航空宇宙利用工学レポート

学籍番号　　18NA603
氏名　　葛巻壱成

I.　はじめに

コンピュータの高性能化と価格低下に伴い，最適制御問題をリアルタイムで解くモデル予測制御に対する関心が高まっている．そこで，当グループでは，モデル予測制御について調査し，自動車の制御シミュレーションにそれを組み込むことによって，どのような制御が行われるのかについて調べた．

II.　モデル予測制御とは

モデル予測制御とは，目標値と現在値を入力とした閉ループ系に最適制御問題の求解を組み込んだ制御手法である．古典制御や一般的なフィードバック制御では，ある定められた制御則に従ってある時点の入力を決定する．そのために，計算負荷が小さく，ある程度の誤差を有していても収束性を保証できることが利点である．しかし，制御の性能が制御則やそのゲインに依存することや，状態量や入力の制約を考慮した制御性能の最大化について課題がある．また，最適フィードフォワード制御の場合には，制御の目的を数式化することにより，それに最適化された入力プロファイルを予め計算する．一方で，事前に考慮しなかった外乱等に対しての冗長性の確保が課題である．モデル予測制御では逐次的に最適制御問題を解くために，その時点における最適な入力を常に得ることができる．これにより，冗長性とある目的に合わせた最適性を両立した制御が可能である．

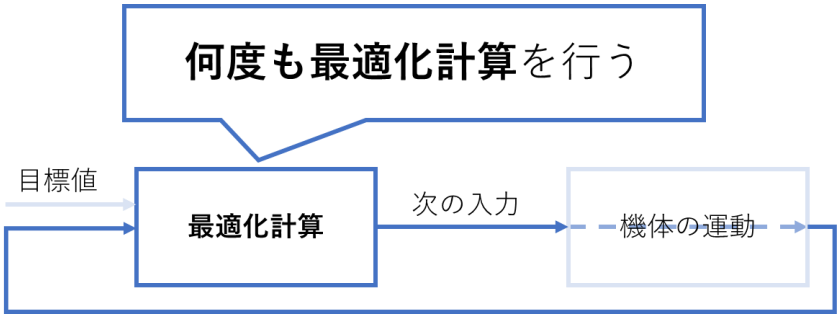

Fig. 1 モデル予測制御のイメージ

III. モデル予測制御の特徴

モデル予測制御は制約付きの最適制御問題を逐次解くという特徴がある．そのために，モデル予測制御の利点として，各時点における最適軌道を得られることが挙げられる．そのため，何らかの外乱によって軌道追従に誤差が生じたとしても，誤差を有する状態での最適軌道を再計算することができる．また，モデルや計算資源に応じて解法を選択することで，非線形なシステムや他入力・多出力システム，入力等に制約があるような系に対しても適用することができる．一方で，最適化計算そのものの計算量が多いために，モデルや解法によってはモデル予測制御を実現するために大きな計算資源が必要となる場合があることが欠点である．

IV. モデル予測制御の実装例

I. 制御対象

本稿では，多入力・多出力系の制御として，自動車の制御を行うこととした．このとき車両モデルは講義資料のものを用いた．θは車両の方向，vは速さ，ϕは操舵角である．

$$x = (x_1\ x_2\ x_3\ x_4\ x_5)^T = (x\ y\ \theta\ v\ \phi)^T$$

$$\dot{x} = \left(v\cos\theta\ \ v\sin\theta\ \ \frac{v}{l}\tan\theta\ \ a\ \ \dot{\phi}\right)^T$$

$$u = \left(a\ \ \dot{\phi}\right)^{\mathrm{T}}$$

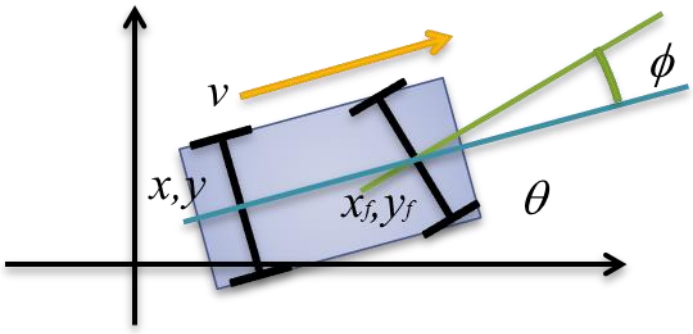

Fig. 2 車両モデル(講義資料より)

II. 制御方法

講義内のプレゼンテーションでは連続変化法によるプログラムを示したが，プログラムに何らかの問題があることを指摘された．そこで本稿では，Python 言語の凸最適化のためのモデリングツール CVXPY を用いて，逐次最適化計算を実行した．CVXPY では以下の線形システムについて最適制御問題を解くことができる．このとき，x_tは状態量，u_tは制御入力，lは評価関数である．

$$\text{minimize} \quad \sum_{t=0}^{T-1} l(x_t, u_t) + l_T(x_T)$$

$$\text{subject to} \quad x_{t+1} = Ax_t + Bu_t$$

まず，車両モデルから制約を求める．モデルは非線形方程式であるために，モデルを各時点の状態近傍で線形化してから計算することとした．状態方程式を線形化すると下式が得られる．

$$\dot{x_n} = \begin{pmatrix} vcos\theta \\ vsin\theta \\ \frac{v}{l}tan\theta \\ a \\ \dot{\phi} \end{pmatrix}$$

$$\approx \begin{bmatrix} 0 & 0 & -v_n sin\theta_n & cos\theta_n & 0 \\ 0 & 0 & v_n cos\theta & sin\theta_n & 0 \\ 0 & 0 & 0 & \frac{tan\phi_n}{l} & \frac{v_n}{lcos^2\phi_n} \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{pmatrix} x \\ y \\ \theta \\ v \\ \phi \end{pmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{pmatrix} a \\ \dot{\phi} \end{pmatrix} + \begin{pmatrix} v_n cos\theta_n \\ v_n sin\theta_n \\ \frac{v_n}{l}tan\theta_n \\ a_n \\ \dot{\phi}_n \end{pmatrix}$$

$$= A'x_n + B'u_n + C'_n$$

これより離散時間における状態量の漸化式を求めると，

$$\begin{aligned} x_{n+1} &= x_n + \dot{x_n}\Delta t \\ &= x_n + (A'x_n + B'u_t + C'_n)\Delta t \\ &= (E + A'\Delta t)x_n + (B'\Delta t)u_t + C'_n\Delta t \end{aligned}$$

これは，CVXPY によって解くことのできる形であるために，各状態について上式の各項を求めることによって，最適化計算を行った．

III. 計算例

計算するうえで，以下の条件を用いた．ここでは，位置の誤差と入力を組み合わせた評価関数が最小となるような入力を逐次求めることとした．計算環境には最近の一般的な性能のノートパソコンを用いた．また，使用したプログラムを第 VI 章に示した．

- 初期条件：位置$(-10.0, 10.0)[m]$，速度$(0.0, 0.0)[m/s]$
- 目標条件：位置$(0.0, 0.0)$[m]，速度$(0.0, 0.0)[m/s]$
- 終了条件：$x^2 + y^2 + v^2 < 1.0$
- 計算時間の刻み幅および最適計算のステップ幅：0.1[s]
- 最適計算のステップ数$\mathrm{T} = 10$(すなわち，1.0[s]後までの予測計算を行う)
- 評価関数：$l = 100\sum_{t=0}^{T-1}(x^2 + y^2) + 0.01\sum_{t=0}^{T-1}(a^2 + \dot{\phi}^2)$
- 制約条件：加速度$|\mathrm{a}| \leq 10.0[m/s^2]$，操舵角速度$\dot{\phi} \leq 3[rad/s]$
- 計算環境：Python 3.6.6 for win32@ Core i5-7300U(2.6GHz)

これにより計算した結果を Fig.3 および Fig.4 に示した．このとき，目標地点への到達時間が 11.4[s]に対して計算時間が 5.4[s]であったため，最適計算がシミュレーション上の経過時間よりも速く終了した．したがって，この制御は実行可能であると考える．Fig.1 には実際に車両が辿った軌跡を赤で示し，青線は各時点で最適計算することによって得られた予測軌道を示した．時間経過に従って，目標位置に向かう軌道が描けていることが分かる．また，シミュレーション開始からしばらくの間で目標位置の方向に予測軌道が向かわなかった．これはモデルを線形化しているために、走り出し始めの状態では操舵が効きづらいためだと考える．そのため、非線形モデルで最適制御を行うことや，小回りの利きやすい車両モデルを用いることで改善すると考える．Fig.2 には車両の速度と，加速度と操舵角の時間変化を示した．これより，加速・方向転換・減速という順で操作が行われたことが分かる．また，加速度と操舵角速度が頻繁に入れ替わるような入力となった．これより，細かに入力を変えながら車体を操作していることが分かった．

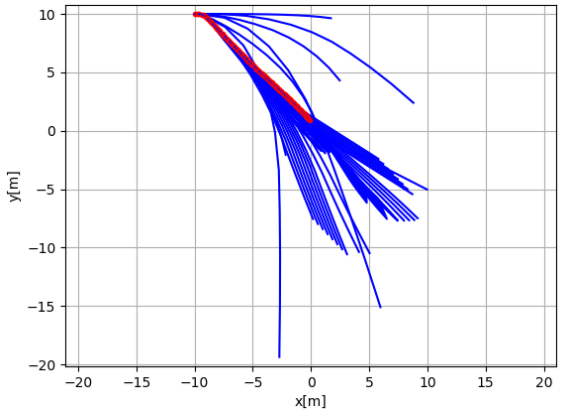

Fig. 3 運動の軌跡と各時点の予測軌道

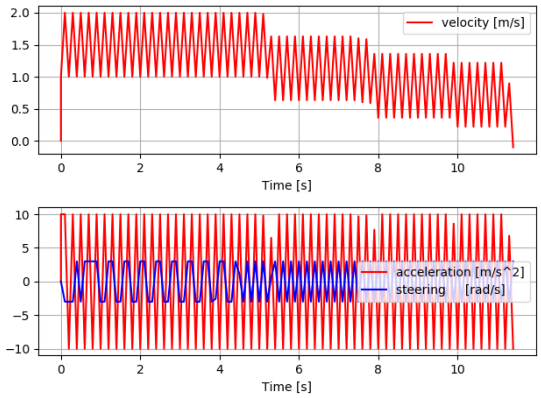

Fig. 4 速度と入力の時間変化

V. まとめ

本稿ではモデル予測制御について他の制御手法との違いや、利点と欠点について述べた．その上で，自動車のモデルを用いたシミュレーションを行い，計算時間がシミュレーション時間を下回ることや、頻繁に制御を変えながら，加速，方向転換，減速という 3 過程で車両操作を行ったことが分かった．

VI. プログラムコード

```
#! /usr/bin/python
# -*- coding: utf-8 -*-

import cvxpy
import numpy as np
from cvxpy import *
import matplotlib.pyplot as plt
from math import *
import time

dt = 0.1  # [s] discrete time
lr = 1.0  # [m]
T  = 10   # number of horizon

def LinealizedModel(xb, u, dt, lr):
    x     = xb[0]
    y     = xb[1]
    theta = xb[2]
    v     = xb[3]
    phi   = xb[4]
```

```
    a     = u[0]
    dPhi  = u[1]

    A = np.eye(xb.shape[0])
    A[0, 2] = dt * sin(theta) * -v
    A[0, 3] = dt * cos(theta)
    A[1, 2] = dt * cos(theta) *  v
    A[1, 3] = dt * sin(theta)
    A[2, 3] = dt * tan(phi) / lr
    A[2, 4] = dt * v / (lr * cos(phi) **2)

    B = np.zeros((xb.shape[0], u.shape[0]))
    B[3, 0] = dt
    B[4, 1] = dt

    C = np.zeros((xb.shape[0], 1))
    C[0] = v * cos(theta)
    C[1] = v * sin(theta)
    C[2] = v * tan(phi) / lr

    return A, B, C


def NonlinearModel(xb, u, dt, lr):
    x     = xb[0]
    y     = xb[1]
    theta = xb[2]
    v     = xb[3]
    phi   = xb[4]

    a     = u[0]
    dPhi  = u[1]

    A = xb
    A[0] += dt * cos(theta) * v
    A[1] += dt * sin(theta) * v
    A[2] += dt * tan(phi)   * v / lr
    A[3] += dt * a
    A[4] += dt * dPhi

    return A


def CalcInput(A, B, C, x, u):

    x_0 = x[:]
    x = Variable(x.shape[0], T + 1)
    u = Variable(u.shape[0], T)
```

```
    # MPC controller
    states = []

    # 予測する各区間に対して評価関数と制約の設定をする
    for t in range(T):
        cost = 100 *sum_squares(x[0:2,t+1]) + 0.001 * sum_squares(u[:,t])  # 位置の
誤差と入力が小さくなるように評価関数を設定
        constr = [x[:,t+1] == A*x[:,t] + B*u[:,t]+ C , norm(u[0,t], 'inf') <= 10,
norm(u[1,t], 'inf') <= 3.0] # 運動方程式と入力の範囲が制約条件

        # 終端コストとして速度を設定
        # if t == T:
        #     cost += 10000 * x[3,t+1]**2

        states.append( Problem(Minimize(cost), constr) )

    prob = sum(states)

    # 初期条件を制約に設定
    prob.constraints += [x[:,0] == x_0, x[3,T] == 0.0]
    prob.solve() # 凸最適化

    if prob.status != OPTIMAL:
        print("Cannot calc opt")

    return u, x, prob.value


def GetListFromMatrix(x):
    return np.array(x).flatten().tolist()


def Main():
    x0 = np.matrix([-10.0, 10.0, 0.0, 0.0, 0.0]).T  # [x,y,the, v, phi]
    x = x0
    u = np.matrix([0.0, 0.0]).T  # [a,dPhi]
    plt.figure(num=None, figsize=(12, 12))

    log = np.zeros((8,1))
    state = np.vstack((0.0,x[:,0],u[:,0]))
    log[:,0] = np.squeeze(np.asarray(state))

    start = time.time()
    for i in range(1000):
        A, B, C = LinealizedModel(x, u, dt, lr)        # 現在の状態周りで系を線形化
        ustar, xstar, cost = CalcInput(A, B, C, x, u)  # 最適入力問題を解いて入力を
求める

        # 入力の計算
        u[0, 0] = float(ustar[0, 0].value)
```

```
        u[1, 0] = float(ustar[1, 0].value)

        x = NonlinearModel(x, u, dt, lr) # 1ステップだけシミュレーション

        # グラフ描画用はじめ

        state = np.vstack((i*dt,x[:,0],u[:,0]))
        log   = np.hstack((log,np.asarray(state)))

        np.set_printoptions(precision=2, floatmode='fixed', suppress=True)
        print(log[:,i+1].T)

        # plt.plot(0,0, 'xb')
        #               plt.plot(GetListFromMatrix(xstar.value[0,                :]),
GetListFromMatrix(xstar.value[1, :]), '-b') # 状態ホライズン
        # plt.plot(x[0], x[1], '.r') # 実際の位置
        # plt.axis("equal")
        # plt.xlabel("x[m]")
        # plt.ylabel("y[m]")
        # plt.grid(True)
        #
        # plt.pause(0.0001)

        # グラフ描画用おわり

        # 終了判定はじめ

        val = x[0] **2 + x[1] ** 2 + x[3] ** 2
        if (val < 1.0):
            print("Goal")
            break

        # 終了判定おわり

    # plt.show()

    end = time.time()
    duration = end - start
    print(f'duration: %d [s]', duration)

    # 速度・加速度・ハンドル角速度の表示
    plt.figure()

    plt.subplot(2,1,1)
    plt.plot(log[0,:],log[4,:],'r',label='velocity [m/s]')
    plt.xlabel('Time [s]')
    plt.legend()
    plt.grid(True)

    plt.subplot(2,1,2)
```

```
    plt.plot(log[0,:],log[6,:],'r',label='acceleration [m/s^2]')
    plt.plot(log[0,:],log[7,:],'b',label='steering     [rad/s]')
    plt.xlabel('Time [s]')
    plt.legend()
    plt.tight_layout()
    plt.grid(True)

    plt.show()


if __name__ == '__main__':
    Main()
```

VII. 参考文献

1. MyEnigma(2016)"Model Predictive Control: モデル予測制御入門"(URL: https://myenigma.hatenablog.com/entry/2016/07/25/214014). 2018 年 8 月 12 日閲覧
2. Steven Diamond ほか (2018) "CVXPY documentation" (URL: http://www.cvxpy.org). 2018 年 8 月 12 日閲覧