公開コピー誌

行列ライブラリEigenのメモ

暗黒通信団

❶ はじめに

本稿は、C++ の行列演算において標準的な位置を占めつつある Eigen ライブラリの基本的な使い方を示したものである。Eigen はヘッダのみで構成される純粋なテンプレートライブラリで、コンパイルは不要だ。Eigen の特長は高速であることだ。長い行列演算を遅延評価し、無駄な中間行列を作らずに計算する。そして高速な CPU 命令を使いベクトル化している。記述は簡便で、コンパイラはまっとうな C++ コンパイラであれば何でも使える[*1]。画面出力は `cout` だけで大体 OK だ。インストールも「`http://eigen.tuxfamily.org`」から最新版をおとし、`eigen/Eigen` 以下を`/usr/include` あたりに展開するだけだ。

こんなに便利なのに日本語のドキュメントが少なすぎるので、日本人はいまだに有料の MATLAB を使い続けていたりする。あまりに悲しいので啓蒙用のノートを作ることにした。

ソースを書く際には、「`Eigen/Dense`」を include しておけば、`Eigen/Core, Eigen/Geometry, Eigen/LU, Eigen/Cholesky, Eigen/SVD, Eigen/QR, Eigen/Eigenvalues` が勝手に読み込まれる。読み込まれないのは `Eigen/Householder` のみである。ただし後述するスパース行列の場合は、「`Eigen/Sparse`」を include する。`Eigen/Dense` は楽でよいのだが、テンプレートライブラリの宿命としてコンパイルに時間が掛かる。そこで使う機能が限定されているなら、`Eigen/Core` と、機能に特化したヘッダを個別に include する方が良い。

❷ 基本

以下では「`using namespace std;`」と「`using namespace Eigen;`」を前提にする。この書法を嫌う人は多いが、これを前提にしないとかなり記述が煩雑になるので、気になる人は脳内でいちいち「`std::`」とか「`Eigen::`」をつけてほしい。また（これも一般論だが）外部の関数で Eigen のインスタンスを呼ぶときは参照渡しかポインタ渡しにするべきである。

❶ クラスの命名規則

基本となるクラスは `Matrix` と `Array` と `Map` であるが、`Map` はデータコンバート用のクラスなので後回しにする。そして `Array` と `Matrix` の違いは演算規則が行列的かどうかだ。例えば `Array` 同士の掛け算は単に要素ごとの掛け算をするだけだが、`Matrix` の掛け算は行列の積になる。以下は `Matrix` クラスとその派生を宣言する方法である。

- `Matrix3d` (3×3 で double 型の 2 次元行列)、 `Matrix3f` (同 float 型)、 `Matrix3i` (同 int 型)、 `Matrix3b` (同じ bool 型)、 `Matrix3cd` (同 complex double 型)、`Matrix3cf` (同 complex float 型) である。
- 次数は 4×4 まで直接指定。それ以外はテンプレートで初期化する。例えば「`Matrix<double, 6, 6>`」といった具合だ。長方行列の宣言もこれを使う。「`Matrix<double, 2, 3>`」とすれば、2 行 3 列の行列が作れる。
- メモリの取り方には行優先か列優先かがあるが、これも指定できる。「`Matrix<double, 6, 6, RowMajor>`」または「`Matrix<double, 6, 6, ColMajor>`」とする。デフォルトは `ColMajor` である。単にメモリ上に行列がどう展開されるかなので、座標指定でアクセスする際に指定の順番が入れ替わったりすることはない。
- 行列のサイズが不明の場合は、例えば `MatrixXd(3,3)` とか、`Matrix<double, Dynamic, Dynamic>`などとする。
- ベクトルの場合は同様に `Vector3d` などになる。不定サイズの縦ベクトルも例えば `VectorXd(3);` などとして宣言する。横ベクトルは `RowVectorXd` 型である。

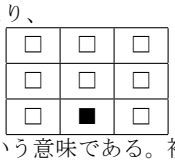
❷ 初期化

`Eigen` では、初期状態で配列の要素は不定である[*2]。だから多くの場合で初期化が必要だ。以下では仮に `Matrix3i A; Vector3i V;` について初期化するとする。初期化の方法は下記に示すようにたくさんある。

- カンマ演算子で初期化： `A << 1,1,1, 1,1,1, 1,1,1;`
- 特殊値で初期化：

```
A = MatrixXd::Zero(3,3);
A = MatrixXd::Ones(3,3);
A = MatrixXd::Constant(3,3,2);
A = MatrixXd::Random(3,3);
A = MatrixXd::Identity(3,3);//単位行列
V = VectorXd::Zero(3);
V = VectorXd::Ones(3);
V = VectorXd::Constant(3,2);
V = VectorXd::Random(3);
V = VectorXd::LinSpaced(3,1,3);//線形補完
V = VectorXd::UInit(3,2);//単位ベクトル
```

- 要素代入による初期化： 要素番号は 0 から始まり、`(行, 列)` で指定する。例えば「`A(2,1)=5;`」というのは、3 行目 2 列目を意味するので、つまり、



の位置を 5 にするという意味である。複素数型を扱う場合には、`A(0,0) = std::complex<int>(3,5);` などとする。
- 要素番号チェック無しの参照：`cout << A.coeff(2,1);`

[*1] 高速化のためには`-DEIGEN_NO_DEBUG` オプションを付けるのがよいとされる。10 倍以上速くなるらしい。

[*2] コンパイル時に`-EIGEN_INITIALIZE_MATRICES_BY_ZERO` オプションを付けるとゼロ初期化してくれるらしい。

「要素番号チェック無し」というのは、メモリ範囲外の添え字を指定していないかどうかをチェックする機構をはずすということ。少し高速になる。

- 要素番号チェック無しの代入：`A.coeffRef(2,1)=4;`

STL や一般の配列から Matrix 形式への変換は Map を使う。

```
double a[16];
for(int i=0;i<16;i++){a[i]=(double)i;}
Matrix4d A=Map<Matrix4d>(a);
```

で、A への代入ができる。二次元配列からは、例えば `A=Map<Matrix<int, Dynamic, Dynamic> >(&(a[0][0]), 4, 4);` といった書き方で初期化できる。第一引数をポインタにする点がポイントだ。STL のコンテナからの変換もこれでできる。

```
vector<double> a(16);
VectorXd V=Map<VectorXd>(&a[0], a.size());
MatrixXd A=Map<MatrixXd>(&a[0], 4, 4);
```

逆に Matrix から STL コンテナへ戻すにも Map を使う。

```
MatrixXd A(4,4);
A << 1,2,3,4, 5,6,7,8, 9,10,11,12, 13,14,15,16;
Map<MatrixXd>(&a[0], 4, 4) = A;
```

要素タイプの違う Matrix 間の変換は cast を使う。

```
Matrix3d A(1,2,3,4,5,6,7,8,9);
Matrix3f Af = A.cast<float>();
```

といった感じだ。

❸ 演算と操作

四則演算はそのまま実行できる。スカラーの四則演算の場合は、行列から一時的に Array 型にするために `array()` メンバを使う[*3]。例えば「`A.array()+1.0`」などとする。Array 型にしておけば各種の数学演算 (`abs()`, `pow()`, `min()`, `max()`, `exp()`, `log()`, `sqrt()` など) も使えて、要素毎に作用する。

これを応用すると比較演算子も使える。例えば、「`Matrix<bool, dynamic, dynamic> B=A.array()<1.0`」といった初期化が可能になる。意味は、

- `(A.array()<1.0).all()`：すべて満たせば 1
- `(A.array()<1.0).any()`：一つ満たせば 1
- `(A.array()<1.0).count()`：満たす要素数

である。なお、加減乗の計算はできるが、`A.array()` は割られることができないので、その際は逆数を掛けることで対応する。具体的には「`A.array().inverse()`」である。

その他、次のような操作コマンドがある。

- ノルムの計算：`A.norm()`, `V.norm()`
- 二乗ノルムの計算：`A.squaredNorm()`, `V.squaredNorm()` である。
- 一般の l^p ノルムの計算：`A.lpNorm<p>()`, `V.lpNorm<p>()` p を `Infinity` にすれば l^∞ ノルムになる。
- 行列 A と B のスワップ：`A.swap(B):`
- 列参照：`A.col(1)`
- 行参照：`A.row(1)`
- サイズの取得：`A.cols()`, `A.rows`, `V.size()`
- リサイズ：`A.resize()`
  リサイズによって要素の値は完全に失われるので、どこかにバックアップしておく必要がある。
- ベクトルからの対角行列： `A=V.asDiagonal();`
- 転置：`A.transpose()`
- トレース： `A.trace()`
- 要素の合計：`A.sum()`
- 要素の積：`A.prod()`
- 要素の平均：`A.mean()`
- 要素の最小値：`A.minCoeff()`
- 要素の最大値：`A.maxCoeff()`
- 対角行列：`A.diagonal()`
- 行の部分列取り出し：`A.rowwise()`
  これを組み合わせると、例えば、`A.rowwise().sum()`」などとして行列を縮約できたりする。あるいは「`A.rowwise()+=V`」のようなこともできる。
- 列の部分列取り出し：`A.colwise()`
- 部分行列取得：`A.block(0,0,2,2);//[0,0]` から 2×2 行列を返す。
- その他の部分取得：名前は分かりやすい。
  `A.topLeftCorner()`, `A.topRightCorner()`, `A.bottomLeftCorner()`, `A.bottomRightCorner()`, `A.topRows()`, `A.leftCols()`, `A.bottomRows()`, `A.rightCols()`
- ベクトルの先頭と末尾：`V.head()`, `V.tail()`
- ポインタの定義：`MatrixXd::Index`
  「`MatrixXd::Index maxrow, maxcol;`
  `A.maxCoeff(&maxrow, &maxcol);`」などとして使う。

ユーザが自分で作った関数 (`myfunc()`) を Matrix の演算として使いたい場合は、「`A.unaryExpr(ptr_fun(myfunc))`」などとして登録すればよい。

Matrix と Array は、それぞれ `A.array()` と `A.matrix()` で相互変換できるが、Matrix ベースで作業していて、いちいち Array にしたくないときは、次のような簡略関数が用意されている。行列的ではない要素ごとの演算をしたいときに使う。

簡略記法	意味
`A.cwiseMin(A2)`	`A.array().min(A2.array())`
`A.cwiseMax(A2)`	`A.array().max(A2.array())`
`A.cwiseAbs()`	`A.array().abs()`
`A.cwiseSqrt(A2)`	`A.array().sqrt()`
`A.cwiseInverse()`	`A.array().inverse()`
`A.cwiseProduct(A2)`	`A.array() * A2.array()`
`A.cwiseQuotient(A2)`	`A.array() / A2.array()`

❹ 基本的な線形代数

[*3] ちなみに後述するように、Array 型には行列にするためのメンバ `matrix()` がある。これで相互変換できる。

行列計算のほとんどの（面倒な）問題は行列分解に帰着する。基本となる行列分解ソルバの速度と正確さは次の通りである。問題によって選択する[*4]。

名前	方法	速度	正確さ
PartialPivLU	LU 分解	○	△
FullPivLU	LU 分解	×	◎
HouseholderQR	QR 分解	○	△
ColPivHouseholderQR	QR 分解	△	○
FullPivHouseholderQR	QR 分解	×	◎
LLT	コレスキー分解	◎	△
LDLT	コレスキー分解	◎	○

❶ 固有値・固有ベクトル

`Eigen::EigenSolver<Matrix<double,3,3> > es(A);` とし、`es.eigenvalues()` で固有値を `es.eigenvectors()` で固有ベクトルを取り出せる。別の完全な例は以下の通りである。

```
#include <iostream>
#include <Eigen/Dense>
using namespace std;
using namespace Eigen;
int main(){
   Matrix2f A;
   A << 1, 2, 2, 3;
   SelfAdjointEigenSolver<Matrix2f> es(A);
   if (es.info() != Success) abort();
   cout << "固有値：\n"
        << es.eigenvalues() << endl;
   cout << "固有ベクトル：\n"
        << es.eigenvectors() << endl;
```

❷ 逆行列とデターミナント

小さな行列であれば `A.determinant()` と `A.inverse()` で求まる。大きな行列ではそう簡単ではない。$Ax = V$ を x について解く基本的な方法は（例えば LU 分解であれば）、`FullPivLU< Matrix3d > lu(A);` としてピボット行列を作ってから、`Vector3d x =lu.solve(V);` という感じで `x` を求める。以下はこれを一気にやる完全なソースである[*5]。

```
#include <iostream>
#include <Eigen/Dense>
using namespace std;
using namespace Eigen;
int main(){
   MatrixXd A = MatrixXd::Random(100,100);
   MatrixXd b = MatrixXd::Random(100,50);
   MatrixXd x = A.fullPivLu().solve(b);
   cout << x <<endl;
   cout << "ランク：" << lu.rank() <<endl;
   cout << "A.cols()：" << A.cols()<<endl;
   cout << "相対誤差：" << (A*x - b).norm()
                        / b.norm() << endl;
   return 0;
}
```

QR 分解で解くこともできる。

```
#include <iostream>
#include <Eigen/Dense>
using namespace std;
using namespace Eigen;
int main(){
   Matrix3f A;
   Vector3f V;
   A << 1,2,3, 4,5,6, 7,8,10;
   V << 3, 3, 4;
   cout << A.colPivHouseholderQr().solve(V) << endl;
   return 0;
}
```

コレスキー分解でもできる。

```
#include <iostream>
#include <Eigen/Dense>
using namespace std;
using namespace Eigen;
int main(){
   Matrix2f A, b;
   LLT<Matrix2f> llt;
   A << 2, -1, -1, 3;
   b << 1, 2, 3, 1;
   llt.compute(A);
   cout << "解:\n" << llt.solve(b) << endl;
   return 0;
}
```

❸ SVD, 最小二乗法

JacobiSVD クラスを使う。

```
JacobiSVD< Matrix<float, 5, 6> >
   svd(A, ComputeFullU | ComputeFullV);
cout << "特異値："<< svd.singularValues() << endl;
cout << "U 行列：" << svd.matrixU() << endl;
cout << "V 行列：" << svd.matrixV() << endl;
```

以下は `jacobiSvd` を使って最小二乗法を解くもの。

```
#include <iostream>
#include <Eigen/Dense>
```

[*4] 長方形行列の疑似逆行列を求める際には差が露骨に効いてきそうだが、あまり深く理解していない。ごめん。

[*5] ランク落ちについては `A.setThreshold();` でランク計算時の閾値を設定できる。

```
using namespace std;
using namespace Eigen;
int main(){
   MatrixXf A = MatrixXf::Random(3, 2);
   VectorXf V = VectorXf::Random(3);
   cout << "解\n"
    <<A.jacobiSvd(ComputeThinU | ComputeThinV).solve(V)
     <<endl;
   return 0;
}
```

❹ 回転行列

次のようにして二次元と三次元の回転行列を作ることができる。

```
   //2 次元
   //90 度反時計回転
   Matrix2f rot2d=Rotation2Df(3.1415926535);
   //3 次元
   Vector3f axis << 0,0,1; //z 軸;
   //Z 軸周り 90 度反時計回転
   Matrix3f rot3d=AngleAxisf(3.1415926535,axis);
```

これらの回転行列を掛けることで回転演算ができる。

❺ スパース行列について

行列計算における最大の問題は「とにかくメモリが足りなくなる」ということだ。20GB 程度のメモリがある機械でも「MatrixXd m(100000,100000);」は実行時にメモリがとれずに落ちる。行列の要素がほとんど 0 でない場合は諦めるしかないが、多くの要素が 0 であるような行列を扱う場合には「Eigen/Sparse」をインクルードし、SparseMatrix クラスを利用すると良い。これは、行列を配列ではなくリスト構造で表現することでメモリを節約している。値が 0 の要素はそもそもリストとしてメモリ上に存在しなくなる。リストというのはトリプレット（2 個の座標の組と値）を連結したものであり、ゆえに初期化もトリプレットの列を与える方法が簡便である。例を見た方が早い。

```
#include <iostream>
#include <Eigen/Sparse>
#include <Eigen/Dense>
using namespace std;
using namespace Eigen;

int main(){
SparseMatrix<double> A(10,10);
   vector<Triplet<double> > t(3);
   //ここで各 t を設定
   t[0]=Triplet<double>(1,1,3.0);
   t[1]=Triplet<double>(2,3,2.0);
   t[2]=Triplet<double>(4,5,6.0);
   A.setFromTriplets(t.begin(),t.end());

   //普通の行列宣言
   MatrixXd X(10, 10);
   //ここで値を入れる
   //演算は普通に実行可能
   MatrixXd Y=X*A;
   cout << Y <<endl;

   //For 文は Dense でないのでイテレータで回すしかない
   for(int i=0;i<A.outerSize();++i){
      for(SparseMatrix<double>
          ::InnerIterator it(A,i);it;++it){
               cout << it.value() <<",";
               cout << it.row() <<",";//SVector にはない
               cout << it.col() <<",";//SVector にはない
               cout << it.index() <<endl;
      }
   }
return 0;
}
```

ただしスパース行列についてできることは制限されており、できないこともあるので詳細はドキュメントを読むべきである。

❻ ドキュメント

以下は公式のリソースだが英語なので読むのは大変。しかし日本語のリソースは少ない。

- ドキュメント：http://eigen.tuxfamily.org/dox/
- クィックリファレンス:
  http://eigen.tuxfamily.org/dox/QuickRefPage.html

次のスライドは簡潔で参考になるがスパース行列についてはほとんど書かれていない。

- http://home.uchicago.edu/
  ～skrainka/pdfs/Talk.Eigen.pdf

行列ライブラリ Eigen のメモ

2013 年 5 月 18 日 初版（コミックアカデミー）
2013 年 5 月 20 日 Web 公開版
著　者　シンキロウ（しんきろう）
発行者　星野 香奈（ほしのかな）
発行所　同人集合　暗黒通信団（http://www.mikaka.org/~kana/）
〒277-8691 千葉県柏局私書箱 54 号 D 係
頒　価　0 円 / ISBN978-4-87310-***-* C****