プログラムツール **Eigen**

— **3** 次元コンピュータビジョン計算ハンドブック付録 —

金澤 靖

Version 1.0

2016 年 **10** 月

まえがき

本書は，「金谷，菅谷，金澤，3 次元コンピュータビジョン計算ハンドブック，森北出版，2016」[1] の付録として作成したものである．当初は本に含める予定であったが，ページ数の問題から Web 上の文書とした．

3 次元コンピュータビジョン計算ハンドブックでは様々なアルゴリズムが詳細に説明されており，そのままプログラミングすることもできる学生も多いはずである．しかし，その一方でベクトルや行列の演算が多用されているため，そのがネックとなり，難しいと感じてしまう学生も多い．そこで，そのような学生のための橋渡しになればと思い，本書を用意することとした．

本書で紹介する Eigen はベクトルや行列の計算を行うための C++ 言語用のライブラリであり，無償で利用できる．この Eigen はサンプルプログラムで示しているように，C++ であることを意識せず，C 言語のように使うことも可能であり，C++ を知らない人も容易に使い始められる．

本書は C 言語や C++ 言語のプログラミング経験のある人を対象にしている．C 言語や C++ 言語のプログラミングについては，各種の入門書および解説本 [2, 3, 4, 5] や自分のレベルに合った入門書を参照して欲しい．

2016 年 10 月

金澤 靖

目次

1 はじめに

1.1 Eigen とは

Eigen は，ヘッダファイルのみで構成されるフリーなテンプレートライブラリである．したがって，インストールするとき，コンパイルが必要なく，ほとんどの C++ コンパイラで利用できる．記述は直観的で，数式に近い記述が可能である．Eigen には次の特徴がある．

- 汎用性がある．
- 高速である．
- 信頼性がある．
- 記述がエレガントである．
- ほとんどのコンパイラで利用できる．

ライセンスは MPL2 であるため，ユーザが開発した部分の公開の義務はなく，商用アプリケーションにも利用できる．欠点としては，Eigen はテンプレートライブラリであるため，次のことがある．

- コンパイル時のエラーの個所がわかりにくい．
- コンパイルに比較的時間がかかる．
- コードがやや大きくなる．

1.2 Eigen のインストール

Eigen のインストールは容易で，linux の場合は次のように行う．

1. Eigen の Web ページ (`http://eigen.tuxfamily.org/`) から，アーカイブされたファイル（`eigen-eigen-10219c95fe65.tar.bz2` など，“10219c95fe65” の部分はバージョンによって変わる）をダウンロードする．
2. 適当な場所で展開する．
   ```
   $ tar jxvf eigen-eigen-10219c95fe65.tar.bz2
   ```
3. ヘッダファイルを適切な場所 (`/usr/local/include/eigen3` など) にコピーする．
   ```
   $ sudo cp eigen-eigen-10219c95fe65/Eigen /usr/local/include/eigen3
   ```

コンパイルも，ヘッダファイルをコピーした場所を指定すればよい．たとえば，linux で `/usr/local/include/eigen3` にコピーしてあるとき，ソース `ex.cc` をコンパイルするには次のようにする．

```
$ g++ -I/usr/local/include/eigen3 ex.cc
```

以下，本書で必要とするベクトルや行列の演算について簡単に説明する．

2　ベクトルとその演算

2.1　簡単なプログラム

簡単なプログラムとして，3 次元ベクトルの各要素に値を設定し，それを表示するプログラムを示す．

サンプルプログラム 1　ベクトルの簡単なプログラム

```
1  #include <iostream>
2  #include <Eigen/Core> // 基本演算
3
4  using namespace Eigen;
5  using namespace std;
6
7  int main()
8  {
9     Vector3d a; // 3 次元ベクトル, 要素は double
10
11    a << 1.0, 0.7, -0.52; // ベクトルの要素を設定
12    cout << "vector␣a␣=␣\n" << a << endl; // そのまま出力
13    cout << "vector␣a␣=␣" << a.transpose() << endl; // 転置して出力
14
15    cout << "a(0)␣=␣" << a(0) << endl; // 0 番目の要素
16    cout << "a(1)␣=␣" << a(1) << endl; // 1 番目の要素
17    cout << "a(2)␣=␣" << a(2) << endl; // 2 番目の要素
18 }
```

このプログラムでは，以下のことを行っている．

- 2 行目で Eigen の基本的な行列演算のヘッダファイルを指定している．
- 4 行目で名前空間 Eigen を指定している．この名前空間を指定しない場合は，スコープ解決演算子 (::) を使って，以下で `Eigen::Vector3d` のように指定する必要がある．
- 9 行目では `double` 型の 3 次元ベクトル変数 `a` を定義し，11 行目でその各要素の値を設定している．このように，カンマ演算子 (,) と左シフト演算子 (<<) を組み合わせて簡単に変数を初期化できる．
- 12 行目のように，標準出力に出力することができる．ただし標準では，ベクトルは行ベクトルとして扱われるが，13 行目のように転置して出力すると，結果が見やすくなる．
- 15〜17 行目は，各要素を直接に指定して出力している．このように括弧演算子 (()) を使って要素に直接アクセスできる．このとき，デフォルトでは要素の添字が有効化どうかチェックをしてくれる．

サンプルプログラム 1 の出力結果は，次のようになる．

```
1  vector a =
2     1
```

```
3    0.7
4  -0.52
5  vector a = 1 0.7 -0.52
6  a(0) = 1
7  a(1) = 0.7
8  a(2) = -0.52
```

2.2 ベクトルの初期化

サンプルプログラム 2 に，ベクトルのさまざまな初期化方法を示す．

- 9 行目はコンストラクタによる初期化．
- 12 行目はカンマ演算子とシフト演算子による初期化．
- 13 行目は直接に要素を指定する初期化．
- 19 行目は単位ベクトルへの初期化．
- 20 行目は零ベクトルへの初期化．
- 21 行目はすべての要素が 1 のベクトルへの初期化．
- 22 行目は X 軸方向の単位ベクトルへの初期化．
- 23 行目は Y 軸方向の単位ベクトルへの初期化．
- 24 行目は Z 軸方向の単位ベクトルへの初期化．
- 25 行目はランダムな値での初期化．
- 26 行目はすべて同じ値による初期化．

このように，用途に応じたさまざまな初期化方法が用意されている．

サンプルプログラム 2　ベクトルのさまざまな初期化

```
1  #include <iostream>
2  #include <Eigen/Core> // 基本演算
3
4  using namespace Eigen;
5  using namespace std;
6
7  int main()
8  {
9     Vector3d a(0.5, 1.0, -2.4); // コンストラクタによる初期化
10    Vector3d v1, v2, v3, v4, v5, v6, v7, v8;
11
12    v1 << 1.0, 0.7, -0.52; // カンマ演算子とシフト演算子による初期化
13    v2(0) = 3.0; v2(1) = 1.0; v2(2) = -2.0; // 要素を直接に指定する代入
14
15    cout << "a␣=␣" << a.transpose() << endl;
16    cout << "v1␣=␣" << v1.transpose() << endl;
17    cout << "v2␣=␣" << v2.transpose() << endl;
18
19    v1 = Vector3d::Identity(); // 単位ベクトル (1,0,0)
```

```
20     v2 = Vector3d::Zero(); // 零ベクトル (0,0,0)
21     v3 = Vector3d::Ones(); // (1,1,1)
22     v4 = Vector3d::UnitX(); // (1,0,0)
23     v5 = Vector3d::UnitY(); // (0,1,0)
24     v6 = Vector3d::UnitZ(); // (0,0,1)
25     v7 = Vector3d::Random(); // ランダムな値のベクトル
26     v8 = Vector3d::Constant(0.1); // すべて同じ値のベクトル (0.1,0.1,0.1)
27
28     cout << "v1␣=␣" << v1.transpose() << endl;
29     cout << "v2␣=␣" << v2.transpose() << endl;
30     cout << "v3␣=␣" << v3.transpose() << endl;
31     cout << "v4␣=␣" << v4.transpose() << endl;
32     cout << "v5␣=␣" << v5.transpose() << endl;
33     cout << "v6␣=␣" << v6.transpose() << endl;
34     cout << "v7␣=␣" << v7.transpose() << endl;
35     cout << "v8␣=␣" << v8.transpose() << endl;
36 }
```

2.3 ベクトルの次元と型

ベクトルの次元や要素の型は，プログラム 3 のように宣言できる．命名規則は，

- `Vector[次元][型]`

である．「次元」には `2`, `3`, `4`, `X` があり，それぞれ 2 次元，3 次元，4 次元，動的次元（可変長次元）を表す．「型」には `d`, `f`, `i` があり，それぞれ `double` 型，`float` 型，`int` 型を表す[*1]．

サンプルプログラム 3　ベクトルの次元と型

```
1  #include <iostream>
2  #include <Eigen/Core> // 基本演算
3
4  using namespace Eigen;
5  using namespace std;
6
7  int main()
8  {
9     // よく使われる 2 次元，3 次元，4 次元ベクトルは用意されている
10    Vector2d a; // double 型の 2 次元ベクトル
11    Vector2f b; // float 型の 2 次元ベクトル
12    Vector2i c; // int 型の 2 次元ベクトル
13
14    Vector3d d; // double 型の 3 次元ベクトル
15    Vector3f e; // float 型の 3 次元ベクトル
16    Vector3i f; // int 型の 3 次元ベクトル
17
18    Vector4d g; // double 型の 4 次元ベクトル
19    Vector4f h; // float 型の 4 次元ベクトル
```

[*1] 複素数型も用意されているが，本書では使わないので省略．

```
20    Vector4i p; // int 型の 4 次元ベクトル
21
22    // 可変長次元のベクトルも利用できる
23    VectorXd q; // double 型の可変長次元ベクトル
24    VectorXf r; // float 型の可変長次元ベクトル
25    VectorXi s; // double 型の可変長次元ベクトル
26 }
```

一般の固定長次元のベクトルは，次のプログラム 4 の 7〜10 行目のように typedef と組み合わせて，用意されている型と同様に扱える[*2]．

サンプルプログラム 4　一般の固定長ベクトル

```
1  #include <iostream>
2  #include <Eigen/Core> // 基本演算
3
4  using namespace Eigen;
5  using namespace std;
6
7  typedef Matrix<double,6,1> Vector6d; // 6 次元 (固定長) ベクトル (double)
8  typedef Matrix<double,9,1> Vector9d; // 9 次元 (固定長) ベクトル (double)
9  typedef Matrix<int,6,1> Vector6i; // 6 次元 (固定長) ベクトル (int)
10 typedef Matrix<float,9,1> Vector9f; // 9 次元 (固定長) ベクトル (float)
11
12 int main()
13 {
14    // 上記のベクトルの利用
15    Vector6d a; // double 型の 6 次元ベクトル
16    Vector9f b; // float 型の 9 次元ベクトル
17 }
```

Eigen では，コンパイル時にサイズ (=次元) が決まっているものを「固定長サイズ」(fixed size)，コンパイル時にはサイズがわからないものを「動的サイズ」(dynamic size) とよぶ．おおよそ 16 次元以下であれば，固定長サイズのほうが効率がよい．

2.4　ベクトルの基本演算

サンプルプログラム 5 に，Eigen で利用できるベクトルの基本演算を示す．

- 19 行目はベクトルの代入（コピー）．
- 20 行目は単項のマイナス．
- 26 行目は多重代入．
- 27〜29 行目はベクトルのスカラー倍．
- 40, 41 行目はベクトルの和および差．

[*2] このように，Eigen ではベクトルは N 行 1 列の行列として実装されている．

- 30, 31 行目，42, 43 行目のように，C 言語で用いられる代入演算子も利用可能．
- 50 行目はベクトルの内積．
- 51 行目はベクトル積．これは幾何学演算の一つであるため，3 行目の Geometry のヘッダファイルをインクルードする必要がある．
- 57 行目はベクトルのノルム．これを利用すれば単位ベクトルへの正規化が行えるが，59 行目のように直接正規化するメンバー関数も用意されている．また，2 乗ノルムも 58 行目のように計算できる．

このように，ベクトルの内積やベクトル積はクラスのメンバー関数として実装されている．通常の演算は演算子の多重定義を利用しているため，数式のような直観的な記述が可能である．

サンプルプログラム 5　ベクトルの基本演算

```
1  #include <iostream>
2  #include <Eigen/Core> // 基本演算
3  #include <Eigen/Geometry> // ベクトル積に必要
4
5  using namespace Eigen;
6  using namespace std;
7
8  int main()
9  {
10   Vector3d a, ac, am, sa, as, ad, as2, ad2, a2, a3, b, c, d, vp, n;
11   double ip, s, nrm, sqrnrm;
12
13   a = Vector3d::Random(); // ランダムな要素のベクトルを生成
14   b = Vector3d::Random();
15
16   cout << "a␣=␣" << a.transpose() << endl;
17   cout << "b␣=␣" << b.transpose() << endl;
18
19   ac = a; // ベクトルの値のコピー
20   am = -a; // 単項のマイナス
21
22   cout << "ac␣=␣" << ac.transpose() << endl;
23   cout << "am␣=␣" << am.transpose() << endl;
24
25   s = 3.0;
26   as2 = ad2 = a; // 多重代入
27   sa = s * a; // スカラー倍
28   as = a * s; // スカラー倍
29   ad = a / s; // スカラー除算
30   as2 *= s; // 代入演算子によるスカラー倍
31   ad2 /= s; // 代入演算子によるスカラー除算
32
33   cout << "sa␣=␣" << sa.transpose() << endl;
34   cout << "as␣=␣" << as.transpose() << endl;
35   cout << "ad␣=␣" << ad.transpose() << endl;
36   cout << "as2␣=␣" << as2.transpose() << endl;
37   cout << "ad2␣=␣" << ad2.transpose() << endl;
```

```
38
39     a2 = a3 = a;
40     c = a + b; // ベクトルの和
41     d = a - b; // ベクトルの差
42     a2 += b; // 代入演算子による和
43     a3 -= b; // 代入演算子による差
44
45     cout << "a␣+␣b␣=␣" << c.transpose() << endl;
46     cout << "a␣-␣b␣=␣" << d.transpose() << endl;
47     cout << "a2␣=␣" << a2.transpose() << endl;
48     cout << "a3␣=␣" << a3.transpose() << endl;
49
50     ip = a.dot(b); // ベクトルの内積
51     vp = a.cross(b); // ベクトルのベクトル積
52
53     cout << "dot(a,b)␣=␣" << ip << endl;
54     cout << "cross(a,b)␣=␣" << vp.transpose() << endl;
55
56     n = a;
57     nrm = a.norm(); // ベクトルのノルム (大きさ)
58     sqrnrm = a.squaredNorm(); // ベクトルの 2 乗ノルム
59     n.normalize(); // 単位ベクトルへの正規化
60
61     cout << "norm␣of␣a␣=␣" << nrm << endl;
62     cout << "squared␣norm␣of␣a␣=␣" << sqrnrm << endl;
63     cout << "normalized␣a␣=␣" << n << endl;
64 }
```

異なる次元のベクトルの演算を行おうとすると，次のプログラム 6 のように，コンパイル時にエラーとなる．

サンプルプログラム 6　異なる次元のベクトル演算

```
1  #include <iostream>
2  #include <Eigen/Core> // 基本演算
3  #include <Eigen/Geometry> // ベクトル積に必要
4
5  using namespace Eigen;
6  using namespace std;
7
8  int main()
9  {
10     Vector3d a, b, c;
11     Vector4d d, e, f;
12     double s3, s4;
13
14     a = Vector3d::Random(); // ランダムな要素の 3 次元ベクトルを生成
15     b = Vector3d::Random();
16
17     d = Vector4d::Random(); // ランダムな要素の 4 次元ベクトルを生成
18     e = Vector4d::Random();
19
```

```
20     cout << "a␣=␣" << a.transpose() << endl;
21     cout << "b␣=␣" << b.transpose() << endl;
22
23     cout << "d␣=␣" << d.transpose() << endl;
24     cout << "e␣=␣" << e.transpose() << endl;
25
26     // 同じ次元のベクトルの演算は可能
27     c = a + b;
28     f = d + e;
29     s3 = a.dot(b);
30     s4 = d.dot(e);
31
32     cout << "a␣+␣b␣=␣" << c.transpose() << endl;
33     cout << "d␣+␣e␣=␣" << f.transpose() << endl;
34     cout << "dot(a,b)␣=␣" << s3 << endl;
35     cout << "dot(d,e)␣=␣" << s4 << endl;
36
37     // 以下の演算は, 次元が違うのでコンパイル時にエラー
38     // c = a + d;
39     // f = d + b;
40     // s3 = a.dot(d);
41 }
```

3　行列とその演算

3.1　簡単なプログラム

プログラム 7 に示すように，Eigen では行列も同じように直観的に扱える．カンマ演算子を用いた初期化も，ベクトルと同様に利用できる．

サンプルプログラム 7　簡単なプログラム

```
1  #include <iostream>
2  #include <Eigen/Core> // 基本演算のみ
3
4  using namespace Eigen;
5  using namespace std;
6
7  int main()
8  {
9     Matrix3d M; // 3 × 3 行列, 要素は double
10
11    // 行列に値を代入
12    M << 1.1, 2.2, 3.3,
13         4.4, 5.5, 6.6,
14         7.7, 8.8, 9.9;
15    cout << "matrix␣M␣=␣\n" << M << endl; // そのまま出力
16
17    cout << "M(0,0)␣=␣" << M(0,0) << endl; // (0,0) 要素
18    cout << "M(1,2)␣=␣" << M(1,2) << endl; // (1,2) 要素
```

```
19     cout << "M(2,2)␣=␣" << M(2,2) << endl; // (2,2) 要素
20 }
```

このプログラムの出力は次のようになる．

```
1  matrix M =
2  1.1 2.2 3.3
3  4.4 5.5 6.6
4  7.7 8.8 9.9
5  M(0,0) = 1.1
6  M(1,2) = 6.6
7  M(2,2) = 9.9
```

3.2 行列の初期化

ベクトルと同じように，行列に関してもさまざまな初期化方法が用意されている．ただし，コンストラクタによる初期化は行えない．

サンプルプログラム 8 行列のさまざまな初期化

```
1  #include <iostream>
2  #include <Eigen/Core> // 基本演算のみ
3
4  using namespace Eigen;
5  using namespace std;
6
7  int main()
8  {
9     Matrix3d M1, M2, M3, M4, M5, M6, M7;
10
11    // コンストラクタによる初期化はエラー
12    // Matrix3d M(0.5, 1.0, -2.4, 0.0, 1.1, 0.2, -0.6, 2.1, 3.4);
13
14    // カンマ演算子とシフト演算子による初期化
15    M1 << 1.0, 0.7, -0.52, 0.0, 2.0, -1.4, 3.9, -5.2, 1.22;
16
17    // 要素を直接指定して代入
18    M2(0,0) = 1.23; M2(0,1) = 2.34; M2(0,2) = 3.45;
19    M2(1,0) = 4.56; M2(1,1) = 5.67; M2(1,2) = 6.78;
20    M2(2,0) = 7.89; M2(2,1) = 8.91; M2(2,2) = 9.12;
21
22    M3 = Matrix3d::Identity(); // 単位行列
23    M4 = Matrix3d::Zero(); // 零行列
24    M5 = Matrix3d::Ones(); // すべての要素が 1 の行列
25    M6 = Matrix3d::Random(); // ランダムな値の行列
26    M7 = Matrix3d::Constant(0.1); // すべての要素が同じの行列
27
28    cout << "M1␣=␣\n" << M1 << endl;
29    cout << "M2␣=␣\n" << M2 << endl;
```

```
30     cout << "M3␣=␣\n" << M3 << endl;
31     cout << "M4␣=␣\n" << M4 << endl;
32     cout << "M5␣=␣\n" << M5 << endl;
33     cout << "M6␣=␣\n" << M6 << endl;
34     cout << "M7␣=␣\n" << M7 << endl;
35 }
```

3.3 行列の次元と型

行列の次元や要素の型も，ベクトルと同様に宣言できる．命名規則は，

- Matrix[次元][型]

である．「次元」には 2, 3, 4, X があり，それぞれ 2 次元，3 次元，4 次元，動的次元（可変長次元）を表す．また「型」には d, f, i があり，それぞれ double 型，float 型，int 型を表す[*3]．

サンプルプログラム 9　行列の次元と型

```
1  #include <iostream>
2  #include <Eigen/Core> // 基本演算のみ
3
4  using namespace Eigen;
5  using namespace std;
6
7  int main()
8  {
9     // よく使われる 2 × 2, 3 × 3, 4 × 4 行列は用意されている.
10    Matrix2d A; // double 型の 2 × 2 行列
11    Matrix2f B; // float 型の 2 × 2 行列
12    Matrix2i C; // int 型の 2 × 2 行列
13
14    Matrix3d D; // double 型の 3 × 3 行列
15    Matrix3f E; // float 型の 3 × 3 行列
16    Matrix3i F; // int 型の 3 × 3 行列
17
18    Matrix4d G; // double 型の 4 × 4 行列
19    Matrix4f H; // float 型の 4 × 4 行列
20    Matrix4i P; // int 型の 4 × 4 行列
21
22    // 可変長サイズの行列も利用できる
23    MatrixXd Q; // double 型の可変サイズ行列
24    MatrixXf R; // float 型の可変サイズ行列
25    MatrixXi S; // double 型の可変サイズ行列
26 }
```

[*3] 行列に対しても複素数型が用意されているが，本書では使わないので省略.

3.4 行列の演算

ベクトルと同じように，行列の演算もプログラム 10 に示すように直観的な記述ができる．

- 18 行目は代入（コピー）．
- 19 行目は単項のマイナス．
- 25〜27 行目はスカラーとの乗算および除算．
- 29, 30 行目は代入演算子によるスカラーとの乗算および除算．
- 38〜40 行目は行列の加算と減算，乗算．
- 41〜43 行目は代入演算子による加算と減算，乗算．

サンプルプログラム 10　行列の和，差，スカラー倍

```
1  #include <iostream>
2  #include <Eigen/Core> // 基本演算
3
4  using namespace Eigen;
5  using namespace std;
6
7  int main()
8  {
9     Matrix3d A, Ac, Am, As, As2, sA, Ad, Ad2, A2, A3, A4, B, C, D, E;
10    double s;
11
12    A = Matrix3d::Random(); // ランダムな要素の行列を生成
13    B = Matrix3d::Random();
14
15    cout << "A␣=␣\n" << A << endl;
16    cout << "B␣=␣\n" << B << endl;
17
18    Ac = A; // 代入（値のコピー）
19    Am = -A; // 単項のマイナス
20
21    cout << "Ac␣=␣\n" << Ac << endl;
22    cout << "Am␣=␣\n" << Am << endl;
23
24    s = 3.0;
25    sA = s * A; // スカラー倍
26    As = A * s; // スカラー倍
27    Ad = A / s; // スカラー除算
28    As2 = Ad2 = A; // 多重代入も可能
29    As2 *= s; // 代入演算子によるスカラー倍
30    Ad2 /= s; // 代入演算子によるスカラー除算
31
32    cout << "sA␣=␣\n" << sA << endl;
33    cout << "As␣=␣\n" << As << endl;
34    cout << "As2␣=␣\n" << As2 << endl;
35    cout << "Ad␣=␣\n" << Ad << endl;
36
```

```
37     A2 = A3 = A4 = A;
38     C = A + B; // 行列の加算
39     D = A - B; // 行列の減算
40     E = A * B; // 行列の乗算
41     A2 += B; // 代入演算子による加算
42     A3 -= B; // 代入演算子による減算
43     A4 *= B; // 代入演算子による乗算
44
45     cout << "A␣+␣B␣=␣\n" << C << endl;
46     cout << "A␣-␣B␣=␣\n" << D << endl;
47     cout << "A␣*␣B␣=␣\n" << E << endl;
48     cout << "A2␣=␣\n" << A2 << endl;
49     cout << "A3␣=␣\n" << A3 << endl;
50     cout << "A4␣=␣\n" << A4 << endl;
51 }
```

次のプログラム 11 のように，行列のサイズが異なる演算があると，コンパイル時にエラーとなる．

サンプルプログラム 11　異なるサイズの行列の乗算

```
1  #include <iostream>
2  #include <Eigen/Core> // 基本演算
3
4  using namespace Eigen;
5  using namespace std;
6
7  int main()
8  {
9     Matrix3d A, B, C;
10    Matrix4d D, E, F;
11    double s;
12
13    A = Matrix3d::Random(); // ランダムな要素の 3 × 3 行列を生成
14    B = Matrix3d::Random();
15
16    D = Matrix4d::Random(); // ランダムな要素の 4 × 4 行列を生成
17    E = Matrix4d::Random();
18
19    cout << "A␣=␣\n" << A << endl;
20    cout << "B␣=␣\n" << B << endl;
21
22    cout << "D␣=␣\n" << D << endl;
23    cout << "E␣=␣\n" << E << endl;
24
25    C = A * B; // 乗算可能
26    F = D * E; // 乗算可能
27
28    cout << "A␣*␣B␣=␣\n" << C << endl;
29    cout << "D␣*␣E␣=␣\n" << F << endl;
30
31    // 以下は乗算が不可能なため, コンパイル時にエラー
```

```
32    // F = A * D;
33    // F = C * B;
34 }
```

3.5 ほかのサイズの行列や特殊な行列

一般の固定長次元のベクトルや正方行列以外は，次のプログラム 12 の 7〜10 行目のように typedef と組み合わせて，用意されている型と同様に扱える．

サンプルプログラム 12　一般の固定サイズ行列

```
1  #include <iostream>
2  #include <Eigen/Core> // 基本演算のみ
3
4  using namespace Eigen;
5  using namespace std;
6
7  typedef Matrix<double,6,6> Matrix6d; // 6 × 6(固定長) 行列 (double)
8  typedef Matrix<double,9,9> Matrix9d; // 9 × 9(固定長) 行列 (double)
9  typedef Matrix<int,6,6> Matrix6i; // 6 × 6(固定長) 行列 (int)
10 typedef Matrix<float,9,9> Matrix9f; // 9 × 9(固定長) 行列 (float)
11
12 typedef Matrix<double,6,3> Matrix63d; // 6 × 3(固定長) 行列 (double)
13
14 int main()
15 {
16    // 上記の行列の利用
17    Matrix6d A; // double 型の 6 × 6 行列
18    Matrix9f B; // float 型の 9 × 9 行列
19    Matrix63d C; // float 型の 6 × 3 行列
20 }
```

プログラム 13 に示すように，対角行列に対する型や初期化も用意されており，通常の行列と同様に演算できる．

サンプルプログラム 13　対角行列

```
1  #include <iostream>
2  #include <Eigen/Core> // 基本演算のみ
3
4  using namespace Eigen;
5  using namespace std;
6
7  // 3 × 3 の対角行列
8  typedef DiagonalMatrix<double,3> DiagMatrix3d;
9
10 int main()
11 {
12    DiagonalMatrix<double,3> D1; // 通常の宣言
```

```
13    DiagMatrix3d D2; // 上記 typedef を用いた宣言
14
15    // コンストラクタによる初期化
16    DiagonalMatrix<double,3> D3(1.23, 2.34, 3.45);
17
18    Vector3d v1, v2;
19
20    v1 = Vector3d::Random();
21    D1.diagonal() = v1; // ベクトルによる初期化
22    v2 = Vector3d::Ones();
23    D2 = v2.asDiagonal(); // ベクトルを対角行列に変換する初期化
24
25    cout << "D1␣=␣\n" << D1.diagonal() << endl;
26    cout << "D2␣=␣\n" << D2.diagonal() << endl;
27    cout << "D3␣=␣\n" << D3.diagonal() << endl;
28
29    Matrix3d M, M1, M2;
30
31    M = Matrix3d::Random();
32
33    M1 = D1; // 通常の行列への代入
34    M2 = D1 * M; // 通常の行列との積
35
36    cout << "M1␣=␣\n" << M1 << endl;
37    cout << "M2␣=␣\n" << M2 << endl;
38 }
```

3.6 ベクトルと行列の演算

ベクトルと行列との演算もプログラム 14 のように，数式のような直観的な記述ができる．計算できない組み合わせもコンパイル時にエラーとなるため，正しい演算かどうかチェックできる．

- 23 行目は行列 × 列ベクトルであり，得られる結果は列ベクトル．25 行目は乗算が計算できないので，コンパイル時にエラー．
- 28 行目は行ベクトル × 行列であり，得られる結果は行ベクトル．30 行目は乗算が計算できないので，コンパイル時にエラー．
- 37 行目は代入演算子による行ベクトルと行列との積．39 行目のように，行列と列ベクトルとの乗算代入演算子はエラー．
- 44, 45 行目は行ベクトルと列ベクトル間の変換．
- 48 行目は列ベクトルと行ベクトルの積で，結果は行列．
- 53 行目は行ベクトルと列ベクトルの積（=内積と等価）で，結果はスカラー．
- 61, 62 行目のように，メンバー関数 `row()` や `col()` を使うことで，行列の各行や各列を取り出せる．

サンプルプログラム 14　ベクトルと行列の演算

```
1  #include <iostream>
2  #include <Eigen/Core> // 基本演算
3
4  using namespace Eigen;
5  using namespace std;
6
7  int main()
8  {
9     Vector3d c1, c2, c3;
10    RowVector3d r1, r2, r3; // 行ベクトル
11    Matrix3d M1, M2;
12    double s;
13
14    c1 = Vector3d::Random();
15    r1 = RowVector3d::Random();
16    M1 = Matrix3d::Random();
17
18    cout << "c1␣=␣" << c1.transpose() << endl;
19    cout << "r1␣=␣" << r1 << endl;
20    cout << "M1␣=␣\n" << M1 << endl;
21
22    // 行列×列ベクトル
23    c2 = M1 * c1;
24    // 次式は計算できないので, コンパイル時にエラー
25    // c2 = c1 * M1;
26
27    // 行ベクトル×行列
28    r2 = r1 * M1;
29    // 次式は計算できないので, コンパイル時にエラー
30    // r2 = M1 * r1;
31
32    cout << "M1␣*␣c1␣=␣" << c2.transpose() << endl;
33    cout << "r1␣*␣M1␣=␣" << r2 << endl;
34
35    r3 = r1; c3 = c1;
36    // 以下の代入演算子による行ベクトル×行列は可能
37    r3 *= M1;
38    // 以下の代入演算子による行列×列ベクトルは不可
39    // c3 *= M1;
40
41    cout << "r1␣*␣M1␣=␣" << r3 << endl;
42
43    // 行ベクトルと列ベクトルの変換
44    c3 = r1.transpose();
45    r3 = c1.transpose();
46
47    // 列ベクトルと行ベクトルの積 (=行列)
48    M2 = c3 * r3;
49
50    cout << "c3␣*␣r3␣=␣\n" << M2 << endl;
51
52    // 行ベクトルと列ベクトルの積 (=内積)
```

```
53     s = r3 * c3;
54
55     cout << "r3␣*␣c3␣=␣" << s << endl;
56
57     // 行列の列や行の取り出し
58     Vector3d mc0, mc1, mc2;
59     RowVector3d mr0, mr1, mr2;
60
61     mc0 = M1.col(0); mc1 = M1.col(1); mc2 = M1.col(2); // 列の取り出し
62     mr0 = M1.row(0); mr1 = M1.row(1); mr2 = M1.row(2); // 行の取り出し
63
64     cout << "Matrix␣M1␣=␣\n" << M1 << endl;
65     cout << "1st␣col.␣of␣M1␣=␣" << mc0 << endl;
66     cout << "2nd␣col.␣of␣M1␣=␣" << mc1 << endl;
67     cout << "3rd␣col.␣of␣M1␣=␣" << mc2 << endl;
68     cout << "1st␣row␣␣of␣M1␣=␣" << mr0 << endl;
69     cout << "2nd␣row␣␣of␣M1␣=␣" << mr1 << endl;
70     cout << "3rd␣row␣␣of␣M1␣=␣" << mr2 << endl;
71 }
```

プログラム 14 の出力は次のようになる.

```
1  c1 = 0.680375 -0.211234 0.566198
2  r1 = 0.59688 0.823295 -0.604897
3  M1 =
4   -0.329554 0.10794 -0.270431
5    0.536459 -0.0452059 0.0268018
6   -0.444451 0.257742 0.904459
7  M1 * c1 = -0.400139 0.389718 0.155266
8  r1 * M1 = 0.513806 -0.128698 -0.686454
9  r1 * M1 = 0.513806 -0.128698 -0.686454
10 c3 * r3 =
11  0.406103 -0.126081 0.337953
12  0.560149 -0.173908 0.466148
13 -0.411557 0.127775 -0.342492
14 r3 * c3 = -0.110297
15 Matrix M1 =
16  -0.329554 0.10794 -0.270431
17   0.536459 -0.0452059 0.0268018
18  -0.444451 0.257742 0.904459
19 1st col. of M1 = -0.329554
20  0.536459
21 -0.444451
22 2nd col. of M1 = 0.10794
23 -0.0452059
24   0.257742
25 3rd col. of M1 = -0.270431
26 0.0268018
27  0.904459
28 1st row of M1 = -0.329554 0.10794 -0.270431
29 2nd row of M1 = 0.536459 -0.0452059 0.0268018
30 3rd row of M1 = -0.444451 0.257742 0.904459
```

4　行列式と逆行列

Eigen では，行列式や逆行列はメンバー関数として用意されている．使い方は簡単で，プログラム 15 のようにすればよい．

- 行列式や逆行列は Core ヘッダファイルでは提供されていないため，3 行目のように LU ヘッダファイルをインクルードする必要がある．
- 19 行目が行列式．
- 20 行目が逆行列．

これらを使うときは，逆行列や行列式を何に使うのかを確認してから使うべきである．たとえば，連立方程式を解くには，後述の LU 分解を用いるほうが一般には効率的で，精度もよい．しかし Eigen では，4×4 程度の固定サイズの行列であれば，逆行列や行列式が効率的に計算できるように実装されている．

サンプルプログラム 15　行列式と逆行列

```
1  #include <iostream>
2  #include <Eigen/Core> // 基本演算のみ
3  #include <Eigen/LU> // 行列の基本的分解ほか
4
5  using namespace Eigen;
6  using namespace std;
7
8  int main()
9  {
10    Matrix3d M; // 3 × 3 行列, 要素は double
11
12    // 行列に値を代入
13    M << 1.0, 0.5, 2.3,
14         -4.8, 0.9, -1.6,
15         3.2, -0.6, -3.7;
16
17    cout << "matrix␣M␣=␣\n" << M << endl; // そのまま出力
18
19    cout << "determinant␣of␣M␣=␣" << M.determinant() << endl; // 行列式
20    cout << "inverse␣of␣M␣=␣\n" << M.inverse() << endl; // 逆行列
21 }
```

5　行列の固有値

5.1　固有値問題

本書でよく用いる対称行列 $\boldsymbol{M}$ に対する固有値問題

$$\boldsymbol{M}\boldsymbol{\theta} = \lambda\boldsymbol{\theta} \tag{1}$$

を解くには，プログラム 16 のように書けばよい．

- 固有値計算には 3 行目の Eigen ヘッダが必要となる．
- 20 行目で，自己随伴行列（実数行列では対称行列）の固有値のソルバーのコンストラクタを呼び出し，固有値と固有ベクトルを計算する．
- 失敗時のチェックには，22 行目のように info() を使えばよい．
- 固有値は，25 行目に示すメンバー関数 eigenvalues() で取り出せる．これは列ベクトルであり，昇順に並んでいる．
- 固有ベクトルは，27，28 行目のようにメンバー関数 eigenvectors() を用いて行列の形で取り出せる（対応する固有値の順に並んだ列ベクトルとなっている）．
- 最小固有値を取り出すには，31 行目の eigenvalues()(0) のように，通常のベクトルと同様に添字 0 を指定すればよい．
- 最小固有値に対する固有ベクトルを取り出すには，35 行目の eigenvectors().col(0) のようにすればよい．

サンプルプログラム 16　固有値問題

```
1  #include <iostream>
2  #include <Eigen/Core> // 基本演算のみ
3  #include <Eigen/Eigen> // 固有値
4
5  using namespace Eigen;
6  using namespace std;
7
8  int main()
9  {
10   Matrix3d M; // 3 × 3 行列, 要素は double
11
12   // 行列に値を代入
13   M << 1.0, 0.5, 2.3,
14        0.5, 0.9, -1.6,
15        2.3, -1.6, 4.8;
16
17   cout << "matrix␣M␣=␣\n" << M << endl; // そのまま出力
18
19   // 自己随伴行列（実数行列では対称放列）の固有値のソルバー
20   SelfAdjointEigenSolver<Matrix3d> ES(M);
```

```
21
22     if (ES.info() != Success) abort(); // 失敗したら終了
23
24     // 固有値はベクトルの形で得られる
25     cout << "The␣eigenvalues␣of␣M␣=␣\n" << ES.eigenvalues() << endl;
26     // 固有ベクトルは行列の形で得られる
27     cout << "The␣corresponding␣eigenvectors␣of␣M␣=␣\n"
28          << ES.eigenvectors() << endl;
29
30     // 最小固有値のみ取り出す
31     double min_eigen = ES.eigenvalues()(0);
32     cout << "The␣minimum␣eigenvalue␣=␣" << min_eigen << endl;
33
34     // 最小固有値に対する固有ベクトル
35     Vector3d min_eigen_vector = ES.eigenvectors().col(0);
36     cout << "The␣eigenvector␣corresponding␣the␣minimum␣eigenvalue␣=␣"
37          << min_eigen_vector.transpose() << endl;
38 }
```

プログラム 16 の出力は次のようになる.

```
1  matrix M =
2     1 0.5 2.3
3   0.5 0.9 -1.6
4   2.3 -1.6 4.8
5  The eigenvalues of M =
6  -0.899817
7    1.41645
8    6.18336
9  The corresponding eigenvectors of M =
10  0.685012 -0.624592 0.375025
11 -0.581021 -0.778923 -0.235994
12 -0.439516 0.0562383 0.896473
13 The minimum eigenvalue = -0.899817
14 The eigenvector corresponding the minimum eigenvalue = 0.685012 -0.581021 -0.439516
```

ループの中などで，何度もコンストラクタを呼び出したくなければ，プログラム 17 のように，最初に一度コンストラクトしておいて（11 行目），途中で計算のみ実行する (21 行目）ことも可能である.

サンプルプログラム 17　コンストラクタだけ呼び出していおいて，後で計算する場合

```
1  #include <iostream>
2  #include <Eigen/Core> // 基本演算のみ
3  #include <Eigen/Eigen> // 固有値
4
5  using namespace Eigen;
6  using namespace std;
7
8  int main()
9  {
```

```
10    Matrix3d M; // 3 × 3 行列, 要素は double
11    SelfAdjointEigenSolver<Matrix3d> ES; // ソルバーのみコンストラクト
12
13    // 行列に値を代入
14    M << 1.0, 0.5, 2.3,
15         0.5, 0.9, -1.6,
16         2.3, -1.6, 4.8;
17
18    cout << "matrix␣M␣=␣\n" << M << endl;
19
20    // M の固有値と固有ベクトルを計算
21    ES.compute(M);
22
23    if (ES.info() != Success) abort(); // 失敗したら終了
24
25    cout << "The␣eigenvalues␣of␣M␣=␣\n" << ES.eigenvalues() << endl;
26    cout << "The␣corresponding␣eigenvectors␣of␣M␣=␣\n"
27         << ES.eigenvectors() << endl;
28 }
```

5.2 一般固有値問題

対称行列 $\boldsymbol{M}$, $\boldsymbol{N}$ に対する一般固有値問題

$$\boldsymbol{M\theta} = \lambda \boldsymbol{N\theta} \tag{2}$$

を解くには，プログラム 18 のようにすればよい．

サンプルプログラム 18　一般固有値問題

```
1  #include <iostream>
2  #include <Eigen/Core> // 基本演算のみ
3  #include <Eigen/Eigen> // 固有値
4
5  using namespace Eigen;
6  using namespace std;
7
8  int main()
9  {
10    Matrix3d M, N; // 3 × 3 行列, 要素は double
11
12    // 行列に値を代入
13    M << 1.0, 0.5, 2.3,
14         0.5, 0.9, -1.6,
15         2.3, -1.6, 4.8;
16    N << 0.3, -0.3, 1.9,
17        -0.3, 1.4, -1.1,
18         1.9, -1.1, 1.5;
19
20    cout << "matrix␣M␣=␣\n" << M << endl; // そのまま出力
21    cout << "matrix␣N␣=␣\n" << N << endl; // そのまま出力
```

```
22
23    // 自己随伴行列（実数行列では対称放列）の一般固有値のソルバー
24    GeneralizedSelfAdjointEigenSolver<Matrix3d> GES(M,N);
25
26    if (GES.info() != Success) abort(); // 失敗したら終了
27
28    // 一般固有値はベクトルの形で得られる。
29    cout << "The␣generalized␣eigenvalues␣of␣M␣and␣N␣=␣\n"
30         << GES.eigenvalues() << endl;
31    // 一般固有ベクトルは行列の形で得られる。
32    cout << "The␣corresponding␣eigenvectors␣of␣M␣and␣N␣=␣\n"
33         << GES.eigenvectors() << endl;
34 }
```

```
1  matrix M =
2     1 0.5 2.3
3   0.5 0.9 -1.6
4   2.3 -1.6 4.8
5  matrix N =
6   0.3 -0.3 1.9
7  -0.3 1.4 -1.1
8   1.9 -1.1 1.5
9  The generalized eigenvalues of M and N =
10 -0.862314
11  0.628437
12   19.5865
13 The corresponding eigenvectors of M and N =
14   1.16331 -0.209096 5.00019
15 -0.542654 0.483655 0.784713
16 -0.338541 -0.147686 -0.554998
```

6 行列の分解

6.1 特異値分解

6.1.1 正方行列の特異値分解

$n \times n$ 行列 $\boldsymbol{M}$ の特異値分解とは，

$$\boldsymbol{M} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^{\top} \tag{3}$$

の形に分解することであり，行列 $\boldsymbol{U}$ および $\boldsymbol{V}$ は直交行列，$\boldsymbol{\Sigma}$ は対角行列で，その対角要素が特異値である．この計算はプログラム 19 のようにすればよい．

- 3 行目は特異値分解に必要なヘッダファイルの指定．
- 21 行目は，ヤコビ法による特異値分解．
- 特異値は，降順にソートしたベクトルとして，24 行目のように出力．

サンプルプログラム 19　正方行列の特異値分解

```
#include <iostream>
#include <Eigen/Core> // 基本演算のみ
#include <Eigen/SVD> // 特異値分解

using namespace Eigen;
using namespace std;

int main()
{
   Matrix3d M; // 3 × 3 行列, 要素は double

   // 行列に値を代入
   M << 1.0, 0.5, 2.3,
        0.5, 0.9, -1.6,
        2.3, -1.6, 4.8;

   cout << "matrix␣M␣=␣\n" << M << endl; // そのまま出力

   // 特異値分解.
// JacobiSVD<Matrix3d> SVD(M); 何も指定しないと, 特異値のみ計算
   JacobiSVD<Matrix3d> SVD(M, ComputeFullU | ComputeFullV);

   // 特異値
   cout << "The␣singular␣values␣of␣M␣=␣\n" << SVD.singularValues() << endl;
   // 行列 U および V
   cout << "The␣U␣matrix␣of␣M␣=␣\n" << SVD.matrixU() << endl;
   cout << "The␣V␣matrix␣of␣M␣=␣\n" << SVD.matrixV() << endl;

   // もとの行列になるかを確認
   Matrix3d M2;
   M2 = SVD.matrixU() * SVD.singularValues().asDiagonal() * SVD.matrixV().transpose();

   cout << "matrix␣M2␣=␣\n" << M2 << endl;
}
```

プログラム 19 の出力は次のようになる.

```
matrix M =
   1 0.5 2.3
 0.5 0.9 -1.6
 2.3 -1.6 4.8
The singular values of M =
 6.18336
 1.41645
0.899817
The U matrix of M =
  0.375025 0.624592 0.685012
 -0.235994 0.778923 -0.581021
  0.896473 -0.0562383 -0.439516
The V matrix of M =
  0.375025 0.624592 -0.685012
```

```
 -0.235994 0.778923 0.581021
  0.896473 -0.0562383 0.439516
matrix M2 =
   1 0.5 2.3
 0.5 0.9 -1.6
 2.3 -1.6 4.8
```

6.1.2 正方行列でない場合

正方行列でない $n \times p$ 行列 $\boldsymbol{M}$ の特異値分解では，$\boldsymbol{U}$ は $n \times m$ 行列，行列 $\boldsymbol{\Sigma}$ は $m \times m$ 行列，行列 $\boldsymbol{V}$ は $p \times m$ 行列であり，m は p と n の小さいほうとなる．プログラム 20 が正方行列でない場合であり，このような場合は可変サイズの行列である必要がある．そして，15 行目のように計算のオプション ComputeThinU | ComputeThinV が必要となる．

サンプルプログラム 20　正方行列でない行列の特異値分解

```
#include <iostream>
#include <Eigen/Core> // 基本演算のみ
#include <Eigen/SVD> // 特異値分解

using namespace Eigen;
using namespace std;

int main()
{
   MatrixXd M = MatrixXd::Random(3,2); // 3 × 2 行列, 要素は double，動的サイズ行列である必要がある.

   cout << "matrix␣M␣=␣\n" << M << endl; // そのまま出力

   // 正方行列でない行列の特異値分解. ComputeThinU と ComputeThinV が必要.
   JacobiSVD<MatrixXd> SVD(M, ComputeThinU | ComputeThinV);

   // 特異値
   cout << "The␣singular␣values␣of␣M␣=␣\n" << SVD.singularValues() << endl;
   // 行列 U, V
   cout << "The␣U␣matrix␣of␣M␣=␣\n" << SVD.matrixU() << endl;
   cout << "The␣V␣matrix␣of␣M␣=␣\n" << SVD.matrixV() << endl;

   // もとの行列になるかを確認
   MatrixXd M2;
   M2 = SVD.matrixU() * SVD.singularValues().asDiagonal() * SVD.matrixV().transpose();

   cout << "matrix␣M2␣=␣\n" << M2 << endl;
}
```

6.2 LU 分解

LU 分解とは，正方行列 $\boldsymbol{M}$ を下三角行列 $\boldsymbol{L}$ と上三角行列 $\boldsymbol{U}$ の積，すなわち

$$\boldsymbol{M} = \boldsymbol{L}\boldsymbol{U} \tag{4}$$

のように分解することである．Eigen では，置換行列 $\boldsymbol{P}$ を用いて

$$\boldsymbol{M} = \boldsymbol{P}^{-1}\boldsymbol{L}\boldsymbol{U} \tag{5}$$

となる分解，あるいはもう一つの置換行列 $\boldsymbol{Q}$ を用いて

$$\boldsymbol{M} = \boldsymbol{P}^{-1}\boldsymbol{L}\boldsymbol{U}\boldsymbol{Q}^{-1} \tag{6}$$

となる分解を求めることができる．ただし，式 (5) の分解を行うためには行列 $\boldsymbol{M}$ は正則でなければならない[*4]．

- 3 行目は LU 分解に必要なヘッダファイルの指定．
- 20 行目で $\boldsymbol{M} = \boldsymbol{P}^{-1}\boldsymbol{L}\boldsymbol{U}$ となる分解を求める．
- 23〜27 行目で分解結果の行列 $\boldsymbol{P}$, $\boldsymbol{L}$, $\boldsymbol{U}$ を計算．ただし，$\boldsymbol{L}$ と $\boldsymbol{U}$ は，メンバー関数 `matrixLU()` により，一つの行列として求まっているため，それぞれの上三角および下三角要素を取り出し，行列 $\boldsymbol{L}$ に単位行列を加える必要がある．
- 36 行目で $\boldsymbol{M} = \boldsymbol{P}^{-1}\boldsymbol{L}\boldsymbol{U}\boldsymbol{Q}^{-1}$ となる分解を求める．
- 39〜44 行目で分解結果の行列 $\boldsymbol{P}$, $\boldsymbol{L}$, $\boldsymbol{U}$, $\boldsymbol{Q}$ を計算．

サンプルプログラム 21　LU 分解

```
1  #include <iostream>
2  #include <Eigen/Core> // 基本演算のみ
3  #include <Eigen/LU> // LU 分解，逆行列，行列式など
4
5  using namespace Eigen;
6  using namespace std;
7
8  int main()
9  {
10   Matrix3d M; // 3 × 3 行列, 要素は double
11
12   // 行列に値を代入
13   M << 5.0, 1.0, -2.0,
14        1.0, 6.0, -1.0,
15        2.0, -1.0, 5.0;
16
17   cout << "matrix␣M␣=␣\n" << M << endl; // そのまま出力
```

[*4] 実際には計算法が異なり，計算速度も異なる．

```
18
19    // 部分ピボットによる M=P^{-1}*L*U への分解
20    PartialPivLU<Matrix3d> LUppv(M);
21
22    // 行列 L, U と P
23    Matrix3d P1= LUppv.permutationP();
24    Matrix3d U1= LUppv.matrixLU().triangularView<Upper>(); // 行列 U
25    Matrix3d L1= LUppv.matrixLU().triangularView<StrictlyLower>(); // 行列 L の一部
26
27    L1 += Matrix3d::Identity(3,3); // 単位行列を足して完全な行列 L となる
28
29    cout << "Matrix␣P1␣=␣\n" << P1 << endl;
30    cout << "Matrix␣L1␣=␣\n" << L1 << endl;
31    cout << "Matrix␣U1␣=␣\n" << U1 << endl;
32
33    cout << "P^{-1}*L*U␣=␣\n" << P1.inverse() * L1 * U1 << endl << endl;
34
35    // 完全ピボットによる M=P^{-1}*L*U*Q^{-1}への分解
36    FullPivLU<Matrix3d> LUfpv(M);
37
38    // 行列 P, Q, L, U
39    Matrix3d P2= LUfpv.permutationP();
40    Matrix3d Q2= LUfpv.permutationQ();
41    Matrix3d U2= LUfpv.matrixLU().triangularView<Upper>(); // 行列 U
42    Matrix3d L2= LUfpv.matrixLU().triangularView<StrictlyLower>(); // 行列 L の一部
43
44    L2 += Matrix3d::Identity(3,3); // 単位行列を足して完全な行列 L となる
45
46    cout << "Matrix␣P2␣=␣\n" << P2 << endl;
47    cout << "Matrix␣L2␣=␣\n" << L2 << endl;
48    cout << "Matrix␣U2␣=␣\n" << U2 << endl;
49    cout << "Matrix␣Q2␣=␣\n" << Q2 << endl;
50
51    cout << "P^{-1}*L*U*Q^{-1}␣=␣\n" << P2.inverse() * L2 * U2 * Q2.inverse() << endl;
52 }
```

プログラム 21 の出力は次のようになる.

```
1  matrix M =
2   5 1 -2
3   1 6 -1
4   2 -1 5
5  Matrix P1 =
6  1 0 0
7  0 1 0
8  0 0 1
9  Matrix L1 =
10         1 0 0
11       0.2 1 0
12       0.4 -0.241379 1
13 Matrix U1 =
14       5 1 -2
```

```
      0 5.8 -0.6
      0 0 5.65517
P^{-1}*L*U =
 5 1 -2
 1 6 -1
 2 -1 5

Matrix P2 =
0 1 0
1 0 0
0 0 1
Matrix L2 =
       1 0 0
 0.166667 1 0
-0.166667 0.448276 1
Matrix U2 =
       6 1 -1
       0 4.83333 -1.83333
       0 0 5.65517
Matrix Q2 =
0 1 0
1 0 0
0 0 1
P^{-1}*L*U*Q^{-1} =
 5 1 -2
 1 6 -1
 2 -1 5
```

6.3 コレスキー分解

コレスキー分解とは，正値実対称行列 $\boldsymbol{M}$ を，下三角行列 $\boldsymbol{L}$ を用いて

$$\boldsymbol{M} = \boldsymbol{L}\boldsymbol{L}^{\top} \tag{7}$$

のように分解することである．次のサンプルプログラム 22 のように書けばよい．

- 3 行目はコレスキー分解に必要なヘッダファイルの指定．
- 20 行目はコレスキー分解．

サンプルプログラム 22　コレスキー分解

```
#include <iostream>
#include <Eigen/Core> // 基本演算のみ
#include <Eigen/Cholesky> // コレスキー分解

using namespace Eigen;
using namespace std;

int main()
```

```
{
   Matrix3d M; // 3 × 3 行列, 要素は double

   // 行列に値を代入 (正値対称行列)
   M << 5.0, 1.0, -2.0,
        1.0, 6.0, -1.0,
        -2.0, -1.0, 5.0;

   cout << "matrix␣M␣=␣\n" << M << endl; // そのまま出力

   // コレスキー分解.
   LLT<Matrix3d> CholD(M);

   // 行列 L
   Matrix3d L= CholD.matrixL();

   cout << "Matrix␣L␣=␣\n" << L << endl;

   // 元の行列になるかを確認
   Matrix3d M2;
   M2 = L * L.transpose();

   cout << "matrix␣M2␣=␣\n" << M2 << endl;
}
```

プログラム 22 の出力は次のようになる.

```
matrix M =
 5 1 -2
 1 6 -1
-2 -1 5
Matrix L =
  2.23607 0 0
 0.447214 2.40832 0
-0.894427 -0.249136 2.03419
matrix M2 =
 5 1 -2
 1 6 -1
-2 -1 5
```

7　具体的な計算例

以下，クラスやモジュールとして用意されていないものも含めて，本書でよく使う計算のプログラム例を示す.

7.1　一般逆行列

本書でよく用いる対称行列 $\boldsymbol{M}$ のランク n の一般逆行列は，プログラム 23 のように書ける.

- 11〜35 行がランク r の一般逆行列を求める関数.
- 44〜50 行はチェック用にランク r の実対称行列を生成している.

サンプルプログラム 23　ランク n の一般逆行列

```
1  #include <iostream>
2  #include <Eigen/Core> // 基本演算のみ
3  #include <Eigen/Eigen> // 固有値
4
5  using namespace Eigen;
6  using namespace std;
7
8  typedef Matrix<double,6,1> Vector6d;
9  typedef Matrix<double,6,6> Matrix6d;
10
11 // n × n 対称行列 M のランク r の一般逆行列
12 template<class MatT>
13 MatT GenInv(const MatT& M, int r)
14 {
15    if (M.cols() != M.rows()) abort(); // 正方行列でない
16
17    int n = M.cols(); // サイズ
18
19    // 自己随伴行列（実数行列では対称放列）の固有値のソルバー
20    SelfAdjointEigenSolver<MatT> ES(M);
21
22    if (ES.info() != Success) abort(); // 失敗したら終了
23
24    // 一般逆行列用の行列を零行列に初期化
25    MatT GenI = MatT::Zero();
26
27    // 一般逆行列の計算
28    for (int i = n - r; i < n; i++)
29    {
30       GenI += ES.eigenvectors().col(i) * ES.eigenvectors().col(i).transpose() / ES.eigenvalues()(i);
31    }
32
33    // 戻り
34    return GenI;
35 }
36
37 int main()
38 {
39    Vector6d v; // 初期化用ベクトル
40    Matrix6d M; // 対象とする行列
41    Matrix6d Mi; // 一般逆行列
42
43    // ランク 5 の対称行列作成
44    Matrix6d T, T2;
45    T = Matrix6d::Random(); // ランダムな要素の行列を生成
46    T2 = (T + T.transpose()) / 2.0; // 対称行列化
47    T = T2 * T2.transpose(); // 正値化
```

```
48    SelfAdjointEigenSolver<Matrix6d> ES(T);
49    M = Matrix6d::Zero(); // 零行列に初期化
50    for (int i = 1; i < 6; i++)
51    {
52       M += ES.eigenvectors().col(i) * ES.eigenvectors().col(i).transpose() * ES.eigenvalues()(i);
53    }
54    cout << "Original␣matrix␣M␣=␣\n" << M << endl;
55
56    // ランク 5 の一般逆行列
57    Mi = GenInv(M, 5);
58    cout << "Generalized␣inverse␣of␣M␣with␣rank␣5␣=␣\n" << Mi << endl;
59 }
```

7.2 連立方程式の解

未知数ベクトル $\boldsymbol{x}$ に関する連立方程式

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b} \tag{8}$$

の解は，行列 $\boldsymbol{A}$ が正則であれば

$$\boldsymbol{x} = \boldsymbol{A}^{-1}\boldsymbol{b} \tag{9}$$

のように逆行列を使って求まるが，一般には計算コストが高いため，前述の LU 分解を使うのがよい．プログラム 24 では，FullPivLU を使った LU 分解を示している．

サンプルプログラム 24　LU 分解による連立方程式の解

```
1  #include <iostream>
2  #include <Eigen/Core> // 基本演算のみ
3  #include <Eigen/LU>
4
5  using namespace Eigen;
6  using namespace std;
7
8  int main()
9  {
10    Matrix3d A;
11    Vector3d x, b;
12
13    // 係数行列のセット
14    A << 1.0, 2.0, 1.0,
15         2.0, 1.0, 0.0,
16        -1.0, 1.0, 2.0;
17    cout << "matrix␣A␣=␣\n" << A << endl;
18
19    // ベクトル b のセット
20    b << -1.0, 2.0, 1.5;
21    cout << "vector␣b␣=␣" << b.transpose() << endl;
22
23    // FullPivLU を使った LU 分解
```

```
24    FullPivLU<Matrix3d> lu(A);
25
26    // LU 分解を使った連立方程式の解
27    x = lu.solve(b);
28
29    cout << "solution␣x␣=␣" << x.transpose() << endl;
30 }
```

7.3 特異値分解による最小 2 乗解

最小 2 乗法は，正規方程式から一般逆行列により求める方法もあるが，これを式どおりに計算すると，内部で逆行列の計算が必要となる．このため，プログラム 25 のような，特異値分解を使った方法がよく用いられる．

サンプルプログラム 25　特異値分解による最小 2 乗解

```
1  #include <iostream>
2  #include <Eigen/Core> // 基本演算のみ
3  #include <Eigen/SVD>
4
5  using namespace Eigen;
6  using namespace std;
7
8  typedef Matrix<double,6,1> Vector6d;
9
10 int main()
11 {
12    MatrixXd A(6,4); // 動的サイズ行列. 後で初期化.
13    Vector6d b;
14    Vector4d x;
15
16    // 係数行列のセット
17    A = MatrixXd::Random(6,4);
18    cout << "matrix␣A␣=␣\n" << A << endl;
19
20    // ベクトル b のセット
21    b = Vector6d::Random();
22    cout << "vector␣b␣=␣" << b.transpose() << endl;
23
24    // JacobiSVD を使った特異値分解
25    JacobiSVD<MatrixXd> svd(A, ComputeThinU|ComputeThinV);
26
27    // 最小 2 乗解
28    x = svd.solve(b);
29
30    cout << "A␣least-squares␣solution␣=␣" << x.transpose() << endl;
31 }
```

8　おわりに

ここでは，本書に出てくる主な計算に対して，Eigen を用いた実装例を示した．Eigen はここに示した使い方だけでなく，いろいろな分野に応用可能なライブラリとなっている．興味のある読者は，Web ページのドキュメントに詳しく正確な記述であるので，それを参照してほしい．

参考文献

[1] 金谷健一, 菅谷保之, 金澤 靖, 3 次元コンピュータビジョン計算ハンドブック, 森北出版, 2016.

[2] B.W. カーニハン, D.M. リッチー, プログラミング言語 C 第 2 版 ANSI 規格準拠, 共立出版, 1989.

[3] 藤原博文, 改訂新版 C プログラミング診断室, 技術評論社, 2003.

[4] B. ストラウストラップ, プログラミング言語 C++ 第 4 版, SB クリエイティブ, 2015.

[5] 大城正典, 詳説 C++ 第 2 版, ソフトバンククリエイティブ, 2005.