Rapid Prototyping 制御系開発において 自動微分を実現するためのコード書き換え手法

—Just In Time 線形近似モデルとディーゼルエンジン吸排気モデルへの適用—

浅井 徹*・仲田 勇人**・松永 彰生**

A Method to Rewrite Codes for Automatic Differentiation
on Rapid Prototyping Platforms for Control System Design

—Just In Time Linearly Approximated Models and Its Application to Air Flow Model of Diesel Engines—

Toru ASAI*, Hayato NAKADA** and Akio MATSUNAGA**

Linearly approximated systems are useful for controlling nonlinear systems. In order to obtain linearly approximated systems, derivatives of right hand side of state equations need to be computed around some operating points. To obtain the values of the partial derivatives, the automatic differentiation is useful. Although the automatic differentiation enables us to compute derivatives efficiently and accurately, tools for automatic differentiation may not be useful for rapid prototyping control system design. We therefore propose a method to rewrite source codes along the ideas of automatic differentiation. The proposed method is easy to understand and the obtained source codes remain readable. Consequently, the proposed method enables us systematically rewriting codes without sacrificing the merits of rapid prototyping developments. We also apply the proposed technique to an air flow model of diesel engines and examine the effectiveness of the technique.

Key Words: automatic differentiation, just in time linear approximation, diesel engines

1. はじめに

自動車用エンジンなどのように、広い運転領域を扱う制御対象を制御する場合には、対象の非線形性を陽に考慮する必要がある。しかしながら、対象の非線形性をそのまま扱う非線形制御則は理論が難解であり、適用対象も限られてしまう。そのため、拡張カルマンフィルタ¹⁾のように、対象の線形近似システムに基づいて線形制御則を適用することは、現在でも有力な制御手法である。

線形近似に基づいて非線形系の制御を行なう場合、各サンプル時刻で状態方程式右辺の偏微分係数を計算する必要がある。このとき、右辺の関数の偏導関数を求めることができれば、必要な引数を与えることで偏微分係数を求めることができる。しかしながら、対象が複雑になると偏導関数の導出が困難になる。特に、状態方程式右辺の計算アルゴリズムに繰

り返しや条件分岐などが含まれる場合には、偏導関数の導出はきわめて困難である。

このような場合でも、前進差分などに基づく数値微分を用いれば複雑な対象に対しても偏微分係数を求めることができる。しかしながら、数値微分では変数の適切な微小変化量を事前に知ることは困難である、得られた偏微分係数の精度が不明確である、などの問題が生じる。微小変化量を変えながら数値微分を繰り返し実行すれば、ある程度、精度を見積ることは可能である。ただし、その場合には偏微分係数の計算時間が増大するため、組み込み制御に用いることは困難となる。

以上の状況に対し、自動微分^{2)~4)}を用いれば複雑な対象に対しても効率的かつ精度よく偏微分係数を計算することができれば、幅広い動作領域をもつ非線形な対象に対して線形近似に基づく高度な制御を適用することが容易になる。実際、自動微分の制御への応用も報告されている^{5)~9)}。

自動微分に関しては、関連するツール¹⁰⁾が開発されており、自動微分を用いることは容易になっている。一般に、自動微分はソースコードの自動変換やオペレータオーバーローディングによって実現されている。一方、制御系の開発効率を高めるために、近年では Rapid Prototyping によって制御系の開発が行なわれている。しかしながら、Rapid Prototyping

* 名古屋大学大学院工学研究科 名古屋市千種区不老町

** トヨタ自動車(株)東富士研究所 裾野市御宿 1200

* Graduate School of Engineering, Nagoya University, Furo-cho, Chikusa-ku, Nagoya

** Higashi-fuji Technical Center, Toyota Motor Corporation, 1200 Mishuku, Susono

(Received June 10, 2016)

(Revised October 20, 2016)

設計環境で用いられるプログラミング言語^(注 1)ではオペレータオーバーローディングを利用できない場合が多い。そのため、Rapid Prototyping で自動微分を用いる場合には、コード変換に基づくものを利用することになる。

コードの自動変換を行なうと、一般に得られるコードの可読性は低いものとなる。また、組み込み制御では微分係数の計算を含むすべての計算に対してゼロ割回避や計算結果の上下制限約を設定するなどの保護処理を行なう必要がある。よって、自動変換を用いると、これらの保護コードを可読性の低いコードに追加し、そのコードに対してデバッグ作業を行なわざるを得なくなる。さらに、元のソースコードの変更に伴いコードの再変換を行なうと、ゼロ割回避などのコードを再度追加する必要が生じる。これらの作業は、Rapid Prototyping の主目的である迅速な開発を阻害するものとなる。以上の理由から、とくに組み込み制御の場合には、自動微分ツールがそもそも利用できないか、あるいは利用できてもコード開発の効率が悪化し、Rapid Prototyping の意義を損なわせる結果となる。

そこで本論文では、自動微分を実現するようにモデルのソースコードを手作業で見通しよく書き換える方法を提案する。提案手法は全微分と同形式のコードを追加することで書き換えを行なうものである。理解が容易である。また、元のソースコードと同程度の可読性を維持しつつ書き換えることが可能なので、ゼロ割回避などのコードの追加も容易である。また、元のソースコードを変更する場合、それに伴うコード変更は対応する微分係数の計算部分に限定される。以上のようにして自動微分を実現すれば、組み込み制御において必要に応じて Just In Time で線形近似システムを求めることができる。このような手法の有効性を、本論文ではディーゼルエンジン吸排気モデルに適用することで検証する。

本論文の構成は以下のとおりである。まず、2 章で本論文の提案手法を示す。3 章では提案手法をディーゼルエンジン吸排気モデルへの適用した結果を示す。

2. 自動微分に基づく偏微分係数の計算

2.1 線形近似システムと自動微分

本論文では非線形な対象に対して線形近似に基づく制御系設計法を適用することを想定し、対象の線形近似システムを得ることを考える。対象のモデルとしてつぎの離散時間非線形システム

$$x_{k+1} = f(x_k, w_k, u_k) \quad (1)$$

$$z_k = g(x_k, w_k, u_k) \quad (2)$$

が与えられた場合を考える。ただし、 $k \in \mathbf{N}$ は時間、 $x_k \in \mathbf{R}^n$ 、 $w_k \in \mathbf{R}^{m_1}$ 、 $u_k \in \mathbf{R}^{m_2}$ 、 $z_k \in \mathbf{R}^p$ はそれぞれ状態、外生入力、制御入力、制御出力である。また、 f, g はいずれも C^1

級の関数であるとする。なお本論文で想定する外生入力未知外乱ではなく、エンジンの運転条件（エンジン回転数、燃料噴射量）など、外部から人為的に与えることのできるものである。

ここで、(1), (2) 式のシステムに対して、ある動作点が与えられた場合を考える。すなわち $x_* = f(x_*, w_*, u_*)$ を満足する x_*, w_*, u_* が与えられた場合を考える。このとき、動作点周りの線形近似システムは次式で与えられる。

$$\xi_{k+1} = A \xi_k + B \nu_k \quad (3)$$

$$\eta_k = C \xi_k + D \nu_k \quad (4)$$

ただし、

$$\xi_k = x_k - x_*, \quad \nu_k = u_k - u_*, \quad (5)$$

$$\eta_k = y_k - y_*, \quad y_* = h(x_*, u_*), \quad (6)$$

$$A = \frac{\partial f}{\partial x}, \quad B = \frac{\partial f}{\partial u}, \quad C = \frac{\partial h}{\partial x}, \quad D = \frac{\partial h}{\partial u} \quad (7)$$

であり、(7) 式の偏微分係数はいずれも $x = x_*, w = w_*, u = u_*$ で評価した値である。

上記のように、非線形システムの線形近似システムを求めるためには、 f, g の偏微分係数を求める必要がある。関数 f, g の計算が複雑な場合、それらの偏導関数を陽に書き下すことは困難である。たとえば、本論文で取り上げるディーゼルエンジン吸排気モデルの場合、 f, g の計算を行なうソースコードには条件分岐、繰り返し、関数呼び出しが含まれる。さらに、そのソースコードの行数は MATLAB の m-ファイルで 916 行である。このように、偏導関数を陽に書き下すことは多くの場合、困難である。

f, g の計算が大規模な場合でも、数値微分に基づいて偏微分係数を近似的に求めることは可能である。しかしながら、数値微分を用いることは計算負荷や精度の点から望ましくない。

これらの問題は、自動微分^{2)~4)}と呼ばれる手法を用いれば解消することができる。自動微分は関数値を求めるプログラムから、その偏微分係数を同時に計算するプログラムを自動的に生成する手法である。通常、自動微分はオブジェクト指向プログラミング言語のオペレータオーバーローディングを用いる、あるいはソースコードの自動変換を行なうなどの手法によって実現されている。

しかしながら、本論文で対象とする Rapid Prototyping に用いられるプログラミング言語は、そもそもオブジェクト指向プログラミングの機能を有していないことが多い。そのため、オペレータオーバーローディングに基づく方法を活用することはできない。

一方、ソースコードの自動変換を行なうと、一般に変換後のコードの可読性は悪化する。たとえば、C 言語で書かれた以下のコード

```
double my_fun(double x)
{
    double y;
```

(注 1) たとえば MATLAB/Simulink 上に実装されている Embedded Matlab Language¹¹⁾など

```
double my_fun_d(double x, double xd, double *my_fun) {
    double y;
    double yd;
    double result1;
    double result1d;
    result1d = (3*x == 0.0 ? 0.0 : 3*xd/(2.0*sqrt(3*x)));
    result1 = sqrt(3*x);
    yd = xd/(1.0+x*x) + result1d;
    y = atan(x) + result1;
    *my_fun = y;
    /* main part */
    return yd;
}
```

Fig. 1 The code transformed by TAPENADE

```
y = atan(x) + sqrt(3*x); /* main part */
return y;
}
```

を TAPENADE¹²⁾で変換すると Fig. 1 のコードが得られる。ただし、スペースを節約するため、Fig. 1 では自動変換で追加されたコメントは割愛した。Fig. 1 のコードのように、一般に変換されたコードは関数値の計算部分のみに限定しても、もとのコードより可読性が悪化する。可読性が悪化すれば、それだけコードの開発効率も悪化する。

またさらに、組み込み処理に用いる必要上、ゼロ割などの深刻なエラーを回避することが必要不可欠である。このとき、関数値の計算ではゼロ割が生じなくても、その微分係数の計算にゼロ割が生じることがある。たとえば、 $\sqrt{x}$ の x における微分係数は $\frac{1}{2\sqrt{x}}$ である。このような場合、ゼロ割の回避を行なうためのコードを変換後のコードに追加する必要がある。Fig. 1 ではゼロ割を回避するための必要最低限のコードが自動的に付加されている。しかしながら、実用上は $x = 0$ の計算のみを回避すればよいだけでなく、 $x = 0$ 近傍の大き過ぎる値を抑制するために計算値の上下限を設定する必要がある。また、 $x = 0$ での値を 0 以外にしたいときにもコードを追加する必要がある。このような保護コードは変換後に得られたコードにしか適用することができない。そのため、もとのソースコードの変更に伴ってコード変換を行なうと、その度に保護コードを追加し直す必要が生じる。通常、Rapid Prototyping では何度もソースコードを改変しながら開発を進めるため、ソースコード改変の度に同じ作業を繰り返すことになること、開発効率は著しく悪化する。

以上の理由により、コード変換に基づく自動微分を用いると Rapid Prototyping の主目的である迅速な開発を著しく阻害する結果となる。以上のような問題があるために、従来、Rapid Prototyping に自動微分を用いることは困難であった。

2.2 基本的なアイデア

Rapid Prototyping で高い開発効率を維持するためには、コードの可読性、保守性を高く保つことが重要である。しか

しながら、自動微分を用いると偏微分係数の計算が可能となるのと引き換えに、コードの可読性、保守性が犠牲になってしまう。ただし、コードの可読性、保守性の悪化はコードの自動生成に起因するものであって、偏微分係数の計算を埋め込むこと自体が本質的に問題になるわけではない。

本論文はコードの可読性、保守性を悪化させることなく偏微分係数の計算を追加する手法を提案する。ただし、本論文で提案するのはコードの自動生成ではなく、手作業による書き換えを行なう際の手法である。以下、提案手法の基本的な考え方を例を用いて説明する。

例として、変数 x, y の関数 $z(x, y) = xy$ を考える。MATLAB 関数を

```
function z = calc_z(x,y)
    z = x*y;
end
```

と定義すれば、与えられた x, y に対して $z(x, y)$ を計算することができる。ここでの目的は、関数 $\text{calc_z}(x, y)$ を書き換えることで、与えられた x, y における関数値 $z(x, y)$ だけでなく偏微分係数 $J_z(x, y)$ も同時に計算する関数

```
function [z, Jz] = calc_z_J(x,y)
```

を作成することである。求めたいものはあくまでも x, y における偏微分係数であり、偏導関数ではないことに注意されたい。ちなみに、関数 $z(x, y)$ の x, y における偏微分係数 $J_z(x, y)$ は

$$J_z(x, y) = \begin{bmatrix} \frac{\partial z}{\partial x} & \frac{\partial z}{\partial y} \end{bmatrix} = \begin{bmatrix} y & x \end{bmatrix} \quad (8)$$

である。

偏微分係数 J_z を計算するために、まず変数 dx, dy を

$$dx = \begin{bmatrix} 1 & 0 \end{bmatrix}, \quad dy = \begin{bmatrix} 0 & 1 \end{bmatrix} \quad (9)$$

とおく。そのうえで、 dz を

$$dz = y dx + x dy \quad (10)$$

と定める。ここで、 dx, dy の係数 y, x は、それぞれ $z(x, y) = xy$ の x, y に関する偏導関数である。(9), (10) 式より、

$$dz = \begin{bmatrix} y & x \end{bmatrix}$$

となるので、 dz が偏微分係数 J_z を与えている。以上より、 $\text{calc_z}(x, y)$ を

```
function [z, Jz] = calc_z_J(x,y)
    dx = [ 1 0 ]; % 追加
    dy = [ 0 1 ]; % 追加
    z = x*y;
    dz = y*dx + x*dy; % 追加
    Jz = dz; % 追加
end
```

と書き換えることにより `calc_z_J(x,y)` が得られる.

(9) 式はつぎのような手順に従ったものである. 今, 独立変数は x, y の 2 つであり, ここでは x をその 1 番目の変数, y をその 2 番目の変数とする. この順序は関数 $z(x,y)$ の引数の定義によって定まるものである. このとき, dx に 2 次元ベクトルの 1 番目の標準基底ベクトル, dy に 2 番目の標準基底ベクトルを割り当てる. これを具体的に記述したものが (9) 式である. もし, 関数 z の定義が $z(y,x) = xy$ であれば, (9) 式を

$$dy = \begin{bmatrix} 1 & 0 \end{bmatrix}, \quad dx = \begin{bmatrix} 0 & 1 \end{bmatrix}$$

とすればよい. ただし, その場合には偏微分係数の順序も逆, すなわち

$$J_z(y,x) = \begin{bmatrix} \frac{\partial z}{\partial y} & \frac{\partial z}{\partial x} \end{bmatrix} = \begin{bmatrix} x & y \end{bmatrix}$$

であることに注意されたい.

以上のことから, 偏微分係数の計算を行なうためには変数スコープの最初に独立変数に関する基底を追加し, その後, 各計算コードの直後にその全微分に対応するコードを追加すればよいことがわかる.

(10) 式の dz は, 形式上は $z = xy$ の x, y に関する全微分

$$dz = \frac{\partial z}{\partial x} dx + \frac{\partial z}{\partial y} dy = y dx + x dy \quad (11)$$

にほかならない. ただし, 数学記号の dx, dy, dz とは異なり, dx, dy, dz は (9), (10) 式によって与えられる (有限の) 変数である. (10) 式は, dx, dy を (9) 式で与えておけば, 数式処理も数値微分も行なうことなく, x, y が与えられるごとに偏微分係数 J_z を計算できることを意味している.

上記の計算は z の導関数ではなく, あくまでも与えられた x, y における偏微分係数を求めていることに注意されたい. また, 関数をこのように書き換えれば, 数値微分に伴う計算誤差は生じない. プログラムで $z(x,y)$ を計算する場合, $z = xy$ の計算式はプログラム上に書かれているはずなので, その直下に (10) 式を書き足すことは容易である.

なお, z の微分を (10) 式の表記に基づいて行なうことは標準的な方法ではなく, 多くは

$$\dot{z} = \dot{x} * y + x * \dot{y}$$

の形式で説明されている. しかしながら, 制御系の状態方程式では引数, 戻り値ともに多変数となることが多いため, (10) 式の表記を用いるほうが扱いが容易である.

2.3 複数の計算式を実行する場合

計算が複数の式の組み合わせで行なわれる場合も, 上記と同様にして書き換えを行なうことができる. たとえば, $z = xy$, $w = z^2 + x$ の計算を経て w を求める場合を考える. このとき, 対応する MATLAB 関数を

```
function w = calc_w(x,y)
    z = x*y;
```

```
    w = z^2 + x;
end
```

とする.

w は本来的には x, y の関数ではあるが, プログラム上では関数 $z(x,y)$ を用いた合成関数として定義されている. すなわち, 数学的には

$$w(x,y) = w(z(x,y), x)$$

である. このとき, x, y における w の偏微分係数は

$$J_w(x,y) = \begin{bmatrix} \frac{\partial w}{\partial z} \frac{\partial z}{\partial x} + \frac{\partial w}{\partial x} & \frac{\partial w}{\partial z} \frac{\partial z}{\partial y} \end{bmatrix} \quad (12)$$

である. (9) 式の dx, dy を用いると, (12) 式を

$$\begin{aligned} J_w(x,y) &= \left(\frac{\partial w}{\partial z} \frac{\partial z}{\partial x} + \frac{\partial w}{\partial x} \right) dx + \frac{\partial w}{\partial z} \frac{\partial z}{\partial y} dy \\ &= \frac{\partial w}{\partial z} \left(\frac{\partial z}{\partial x} dx + \frac{\partial z}{\partial y} dy \right) + \frac{\partial w}{\partial x} dx \end{aligned} \quad (13)$$

と書くこともできる. (10) 式より, さらに (13) 式を

$$J_w(x,y) = \frac{\partial w}{\partial z} dz + \frac{\partial w}{\partial x} dx \quad (14)$$

と書くことができる. 一方,

$$\frac{\partial w}{\partial z} = 2z, \quad \frac{\partial w}{\partial x} = 1$$

であるので, 結局,

$$J_w(x,y) = 2z dz + dx \quad (15)$$

となる. ここで, (15) 式右辺の z は $z = xy$ によって求めた z である. (15) 式は $w(z,x) = z^2 + x$ の全微分

$$dw = 2z dz + dx \quad (16)$$

と形式的には同一である.

以上より, w の偏微分係数を求めるためには, `calc_w(x,y)` をつぎのように書き換えればよいことがわかる.

```
function [ w, Jw ] = calc_w_J(x,y)
    dx = [ 1 0 ]; % 追加
    dy = [ 0 1 ]; % 追加
    z = x*y;
    dz = y*dx + x*dy; % 追加
    w = z^2 + x;
    dw = 2*z*dz + dx; % 追加
    Jw = dw; % 追加
end
```

すなわち, $z = xy$, $w = z^2 + x$ の各計算に対して, それぞれの直後に全微分と同じ式を追記することで w の偏微分係数を計算することができる. 上記のプログラムにおいて, dw の計算を実行する前に dz の計算が実行されていなければならないことに注意されたい.

以上の追記を行なった後に関数値の計算を変更した場合、偏微分係数の計算の変更の手間は関数値の計算と同程度である。たとえば、 z の計算後に z を 1.2 倍にするように変更した場合、偏微分係数の計算も含めた変更は下記のとおりである。

```
function [ w, Jw ] = calc_w_J(x,y)
    dx = [ 1 0 ];
    dy = [ 0 1 ];
    z = x*y;
    dz = y*dx + x*dy;
    z = 1.2*z; % 変更
    dz = 1.2*dz % 変更
    w = z^2 + x;
    dw = 2*z*dz + dx;
    Jw = dw;
end
```

上記の例のように、提案手法では関数値の計算の変更が局所的に留まる場合、偏微分係数の計算も局所的な変更ですますことができる。これにより、偏微分係数の計算を行なうにもかかわらず、コードの可読性や保守性を高く保つことが可能である。

2.4 制御構造と偏微分係数の計算

関数呼び出しがある場合にも、同様な書き換えを行なえば偏微分係数を計算することができる。たとえば、前述の `calc_w` を呼び出すことで計算を実行する以下の MATLAB 関数を考える。

```
function v = calc_v(x,y)
    u = 2 * x + y;
    v = x - y;
    w = calc_w(u,v);
    v = exp(w);
end
```

このとき、 x, y における $v(x, y)$ の偏微分係数も同時に計算することを考える。ここで、もし

$$dw = \frac{\partial w}{\partial x} dx + \frac{\partial w}{\partial y} dy \quad (17)$$

の関係を満たす dw が計算されていれば、 $v(x, y)$ の偏微分係数は

$$dv = \exp(w) dw$$

で与えられる。そこでつぎに dw の計算について考える。

関数 $w = w(u, v) = w(u(x, y), v(x, y))$ の x, y における偏微分係数 $J_w(x, y)$ は

$$\begin{aligned} J_w(x, y) &= \begin{bmatrix} \frac{\partial w}{\partial x} & \frac{\partial w}{\partial y} \end{bmatrix} \\ &= \begin{bmatrix} \frac{\partial w}{\partial u} \frac{\partial u}{\partial x} + \frac{\partial w}{\partial v} \frac{\partial v}{\partial x} & \frac{\partial w}{\partial u} \frac{\partial u}{\partial y} + \frac{\partial w}{\partial v} \frac{\partial v}{\partial y} \end{bmatrix} \\ &= \begin{bmatrix} \frac{\partial w}{\partial u} & \frac{\partial w}{\partial v} \end{bmatrix} \begin{bmatrix} \frac{\partial u}{\partial x} & \frac{\partial u}{\partial y} \\ \frac{\partial v}{\partial x} & \frac{\partial v}{\partial y} \end{bmatrix} \end{aligned} \quad (18)$$

で与えられる。よって、(17), (18) 式より、

$$\begin{aligned} dw &= \begin{bmatrix} \frac{\partial w}{\partial x} & \frac{\partial w}{\partial y} \end{bmatrix} \begin{bmatrix} dx \\ dy \end{bmatrix} \\ &= \begin{bmatrix} \frac{\partial w}{\partial u} & \frac{\partial w}{\partial v} \end{bmatrix} \begin{bmatrix} \frac{\partial u}{\partial x} dx + \frac{\partial u}{\partial y} dy \\ \frac{\partial v}{\partial x} dx + \frac{\partial v}{\partial y} dy \end{bmatrix} \\ &= J_w(u, v) \begin{bmatrix} du \\ dv \end{bmatrix} \end{aligned} \quad (19)$$

が得られる。ただし、

$$du = \frac{\partial u}{\partial x} dx + \frac{\partial u}{\partial y} dy \quad (20)$$

$$dv = \frac{\partial v}{\partial x} dx + \frac{\partial v}{\partial y} dy \quad (21)$$

であり、 $J_w(u, v)$ は u, v における $w(u, v)$ の偏微分係数である。

`calc_w_J` を実行することで得られるのは $J_w(u, v)$ である。また、 u, v の計算の直後に、それぞれの偏微分係数の計算を追記すれば、`calc_w_J` を呼び出す前に du, dv を求めておくことができる。よって、`calc_w_J` を実行後に (19) 式を計算すれば dw を計算することができる。以上のことから、`calc_v` をつぎのように書き換えればよい。

```
function [ v, Jv ] = calc_v_J(x,y)
    dx = [ 1 0 ]; % 追加
    dy = [ 0 1 ]; % 追加
    u = 2 * x + y;
    du = 2*dx + dy; % 追加
    v = x - y;
    dv = dx - dy; % 追加
    [ w, Jw ] = calc_w_J(u,v); % 追加
    dw = Jw * [ du; dv ]; % 追加
    v = exp(w);
    dv = exp(w) * dw; % 追加
    Jv = dv; % 追加
end
```

以上のような方法でプログラムを書き換えれば、関数呼び出しがある場合にも偏微分係数を計算することができる。上記からわかるように、`calc_w_J` は J_w を戻り値とすることで関数内部と関数外部で計算が分離されているので、関数呼び出しの構造を保存したままプログラムの書き換えを行なうことが可能である。実際、`calc_w` がプログラム中のさまざまな箇所から呼び出されているときにも、すべての `calc_w` の呼び出しを `calc_w_J` の呼び出しに置き換えることで偏微分係数の計算が可能である。

同様にして、繰り返しや条件分岐がある場合にもコードの書き換えを行なうことができる。たとえば、繰り返しがある以下のコードを考える。

```
x = 3;
```

```

for k = 1:3
    x = x^2 + 2*x;
end

```

このコードによる計算は本質的には以下のコードと同じである。

```

x0 = 3;
x1 = x0^2 + 2*x0;
x2 = x1^2 + 2*x1;
x3 = x2^2 + 2*x2;
x = x3;

```

これをそのまま書き換えれば以下のコードが得られる。

```

x0 = 3;
dx0 = 1; % 追加
x1 = x0^2 + 2*x0;
dx1 = (2*x0 + 2) * dx0; % 追加
x2 = x1^2 + 2*x1;
dx2 = (2*x1 + 2) * dx1; % 追加
x3 = x2^2 + 2*x2;
dx3 = (2*x2 + 2) * dx2; % 追加
x = x3;
dx = dx3; % 追加

```

これはさらに、以下のコードと等価である。

```

x = 3;
dx = 1; % 追加
for k = 1:3
    x = x^2 + 2*x;
    dx = (2*x + 2) * dx; % 追加
end

```

すなわち、繰り返しがある場合にも、もとのソースコードにおける計算式の直後に、全微分の式を追加すればよい。

また、条件分岐がある場合の書き換えは、たとえば以下のとおりである。

```

if x > 3
    x = 3*(u-3) + 2;
    dx = 3*du; % 追加
else
    x = 3*(3-u) + 2;
    dx = -3*du; % 追加
end

```

以上の考察より、以下の手順でコードを書き換えることにより、関数値の計算と同時に偏微分係数の計算を行なうコードを得ることができる。

- 変数スコープの最初に独立変数に対応する基底ベクトルを定義する。

- 各計算行の直後に、その計算の全微分に対応するコードを追加する。

- 関数呼び出しの場合には、ヤコビアンを介した偏微分係数の計算コードを追加する。

偏微分係数の計算を追記する以上の手法は、自動微分の forward mode あるいは forward accumulation, ボトムアップ型と呼ばれる方法に対応したものである。自動微分の forward mode は実際に計算を行なう順序に沿って、偏微分係数の計算を行なうものである。Fig. 1 は forward mode に従って変換した結果である。本論文の手法は、実際に計算を行なう順序に従って偏微分係数を行なうという意味では自動微分の forward mode と同じである。しかしながら、自動微分とは異なり、偏微分係数の計算を追記する粒度は行単位である。このことにより、もとの関数値の計算コードを破壊することなく偏微分係数の計算を追記することが可能である。Fig. 1 では、もとの関数値を計算するコードが分解されており、そのことによって可読性や保守性も悪化していることに注意されたい。

現在では MATLAB に対する自動微分ツール¹⁰⁾がいくつか開発されており、それらを用いれば偏微分係数を容易に求めることができる。しかしながら、それらはオペレータオーバーローディングを利用しているため、それらを Simulink の Embedded MATLAB ブロックで使用することはできない。よって、Rapid Prototyping ではコード変換に基づく自動微分を採用せざるを得ない。しかし、その場合、ソースコードの改変に伴うコード変換を行なうと、その度に保護コードの追加を行なうことになり、何度も手戻り作業が発生する。これは Rapid Prototyping の意義を大きく損ねるものである。

これに対し、本論文の書き換えを行なえば、コード作成に労力は要するが Embedded MATLAB ブロックでも実行可能なコードが得られる。また書き換えたソースコードの可読性はもとのソースコードと同程度であるため、ゼロ割対策などの例外処理を追加することも容易である。さらに、一度追加した例外処理が反古にされることもない。このように、本論文の提案内容は、Rapid Prototyping の意義を損うことなく偏微分係数計算の実装を可能にする(場当たりのでない)系統的な手法を与えるものである。

2.5 マップ補間に対する偏微分係数の計算

初等関数など、偏微分係数を陽に記述できる関数の偏微分係数は前節までに示した手法で計算可能である。一方、ディーゼルエンジンなどの複雑なモデルでは、マップ補間を用いて計算が行なわれる場合もある。そのような場合においても、補間と同時に偏微分係数を計算することが可能である。

たとえば、2 つの独立変数 x , y に対して補間点 x_i ($i = 1, \dots, N_x$), y_j ($j = 1, \dots, N_y$) が与えられ、さらに各 (x_i, y_j) における値 z_{ij} も与えられているものとする。このとき、点 $x \in [x_i, x_{i+1}]$, $y \in [y_j, y_{j+1}]$ における多重線形補間(いわゆる線形補間)の値は次式で与えられる¹³⁾。

$$f(x, y) = z_{ij}(1 - \bar{x})(1 - \bar{y}) + z_{(i+1)j}\bar{x}(1 - \bar{y}) \\ + z_{i(j+1)}(1 - \bar{x})\bar{y} + z_{(i+1)(j+1)}\bar{x}\bar{y} \quad (22)$$

ただし,

$$\bar{x} = \frac{x - x_i}{x_{i+1} - x_i}, \quad \bar{y} = \frac{y - y_j}{y_{j+1} - y_j}$$

である.

以上より, (22) 式の $f(x, y)$ に対する偏微分係数は次式で与えられる.

$$\frac{\partial f}{\partial x}(x, y) = z_{ij}^x(1 - \bar{y}) + z_{(i+1)j}^x\bar{y} \quad (23)$$

$$\frac{\partial f}{\partial y}(x, y) = z_{ij}^y(1 - \bar{x}) + z_{i(j+1)}^y\bar{x} \quad (24)$$

ただし,

$$z_{ij}^x = \frac{z_{(i+1)j} - z_{ij}}{x_{i+1} - x_i}, \quad z_{ij}^y = \frac{z_{i(j+1)} - z_{ij}}{y_{j+1} - y_j}$$

である.

(23), (24) 式の計算を行なえば (22) 式の偏微分係数を得ることができる. よって, (22)~(24) 式の計算を行なう関数を作成すれば, マップ補間に対しても関数値と同時に偏微分係数を計算することが可能である. その際, 偏微分係数は (23), (24) 式に基づいて計算しても構わないし, 提案手法に沿って, (22) 式を計算する各コードに対して全微分の式を追加することで行なっても構わない.

ここでは2変数に対する多重線形補間の場合を示したが, 3変数以上の多重線形補間¹⁴⁾やスプライン補間¹⁵⁾などの場合に対しても同様にして計算可能である.

3. ディーゼルエンジン吸排気モデルへの適用

3.1 偏微分係数計算の検証

前章で述べた書き換えをディーゼルエンジン吸排気モデルに適用した. 吸排気モデルは文献 16)~22) に準拠したものである. これを (1), (2) 式で記述すると, $n = 10$, $m_1 = 4$, $m_2 = 3$, $p = 2$ である. 外生入力, 制御入力, 出力の内訳は以下のとおりである.

外生入力 燃料噴射量, エンジン回転数, 大気温度, 大気圧

制御入力 スロットル, VGT ベーン, EGR 弁

制御出力 EGR 率, 過給圧

関数 f , g の値を計算するコード (`engine_core.value.m`) を Embedded Matlab Language (EML)¹¹⁾ で記述したところ, コードの行数は 916 行であった. ただし, 本コードでは初等関数のほか, 条件分岐や繰り返し, 関数呼び出しは用いられているが, マップによる補間は用いられていない.

つぎに, 2章の手法でコードを書き替えて `engine_core.df.m` を作成した. `engine_core.df.m` の行数は 2440 行となった. ただし, `engine_core.df.m` には `engine_core.value.m` にはなかった例外処理なども含まれている.

実際に書き換えを行なった例を Fig. 2 に示す. これは, シリンダ流入流量を計算する部分である. この関数の引数のうち, システムの状態や入力に関連するのは第 1 引数 (`T_out_intake_man`), 第 2 引数 (`p_out_intake_man`) のみであるので, 偏微分係数を計算するための基底もそれらに対応したもののみが定義されている (5, 6 行目). 偏微分係数の計算を行なうように追記した部分は 15 行目と 20~24 行目の部分である. また, 計算した J を戻り値で返すようにした

```

1 function [W_in_cyl, J] = ...
2   calc_cylinder_flow(T_out_intake_man, p_out_intake_man, ...
3     engine_speed, vol_eff_param)
4
5   d_T_out_intake_man = [ 1 0 ];
6   d_p_out_intake_man = [ 0 1 ];
7
8   kPa_to_Pa = 1000;
9   R_air = 287.0;
10  displacement = ***
11
12  x = vol_eff_param;
13
14  vol_eff = ***
15  d_vol_eff = ***
16
17  W_in_cyl = vol_eff*p_out_intake_man*engine_speed ...
18    * displacement/(120*R_air*T_out_intake_man);
19
20  J = engine_speed*displacement/(120*R_air) * ...
21    (p_out_intake_man/T_out_intake_man*d_vol_eff ...
22    + vol_eff/T_out_intake_man*d_p_out_intake_man ...
23    - vol_eff*p_out_intake_man/(T_out_intake_man^2)* ...
24    d_T_out_intake_man);
25
26 end

```

Fig. 2 A part of engine_core.df.m

Table 1 Computation time

		min. (μ s)	mean (μ s)	max. (μ s)
Matlab	engine_core_value.m	348.9	376.7	1863
	$\times 14$	4884	5273	26082
	engine_core_df.m	745.0	806.6	5244
Simulink	engine_core_value.m	—	15.16	—
	$\times 14$	—	212.2	—
	engine_core_df.m	—	38.59	—

(1 行目). Fig. 2 より, ソースコードの可読性はもとの関数とほぼ同等であることが確認できる. 10, 14, 15 行目の右辺は明示されていないが, 実際には, ある式が記述されている. Fig. 2 のコード中には T.out_intake_man による除算が含まれている. 通常, 除算を行なう際にはゼロ割回避のコードが必要である. しかし, この場合には T.out_intake_man がゼロにならないような処理が関数を呼び出す前に行なわれているので, Fig. 2 にはそのコードは含まれていない.

engine_core_value.m と engine_core_df.m の計算時間の比較を Table 1 に示す. 時間の単位は μ s である. 計算を実行した環境は以下のとおりである.

- CPU : Core-i5-3340M, 2.7 GHz
- Memory : 8 GB
- OS : Windows 8.1 Pro 64 bit
- Matlab R2010bSP1 (32 bit)

Table 1 の 1, 3 行目に示されている計算時間は, それぞれの関数呼び出しを Matlab で 10000 回実行したときの結果である. 計算時間の計測には tic/toc コマンドを使用した. 計測結果から, tic/toc コマンドによる計測の分解能は約 0.4μ s である. また, 同一の計算を繰り返しているにもかかわらず計測結果がゆらいているため, Table 1 には計測結果の最小値, 平均値, 最大値を示してある. 計測結果のゆらぎはプロセスのコンテキストスイッチによるものと考えられるので, 最小値が関数の本来の計算時間に最も近い値であると考えられる.

一方, Table 1 の 4, 6 行目に示されている計算時間は, それぞれの関数呼び出しを Simulink で 10000 回実行したときの結果である. Simulink の計算はアクセラレータモードで実行した. ソルバーオプションには

- 種類 : 固定ステップ
- ソルバー : 離散 (連続状態なし)

を設定し, コンパイラの最適化レベルには「最適化オン (高速に実行)」を設定した. 関数の呼び出しには Embedded MATLAB Function ブロック, 計算時間の計測にはプロファイラーをそれぞれ用いている. プロファイラーは関数呼び出しの総時間を計測するため, 得られる計算時間は平均値のみである. Table 1 より, Matlab, Simulink いずれの場合においても engine_core_df.m の計算時間は engine_core_value.m のおおよそ 2 倍程度であることが確認できる.

一方, 偏微分係数 $\frac{\partial f_i}{\partial x_j}$ ($j = 1, \dots, n$), $\frac{\partial f_i}{\partial u_j}$ ($j = 1, \dots, m_2$)

を前進差分に基づく数値微分で求める場合には, 与えられた微小量 $h_j > 0$ ($j = 1, \dots, n$), $\ell_j > 0$ ($j = 1, \dots, m_2$) に対して, それぞれ

$$\frac{f_i(x + h_j e_j, w, u) - f_i(x, w, u)}{h_j} \quad (25)$$

$$\frac{f_i(x, w, u + \ell_j b_j) - f_i(x, w, u)}{\ell_j} \quad (26)$$

を計算する. ただし, e_j, b_j はそれぞれ n 次元空間, m_2 次元空間における第 j 番目の自然基底ベクトルである. (25), (26) 式を計算するためには, $f(x, w, u), f(x + h_j e_j, w, u), f(x, w, u + \ell_j b_j)$ を計算する, すなわち, engine_core_value.m を $1 + n + m_2 = 14$ 回実行する必要がある. よって, 数値微分の計算時間は engine_core_value.m の 14 倍になる. Table 1 の 2, 5 行目は, それぞれ Matlab, Simulink での engine_core_df.m の計算時間を 14 倍した時間を示している. この比較から, 自動微分が偏微分係数を効率的に計算できることがわかる.

つぎに, 計算結果の精度についての検証を行なう. ここで用いている吸排気系モデルのプログラムは複雑であるため, 偏微分係数の真値を知ることは困難である. そこで, 数値微分による計算結果と自動微分による計算結果との差を計算することによって, 計算結果の精度を評価する.

f の引数 x, w, u が与えられたものとする. このとき, f の第 i 成分 f_i の x_j に関する偏微分係数 $\frac{\partial f_i}{\partial x_j}$ の数値微分に基づく計算を (25) 式に基づいて行なう. 具体的には, $h_j = 10^{-N} x_j$ とした

$$\nu_{ij}(N) = \frac{f_i(x + 10^{-N} x_j e_j, w, u) - f_i(x, w, u)}{10^{-N} x_j}$$

を用いる. ただし, N は数値微分における微小変化を与えるための自然数である. 一方, $\frac{\partial f_i}{\partial x_j}$ を自動微分で求めた値を α_{ij} とする. 以上の 2 つの偏微分係数に対し, α_{ij} に対する $\nu_{ij}(N)$ の相対誤差

$$\varepsilon_{ij}(N) = \frac{|\nu_{ij}(N) - \alpha_{ij}|}{|\alpha_{ij}|}$$

を各 N に対して計算する.

$N = 1, \dots, 6$ に対して $\varepsilon_{11}, \varepsilon_{63}$ を求めた結果を, それぞれ Fig. 3, Fig. 4 に示す. いずれの図においても, N が大きくなると相対誤差が小さくなっている. このことは, 数値微分の精度を高めると, 偏微分係数の計算値が自動微分の計算値に近づくことを示している. すなわち, 自動微分は高精度に偏微分係数を計算できていることがわかる. 数値微分の場合には, Fig. 3, Fig. 4 のように微小変化量の与え方によって結果が変化し, 十分な精度を与える N は事前には不明である. 一方, 自動微分ではパラメータ探索の必要はない.

3.2 線形近似システムの検証

engine_core_df.m を用いれば, オンラインで Just In Time に線形近似システムを計算することができる. これを検証するために, 線形近似システムに基づくシミュレーションを考える. (1) 式の状態方程式において, 動作点 x_* が一つ定めれば (3) 式の線形近似システムを求めることができる. ここで,

Fig. 3 Relative error of $\frac{\partial f_1}{\partial x_1}$

Fig. 4 Relative error of $\frac{\partial f_6}{\partial x_3}$

動作点が運転中に変化する場合には線形近似システムも運転中に変化する。一方、`engine_core_df.m` は各サンプル時刻で線形近似システムを求めるので、運転中に動作点が変わっても、求まる線形近似システムは各時刻の動作点に対応したものとなる。

上記の方法の有効性を検証するために、`engine_core_df.m` に基づくシミュレーションを行なう。ディーゼルエンジン吸排気モデルは連続時間部分と離散時間部分の両方から構成されている。そこで、連続時間部分はオイラー前進差分を用いて離散化を行なった。離散化のサンプル周期は 1 ms である。

応答を求める際に与えた外生入力と動作点は以下のとおりである。外生入力であるエンジン回転数、大気圧、大気温はそれぞれ 3000 RPM, 298.15 K, 101.4 KPa とし、燃料噴射量は Fig. 5 のように与えた。通常、エンジン制御を行なう場合、外生入力に対する動作点を事前にテーブルで定めておくことが多いが、ここでも同じ手法で動作点を与える。与えられた外生入力に対する制御入力動作点は Fig. 6 のとおりである。ただし、Fig. 6 は各入力ごとに 10 秒間の最大値で正規化したものである。また、スロットル、VGT ペーン、EGR 弁それぞれに対して、0 が全開、全開、全閉に対応する。状態の動作点もあらかじめオフラインで求めておいたものを使用した。

Fig. 5 Response of fuel

Fig. 6 Responses of operating control inputs

Fig. 7 Response of pressure in intake manifold

Fig. 7 は上記の外生入力と Fig. 6 の制御入力に対する過給圧の応答を求めた結果である。ただし、過給圧は時刻 0 における値が 1 となるように正規化したものである。Fig. 7 の点線の応答は非線形状態方程式を用いたときの応答、実線の応答は `engine_core_df.m` によって各サンプル時刻で得られる線形近似システムに基づいて応答を計算した結果である。Fig. 7 の点線と実線がほぼ同じ値を示していることから、この対象に対して拡張カルマンフィルタなど、線形近似に基づく手

法が有効に機能することが期待できる。実際に本コードに基づいて拡張カルマンフィルタや最適制御を設計できること、さらにはそれらに基づいて制御実験を行なった結果も示されている²³⁾。

4. おわりに

非線形システムの制御の際に用いられる線形近似システムを自動微分に基づいて計算するために、自動微分を実現する見通しのよいコード書き換え手法を提案した。また、自動微分をディーゼルエンジン吸排気モデルに適用し、偏微分係数を高精度かつ効率的に計算できることを示した。本論文では線形近似システムを動作点近傍で求める例を示した。拡張カルマンフィルタでは動作点ではない点で評価した偏微分を用いるが、その目的にも書き換えたコードをそのまま用いることができる。

参 考 文 献

- 1) 足立, 丸山: カルマンフィルタの基礎, 東京電機大学出版局 (2012)
- 2) 久保田, 伊理: アルゴリズムの自動微分と応用, コロナ社 (1998)
- 3) M. Bartholomew-Biggs, S. Brown, B. Christianson and L. Dixon: Automatic Differentiation of Algorithms, J. Comput. Appl. Math., **124**-1-2, 171/190 (2000)
- 4) R. Griesse and A. Walter: Evaluating Gradients in Optimal Control: Continuous Adjoint Versus Automatic Differentiation, J. Optimiz. Theor. Applicat., **122**-1, 63/86 (2004)
- 5) S.L. Campbell, E. Moore and Y. Zhong: Utilization of Automatic Differentiation in Control Algorithms, IEEE Trans. Autom. Contr., **39**-5, 1047/1051 (1994)
- 6) K. Röbenack and K.J. Reinschke: Nonlinear Observer Design Using Automatic Differentiation, *Automatic Differentiation of Algorithms: From Simulation to Optimization*, 137/142, Springer-Verlag (2002)
- 7) Y. Cao and R. Al-Seyab: Nonlinear Model Predictive Control Using Automatic Differentiation, 2003 ECC, 2008/2013 (2003)
- 8) 井前 譲: 微分遺伝的プログラミングによる非線形制御系設計, 計測と制御, **42**-10, 804/808 (2003)
- 9) 高橋, 川原: 感度解析に自動微分を用いた非圧縮粘性流体における逆解析手法, 応用力学論文集, **7**, 305/312 (2004)
- 10) autodiff.org, <http://www.autodiff.org/>
- 11) *Embedded MATLAB Users' Guide*, The MathWorks (2007)
- 12) TAPENADE, <http://tapenade.inria.fr:8080/tapenade/index.jsp>
- 13) G. Dahlquist and Å. Björck: *Numerical Methods in Scientific Computing*, Volume 1, SIAM (2008)
- 14) A. Weiser and S.E. Zarantonello: A Note on Piecewise Linear and Multilinear Table Interpolation in Many Dimensions, Math. Comput., **50**-181, 189/196 (1988)
- 15) 桜井 明(編): スプライン関数入門, 東京電機大学出版局 (1971)
- 16) I. Kolmanovsky, P. Moraal, M. van Nieuwstadt and A. Stefanopoulou: Issues in modeling and control of intake flow in variable geometry turbocharged engines, Chapman and Hall CRC research notes in mathematics, 436/445 (1999)
- 17) J.B. Heywood: *Internal Combustion Engine Fundamentals*, McGraw-Hill Book Company (1998)
- 18) J. Wahlstrom and L. Eriksson: Modeling Diesel Engines

with a Variable-Geometry Turbocharger and Exhaust Gas Recirculation by Optimization of Model Parameters for Capturing Nonlinear System Dynamics, J. Automob. Eng., **225**-7, 960/986 (2011)

- 19) J. Faust, J. Sun, K. Butts, Z. Lu and S. Tanaka: Parameterization and Adaptation of Gasoline Engine Air System Model Via Linear Programming Support Vector Regression, American Control Conference 2012, 1290/1295 (2012)
- 20) Z. Lu, J. Sun and K. Butts: Linear Programming Support Vector Regression with Wavelet Kernel: A New Approach to Nonlinear Dynamical System Identification, Math. Comput. Simulat., **79**-7, 2051/2063 (2009)
- 21) P. Moraal and I. Kolmanovsky: Turbocharger Modeling for Automotive Control Applications, SAE 1999-01-0908 (1999)
- 22) L. Eriksson: Modeling and Control of Turbocharged SI and DI Engines, Oil Gas Sci. Technol., **62**-4, 523/538 (2007)
- 23) 浅井, Pham, 仲田, 松永: 拡張カルマンフィルタと最適制御に基づくディーゼルエンジン吸排気制御, 第 3 回計測自動制御学会制御部門マルチシンポジウム, 1E1-3 (2016)

[著 者 紹 介]

浅 井 徹 (正会員)

1996 年東京工業大学大学院理工学研究科制御工学専攻修了。同年日本学術振興会特別研究員。99 年大阪大学大学院工学研究科助手, 2002 年同研究科講師, 2005 年同研究科助教授, 2007 年同研究科准教授, 2015 年名古屋大学大学院工学研究科准教授となり, 現在に至る。IEEE, システム制御情報学会, 鉄鋼協会の会員。博士 (工学)。

仲 田 勇 人 (正会員)

2005 年京都大学大学院情報学研究科数理工学専攻博士後期課程修了。同年トヨタ自動車 (株) に入社, 現在に至る。モデルベース制御手法に基づくエンジン制御の研究・開発に従事。2013, 2015 年計測自動制御学会制御部門大会技術賞, 2014 年度計測自動制御学会論文賞受賞。博士 (情報学)。

松 永 彰 生 (正会員)

1993 年東京大学大学院工学系研究科機械工学専攻専攻修士課程修了。同年トヨタ自動車 (株) に入社, 現在に至る。ディーゼルエンジン電子制御システムの研究・開発に従事。