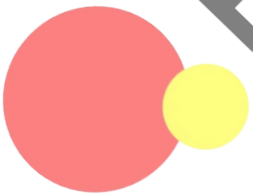
PythonとCasADiで学ぶモデル予測制御

株式会社Proxima Technology



目次

第1章　制御とは？

制御（英：control）とは機械や様々な対象のシステムを意図した通りに動かすという意味の単語です。制御対象になる得るシステムとしては

- 機械的システム：例）ロボット
- 電気的システム：例）発電機
- 流体的システム：例）核融合炉
- 化学的システム：例）石油化学プラント
- 金融的システム：例）アルゴリズムトレード
- 生物的システム：例）細胞培養器

等、幅広く上げることが出来ます。

このように一見すると全く異なる数々のシステムに対して、実は普遍的な理論を構築することが出来、それは一般に制御工学、あるいは制御理論という名前で呼ばれています。本書で取り上げるモデル予測制御という制御手法もその枠組みの中の一つの手法に当たります。

この章ではまず、制御の歴史と近代の制御において中心的な役割を果たしてきたPID制御とその問題点について紹介し、第2章で説明するモデル予測制御の必要性について説明します。

1.1　機械の発明と制御の歴史

人力や畜力以外の動力を持つ機械を設計者の意図通りに動かそうとすると、制御という考え方が必要になります。最も古い制御の例としてよく知られているのが、紀元前 3 世紀ごろのアレクサンドリアのクテシビオスによる水時計です。通常水時計では水位が下がるとともに流速が低下していくとが大きな問題となるのですが、この時計では”浮き”をうまく使うことで水位の変動に応じて人形についた針の位置と時計自体の回転をうまく調整し、流速の変化をうまく打ち消しています。

実社会において制御が本格的に重要な役割を果たし始めるのは蒸気機関が発明され、普及し始めた産業革命以降で、1778 年にジェームズ・ワットによって発明された蒸気機関向けの調速機（ガバナー）がその最初の例と言えるでしょう。ワットによる調速機は機械の回転軸の速さと、蒸気スロットルのバルブの開度を逆に比例するように設計されているため、もし回転軸の回転が速すぎればスロットルが狭くなり出力が低下し、逆に遅すぎればスロットルが広くなり出力が上がります。このような方法で、軸の回転速度を一定に調整することが出来るようになりました。

その後、蒸気機関や内燃機関、電動機などの動力の普及とともに社会のいたるところで制御の必要性が拡大していき、一方でコンピューターの進歩によって簡単かつ安価に制御ロジックを様々な機械に組み込むことが可能となりました。

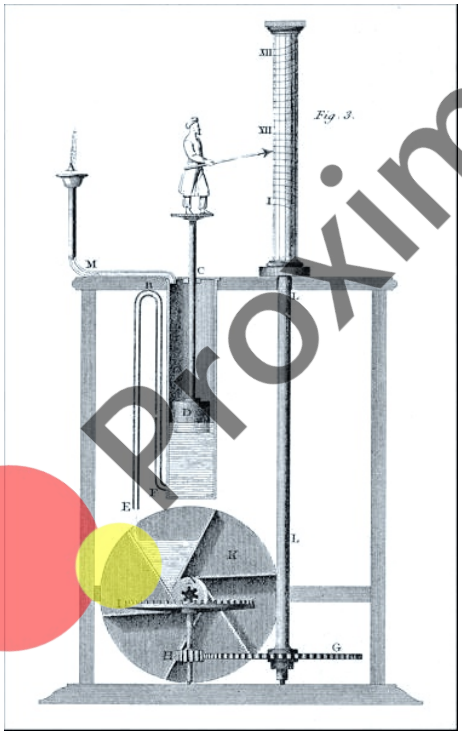


図 1.1: クテシビオスによる水時計

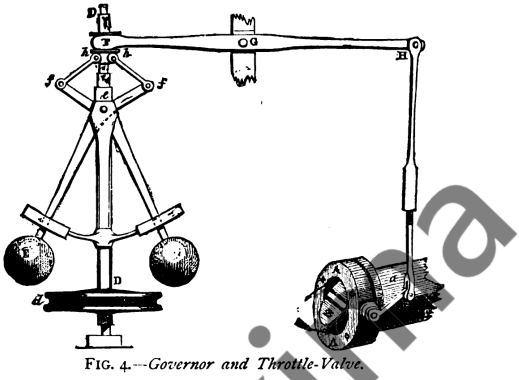


図 1.2: ワットによる調速機（ガバナー）

1.2　フィードフォワード制御とフィードバック制御

制御には一般的にフィードフォワード制御（開ループ制御）とフィードバック制御（閉ループ制御）と呼ばれる 2 種類の手法が存在します。

フィードフォワード制御は最も単純な制御手法で、システムのアウトプット（状態）とは無関係に事前に決められたルールに基づいて実行されます。例えば一昔の炊飯器などは組み込まれたマイコンに事前に調理プログラムが書き込まれており、お米の状態とは無関係にヒーターの電流値を調理プログラムに則って順次調整するのみでした。

フィードバック制御はシステムのアウトプットに応じて制御入力が変動します。例えばエアコンなどは室内の気温を監視しており、設定された気温と実際の室温との差に応じて出力を変動させることで室温の調整を行っています。

このようにフィードバック制御はシステムのアウトプット（状態）を見ながら制御を行うため、環境の変化や様々な外乱に対して対応

することが可能であり、高度な制御を行うときにはほとんど確実に必要になってくる技術となります。その中でも PID 制御という手法はフィードバック制御の王様のような存在で、世の中に最も普及しています。

1.3 PID 制御とその問題点

PID 制御は制御対象の出力 $y(t)$ と設定された目標値 $r(t)$ に対してその誤差

$$e(t) = y(t) - r(t) \tag{1.1}$$

を入力 $u(t)$ の計算に用います。PID の名前の由来は比例（Proportional）、積分（Integral）、微分（Differential）のそれぞれの頭文字を取ったもので、これらは制御入力 $u(t)$ を計算するときに誤差 $e(t)$ を用いて計算される 3 つの項を表しています。

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau) d\tau + K_d \frac{de(t)}{dt} \tag{1.2}$$

比例項は誤差を小さくする“ばね”のような働きをし、微分項は目標値 $r(t)$ 付近に近づいた際にオーバーシュートさせないように減速させる効果を、積分項は定常的に残ってしまった誤差を時間をかけて小さく修正していくような効果を持ちます。

重要なことは PID 制御には制御対象の性質が一切いらないということ、すなわちモデルフリー制御であるということであり、そのおかげで気軽に導入することが可能となります。

一方で、PID 制御には以下のような弱点が存在しています。

- むだ時間に弱い
- 1 変数のみしか扱えない（SISO 系）

- 離散変数が扱えない
- 最適化が出来ない
- 制約条件を扱えない
- 長期的な計画が必要な制御が出来ない

PID 制御は現在の誤差の情報 $e, \dot{e}$ とその現在までの累積 $\int edt$ だけしか用いないため、ある意味では非常に“視野の狭い”制御となってしまいます。このことは例えば制御対象に比較的大きなむだ時間が含まれる際に PID 制御が苦戦してしまう原因となります。

また、PID 制御は基本的には一入力一出力系（SISO）のみを扱うことを前提としており、例えばマルチループの PID を組むことで無理やり多入力多出力系（MIMO）を制御しようとしても複数の状態変数と制御入力の間に複雑な相関関係が存在していれば激しくハンチングを起こしてしまう可能性があります。

また、定式化でも明らかなようにエネルギーなどを最適化・最小化することもできず、離散変数や制約条件を自然な形で取り組むことも不可能です。

さらに、例えば倒立振子の振り上げのようなシチュエーションで「いったん逆方向に台車を動かしてエネルギーを稼いでから振り上げを行う」というようなある程度長期間にわたるシーケンシャルな動作を実現させることも難しいです。

このように、PID 制御には簡単に導入できるという大きなメリットがある一方で、実用の場面ではその能力に限界を感じる場合も少なくはありませんが、これらの問題はモデル予測制御を導入することで解決させることが可能になります。

第2章　モデル予測制御とは？

2.1　モデル予測制御考え方

モデル予測制御（MPC:Model Predictive Control）は名前の通り制御対象のモデルを用いた未来予測に基づく制御法で、PID 制御はモデルフリーであったのに対し、こちらはモデルベースな制御手法となります。予測に基づく制御という意味を理解するため、ここでは自動運転を例に PID 制御と MPC を比較してみましょう。

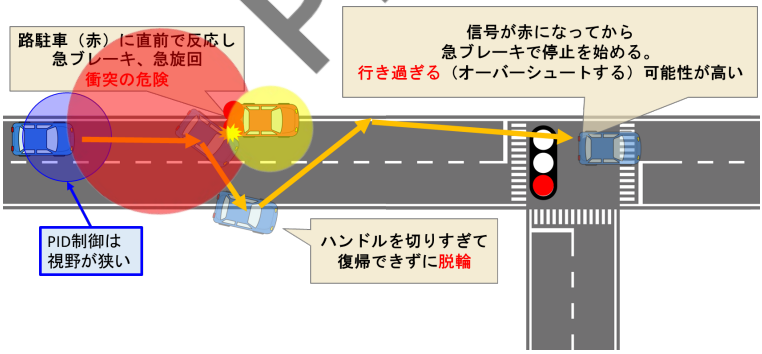


図 2.1: PID 制御での運転のイメージ

PID 制御のイメージは未来の状態を見通さずに現在時点で見えたものに反応するような“下手”な運転に対応します。ペーパードライ

バーの運転のように、路駐車に直前で気づいて急ブレーキ、急ハンドルを切ったり、あるいはハンドルの切りすぎで道路からはみ出てしまったり、信号も赤になってから突然急ブレーキをかけて行き過ぎてしまったりといった場当たり的な運転を行ってしまいます。

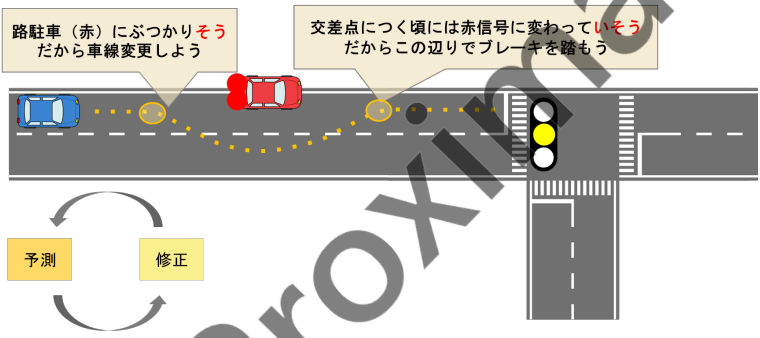


図 2.2: モデル予測制御での運転のイメージ

一方でモデル予測制御での運転のイメージは常に未来を予測しながら「かもしれない運転」を行う“賢い”運転に対応します。路駐車の大分手前からその存在を考慮に入れて運転の軌道のパターンを生成し、また、信号も赤に変わる前からどのあたりで変わるかを考え、停止線からはみ出ないようにするにはどのタイミングでブレーキを踏めばいいか、等々様々な未来に起こりうることを常に予測しながら運転を行います。このように熟練のドライバーが行っているようなことを実際に自動車に限らず機械全般の制御に適用するのがモデル予測制御という制御技術です。

2.2 離散時間モデル予測制御

まずは、モデル予測制御の考え方を学ぶ上で離散時間システムを扱ってみましょう。なお本書では簡単のために、各時刻で全状態変数が観測可能で、さらにシステムは時不変なものを仮定します。

この時離散時間システムは、時刻を表す非負の整数 $k \in \mathbb{Z}_{\geq 0}$ と状態変数 $x_k \in \mathbb{R}^n$、制御変数 $u_k \in \mathbb{R}^m$ と非線形関数 $f : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^n$ を用いて次のような状態方程式で表すことが出来ます。

$$x_{k+1} = f(x_k, u_k) \tag{2.1}$$

さて、このシステムで現在時刻 k において $k+T$ まで予測することを考えましょう。$T \in \mathbb{Z}_{>0}$ はホライズン長と言い、予測する時間幅の長さを表します。時刻 k における状態が x_k だとして、その時の制御入力が $\hat{u}_k$ であれば時刻 $k+1$ の状態は

$$\hat{x}_{k+1} = f(x_k, \hat{u}_k) \tag{2.2}$$

となります。さらに時刻 k から $k+T-1$ までの制御入力の系列 $\{\hat{u}_k, \hat{u}_{k+1}, \dots \hat{u}_{k+T-1}\}$ が与えられれば各時刻の状態は次のように計算できます。

$$\begin{aligned} \hat{x}_{k+2} &= f(\hat{x}_{k+1}, \hat{u}_{k+1}) = f(f(x_k, \hat{u}_k), \hat{u}_{k+1}) \\ \hat{x}_{k+3} &= f(f(f(x_k, \hat{u}_k), \hat{u}_{k+1}), \hat{u}_{k+2}) \\ &\vdots \end{aligned} \tag{2.3}$$

このようにして得られた予測軌道 $\{\hat{x}_{k+1}, \hat{x}_{k+2} \dots \hat{x}_{k+T}\}$ と制御入力の系列 $\{\hat{u}_k, \hat{u}_{k+1} \dots \hat{u}_{k+T-1}\}$ を図示すると下図のようになります。

ここで、新しく参照軌道 $r(l|k)$ という物が登場していますが、これが私たちが実現させたいと考える理想的な状態変数 x の軌道に対応し

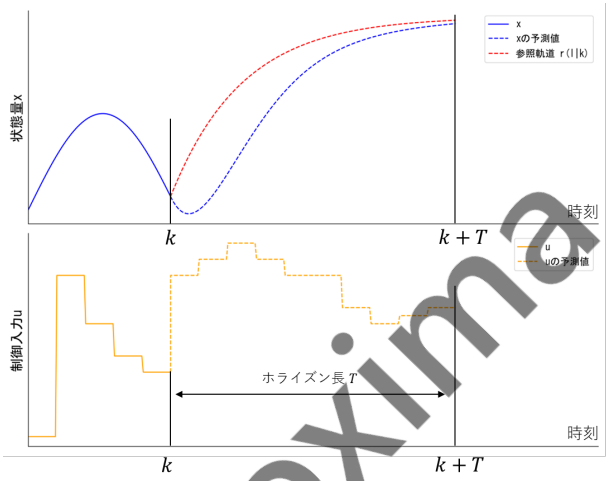


図 2.3: モデル予測制御の概念図

ます。ただ、参照“軌道”という名前ではありますが一定値を取るものでもよく、実際に本書で扱う事例は $r(l|k) \equiv r(\text{const.})$ という形での制御になります。

それでは実際にどのような方法で状態の予測軌道 $\{\hat{x}_{k+l}\}$ と参照軌道 $\{r(l|k)\}$ を近づければよいのでしょうか？制御入力の系列 $\{\hat{u}_{k+l}\}$ を上手く選んでやれば近づけることが出来るはずですが、モデル予測制御ではこれを最適化問題の枠組みとしてとらえることで解きます。すなわち、最も簡単な定式化を考えると以下のような最適化問題を解けばいいことになります。

$$\begin{aligned} \underset{\hat{u}_k, \hat{u}_{k+1}, \ldots, \hat{u}_{k+T-1}}{\text{minimize}} \quad & J = \sum_{l=1}^{T} \left(\hat{x}_{k+l} - r(l|k)\right)^2 \\ \text{subject to} \quad & \hat{x}_{k+l+1} = f(\hat{x}_{k+l}, \hat{u}_{k+l}) \end{aligned} \tag{2.4}$$

この最適化問題を解くことで得られた制御入力の系列 $\{\hat{u}_{k+l}\}$ の内、最初の u_k のみを取り出して現在の入力とします。言い換えると残りの $u_{k+1}, \ldots, u_{k+T-1}$ は制御には直接使われず捨てられることになります。（実際には次の時刻 $k+1$ での最適化計算の際に、初期値として再利用されます。）このような最適化計算を各時刻 k 毎に延々と繰り返して行うため、モデル予測制御は実時間最適化といった呼ばれ方もします。

ただ、先ほどの定式化はあまりにも簡単で、実際にこのようなコスト関数 J に対する最適化問題を解いて得られた u_k を制御入力に用いるとハンチングを起こしてしまったり、制御入力が大きく振動してしまったりするため様々な正則化を入れる必要があります。また、状態量や制御入力に対する制約も存在するので、離散時間のモデル予測制御の最適化問題はより一般的には次のような形で与えられます。

$$
\begin{aligned}
&\underset{\hat{u}_k, \hat{u}_{k+1}, \ldots, \hat{u}_{k+T-1}}{\text{minimize}} \quad J = \phi(\hat{x}_{k+T}) + \sum_{l=0}^{T-1} L_t(\hat{x}_{k+l}, \Delta\hat{x}_{k+l}, \hat{u}_{k+l}, \Delta\hat{u}_{k+l}) \\
&\text{subject to} \quad \begin{cases} \hat{x}_k = x_k \quad \text{(初期条件)} \\ \hat{x}_{k+l+1} = f(\hat{x}_{k+l}, \hat{u}_{k+l}) \\ x_{lb} \leq \hat{x}_{k+l} \leq x_{ub} \\ u_{lb} \leq \hat{u}_{k+l} \leq u_{ub} \end{cases}
\end{aligned}
\tag{2.5}
$$

ただし、$\Delta\hat{x}_{k+l} = \hat{x}_{k+l} - \hat{x}_{k+l-1}$ は状態の時間変動、$\Delta\hat{u}_{k+l} = \hat{u}_{k+l} - \hat{u}_{k+l-1}$ は制御入力の時間変動、ϕ は終端コストを表します。また、$\hat{x}_k = x_k$ と $\hat{u}_{k-1} = u_{k-1}$ は予測値ではなく実際に実現した値となります。

2.3　連続時間モデル予測制御 †

連続時間システムの場合は差分方程式ではなく、微分方程式によってシステムのダイナミクスは記述されます。本書では離散時間の場合と同様に全状態変数は観測可能で、さらにシステムは時不変なものを仮定します。この時、連続時間システムは時刻を表す実数 $t \in \mathbb{R}$ と状態変数 $x_t \in \mathbb{R}^n$、制御変数 $u_t \in \mathbb{R}^m$ と非線形関数 $f : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^n$ を用いて次のような状態方程式で表すことが出来ます。

$$\dot{x}_t = f(x_t, u_t) \tag{2.6}$$

また、この時の最適化問題は次のように与えることが出来ます。

$$\begin{aligned} \underset{\{\hat{u}_{t+\tau}|\tau\in[0,T]\}}{\text{minimize}} \quad & J = \phi(\hat{x}_{t+T}) + \int_0^T L_t(\hat{x}_{t+\tau}, \partial_t\hat{x}_{t+\tau}, \hat{u}_{t+\tau}, \partial_t\hat{u}_{t+\tau})d\tau \\ \text{subject to} \quad & \begin{cases} \partial_t\hat{x}_{t+\tau} = f(\hat{x}_{t+\tau}, \hat{u}_{t+\tau}) \\ x_{lb} \le \hat{x}_{t+\tau} \le x_{ub} \\ u_{lb} \le \hat{u}_{t+\tau} \le u_{ub} \end{cases} \end{aligned} \tag{2.7}$$

この最適化問題は連続的な関数 u_t に対する汎関数 J の最小化であり、有限個の変数しか扱えないコンピューターでこれを直接解くことはできません。したがって、何らかの方法で離散化を行う必要ががありますが、その方法については 5,6 章で詳しく議論することにします。

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、Ubuntu 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 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 Anacoonda 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。

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')
print(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

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

```
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: 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、本書の枠組み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, 00],
 [00, @1, 00, 00],
 [00, 00, @1, 00],
 [00, 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 非線形最適化問題

CasADi の NLP（非線形最適化問題）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。

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

（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 nlpsol を使って解いてみましょう。

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

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**2 + 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:  [[ 0.92387953]
 [-0.38268343]]
```

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

levs = 10 ** np.linspace(-2, 2.1, 14)

fig, ax = plt.subplots(figsize=(8,6))

ax.scatter(x_opt[0], x_opt[1], c="red")
cs = ax.contour(X,Y,Z,norm=LogNorm(),levels=levs)

ax.add_patch(plt.Circle(xy=(0,0),radius=1,fill=False))
fig.colorbar(cs)
plt.savefig("NLP_2D.png")
plt.show()
```

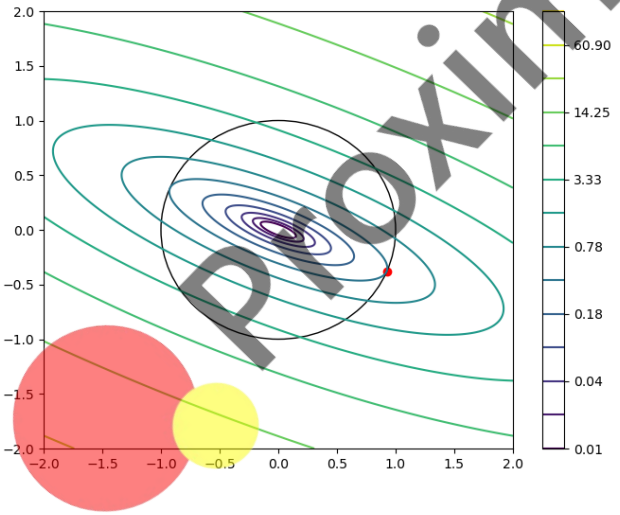


図 3.1: IPOPT による最適化の結果

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）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() の使い方を学びましょう。

次にような微分方程式と初期条件を考えます。

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

この方程式の解は明らかに $x(t) = e^{-2t}$ ですが、これは casadi.integrator() によって次にように数値的に計算できます。

```
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)  #真の解

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()
```

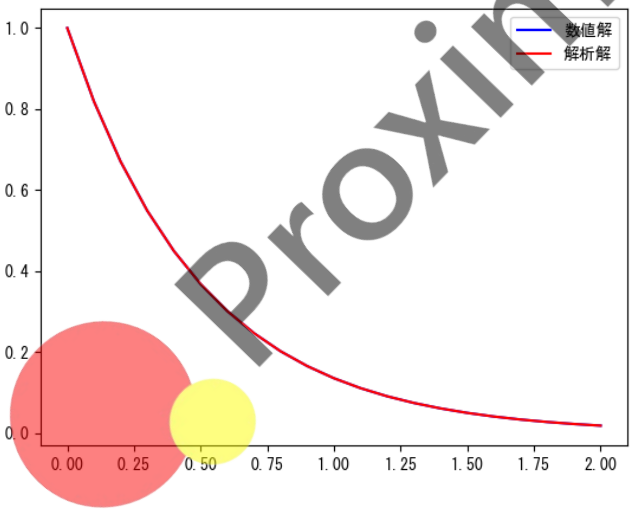


図 3.2: 解析解と数値解の比較

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()
```

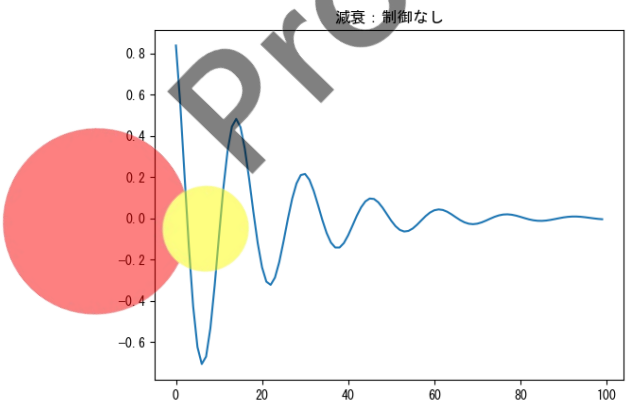



図 4.1: 制御入力 0 の時の振動子の様子

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
x0 = np.array([[0,0]]).T #初期値
z_ref = 1 #参照軌道
```

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。

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

```
_x_0 = casadi.SX.sym('x0')
_x_1 = casadi.SX.sym('x1')
_u = casadi.SX.sym('u')

_X_traj = casadi.SX.sym('x_traj', n_horizon, 2)
_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

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

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。

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

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

    # の時間変化に対する正則化u
    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()
```

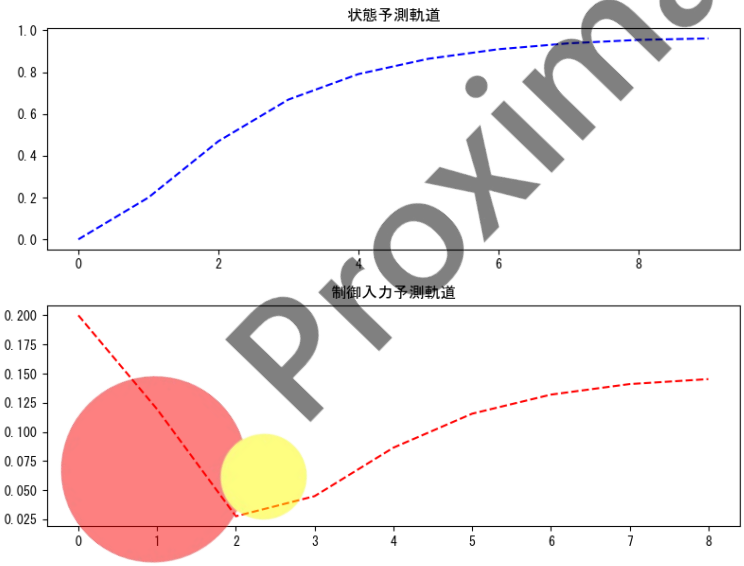


図 4.2: 予測軌道

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("制御MPC", 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') # に保持buffer
        buf.seek(0)

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

        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")
```

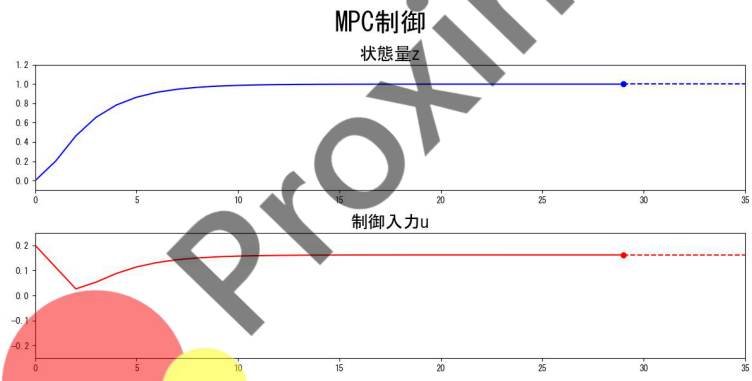


図 4.3: 離散時間モデル予測制御の様子