 1.0$ に止めてみましょう。

4.2　最適化問題の定式化

まずは各種パラメータを設定しましょう。

```
n_horizon = 10 # ホライズン長
x_k = np.array([0, 0]).T # 初期値
z_ref = 1 # 参照軌道（z=1）
```

ダイナミクスを一階の差分方程式で表す必要があるため、新たな変数 $x_k = (z_{k+1}, z_k)^T$ を導入して次のように状態方程式を書き換えます。

$$
\begin{aligned}
x_{k+1}^{(0)} &= 1.750x_k^{(0)} - 0.902x_k^{(1)} - 0.01{x_k^{(1)}}^3 + u_k := f^{(0)}(x_k, u_k) \\
x_{k+1}^{(1)} &= x_k^{(0)} := f^{(1)}(x_k, u_k)
\end{aligned}
\tag{4.2}
$$

```
_x_0 = casadi.SX.sym('x_0') # xの第0成分
_x_1 = casadi.SX.sym('x_1') # xの第1成分
_u = casadi.SX.sym('u') # 制御入力u

# xの予測軌道
_X_traj = casadi.SX.sym('x_traj', n_horizon, 2)
# uの予測軌道
_U_traj = casadi.SX.sym('u_traj', n_horizon-1, 1)

# 状態方程式
_f = casadi.Function('dynamics', [_x_0, _x_1, _u],
         [casadi.vertcat(
            oscillator_eq(_x_0, _x_1, _u),
            _x_0
        )])
```

さらに、変数の制約は

$$
\begin{aligned}
-\infty \leq &x_k \leq \infty \quad (\text{制約なし}) \\
-0.2 \leq &u_k \leq 0.2
\end{aligned}
\tag{4.3}
$$

とします。これらをまとめると、以下のコードになります。

```
def make_constraints(x_k):
    # 各種制約条件
    g = []
    lbg = []
    ubg = []

    # 初期条件の制約
    g += [_X_traj[0,:].T - x_k]
    lbg += [0, 0]
    ubg += [0, 0]

    # 状態方程式の制約
    for l in range(1, n_horizon):
        g += [_X_traj[l,:].T - \
        _f(_X_traj[l-1,0],_X_traj[l-1,1], _U_traj[l-1])]
        lbg += [0, 0]
        ubg += [0, 0]

    # 状態と入力に対する制約
    lbw = [-np.inf]*_X_traj.numel() + \
          [-0.2]*_U_traj.numel()
    ubw = [np.inf]*_X_traj.numel() + \
          [0.2]*_U_traj.numel()

    return g, lbg, ubg, lbw, ubw

g, lbg, ubg, lbw, ubw  = make_constraints(x_k)
```

コスト関数 J は状態変数とその時間変化、制御入力の時間変化の 3 つの項から次のように定義します。

$$J = 0.1 \sum_{l=0}^{T} (x_{k+l}^{(0)} - z_{\mathrm{ref}})^2 + 0.5 \sum_{l=0}^{T-1} \Delta u_{k+l}^2 + 0.1 \sum_{l=0}^{T-2} \Delta x_{l+k}^2 \tag{4.4}$$

対応するコードは

```
def make_J(_X_traj, _U_traj, z_ref):
    #コスト関数
    J = 0

    # 目標との二乗誤差
    for l in range(1, n_horizon):
        J += 0.1*(_X_traj[l, 0] - z_ref)**2

    # 速度に対する正則化
    for l in range(n_horizon-1):
        J += 0.5*(_X_traj[l+1,0] - _X_traj[l,0])**2

    # uの時間変化に対する正則化
    for l in range(n_horizon-2):
        J += 0.1*(_U_traj[l+1,0] - _U_traj[l,0])**2

    return J

J = make_J(_X_traj, _U_traj, z_ref)
```

となります。したがって、最適化問題

$$
\begin{aligned}
&\underset{\{u_{k+l}\}}{\text{minimize}} \quad J \\
&\text{subject to} \quad \begin{cases} x_{k+l+1} = f(x_{k+l}, u_{t+l}) \\ -\infty \leq x_{t+l} \leq \infty \quad (\text{制約なし}) \\ -0.2 \leq u_{t+l} \leq 0.2 \end{cases}
\end{aligned} \tag{4.5}
$$

は以下のコードで表されます。

```
def programming(_X_traj, _U_traj, x_k):
    g, lbg, ubg, lbw, ubw  = make_constraints(x_k)
    nlp = {'x': casadi.vertcat(
        casadi.reshape(_X_traj,-1,1), _U_traj),
        'f':J,
        'g':casadi.vertcat(*g)}
    sol = casadi.nlpsol('sol', 'ipopt', nlp)

    res = sol(
        x0=np.zeros(n_horizon*3-1),
        lbx=lbw, ubx=ubw,
        lbg=lbg, ubg=ubg)

    return res

res = programming(_X_traj, _U_traj, x_k)
opt = np.array(res['x'])

x_traj_res = opt[:n_horizon]
u_traj_res = opt[2*n_horizon:]
```

この結果によって得られた予測軌道は次のようになります。

```
fig, axes = plt.subplots(2,1,figsize=(8,6))

axes[0].plot(x_traj_res, c="blue", linestyle="--")
axes[0].set_title("状態予測軌道")
axes[1].plot(u_traj_res, c="red", linestyle="--")
axes[1].set_title("制御入力予測軌道")

plt.tight_layout()
plt.savefig("chap4_predict_traj.png")
plt.show()
```

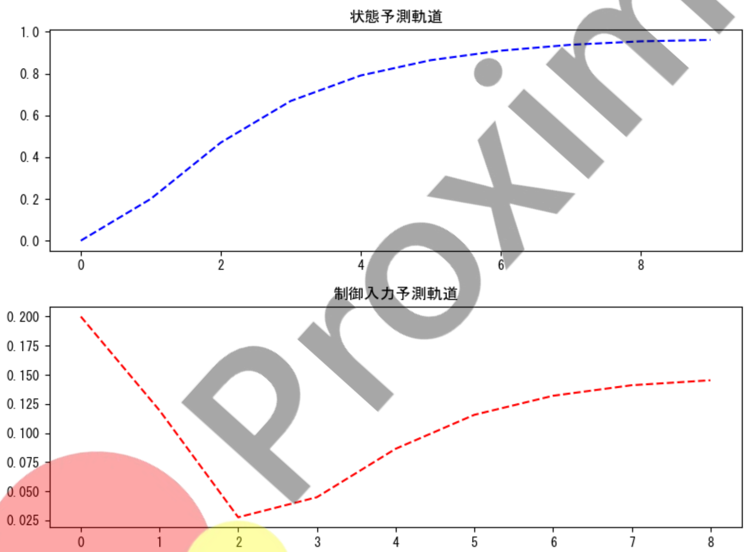
状態予測軌道
1.0
0.8
0.6
0.4
0.2
0.0
0
2
4
6
8
制御入力予測軌道
0.200
0.175
0.150
0.125
0.100
0.075
0.050
0.025
0
1
2
3
4
5
6
7
8

図 4.2: 予測軌道

4.3　モデル予測制御による制御

実際にモデル予測制御を行うには、次のコードのように前述の計算を毎時刻繰り返します。

```
import io
from PIL import Image

X_k = []
U_k = []

x_k = np.array([0, 0])
sim_time = 30

images = []

for k in range(sim_time):
    res = programming(_X_traj, _U_traj, x_k)
    opt = np.array(res['x'])
    x_traj_res = opt[:n_horizon]
    u_traj_res = opt[2*n_horizon:]

    z_tp2 = oscillator_eq(x_k[0],x_k[1],u_traj_res[0])
    X_k.append(x_k[0])
    U_k.append(u_traj_res[0])

    x_k = np.array([z_tp2, x_k[0]])

    fig, axes = plt.subplots(2,1,figsize=(12,6))

    fig.suptitle("MPC制御", fontsize=30)

    axes[0].set_title("状態量z", fontsize=20)
    axes[0].plot(X_k, c="blue")
    axes[0].scatter(len(X_k)-1, X_k[-1], c="blue")
    axes[0].plot(range(len(X_k)-1, len(X_k) + \
            x_traj_res.shape[0]-1), \
            x_traj_res, c="blue", linestyle="--")
    axes[0].set_xlim(0,sim_time+x_traj_res.shape[0]-5)
    axes[0].set_ylim(-0.1, 1.2)
```

```
    axes[1].set_title("制御入力u", fontsize=20)
    axes[1].plot(U_k, c="red")
    axes[1].scatter(len(U_k)-1, U_k[-1], c="red")
    axes[1].plot(range(len(X_k)-1, len(X_k) + \
            u_traj_res.shape[0]-1), \
            u_traj_res, c="red", linestyle="--")
    axes[1].set_xlim(0,sim_time+x_traj_res.shape[0]-5)
    axes[1].set_ylim(-0.25,0.25)

    plt.tight_layout()
    buf = io.BytesIO()
    plt.savefig(buf, format='jpg') # bufに保持
    buf.seek(0)

    dst = np.array(Image.open(buf)) # bufからの読み出し

    plt.cla()
    plt.clf()
    plt.close()
    images.append(Image.fromarray(dst))

images[0].save('chap4_descrete_MPC.gif',
            save_all=True, append_images=images[1:],
            optimize=False, duration=100, loop=0)

print("END")
```

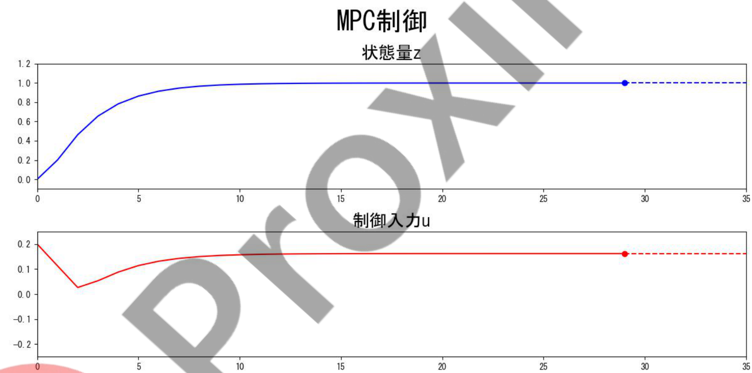

図 4.3: 離散時間モデル予測制御の様子