Biblioteka Eigen

Filip Napierała, Robert Tyma

23 listopada 2016

Czym jest Eigen?

- Jest to otwarta biblioteka dla języka C++. Służy do operacji na macierzach, wektorach oraz do ułatwiania obliczeń w algebrze liniowej
- Najnowsza wersja biblioteki: 3.3.0
- Dokumentacja: `https://eigen.tuxfamily.org/dox/`

Instalacja biblioteki Eigen i konfiguracja środowiska

Bibliotekę Eigen możemy zainstalować w naszym systemie w dwojaki sposób:

- za pomocą menadżera pakietów Synaptic,
- poprzez wpisanie w konsoli:
  sudo apt-get install libeigen3-dev

Dla usprawnienia pracy warto ponadto w pliku .pro naszego projektu dodać linijkę:

CONFIG+=C++14

Instalacja biblioteki Eigen i konfiguracja środowiska

```
#include <iostream>
#include <eigen3/Eigen/Dense>

using namespace std;
using namespace Eigen;
```

Deklarowanie macierzy i wektorów

Eigen oferuje kilka gotowych typów macierzy i wektorów (templates), na przykład:

- **Matrix4f** macierz 4x4,
- **Vector3f** wektor trzyelementowy,
- **MatrixXf** macierz o nieznanych wymiarach (wartość dynamic jako rozmiar).

Konstruktor macierzy wygląda w ten sposób:

```
MatrixXd mat1(2,2);
MatrixXf mat2;
Matrix4i mat3;
Vector4d mat4(1.0, 1.0, 2.0, 2.0);
```

Dostęp do elementów macierzy

Do elementów macierzy dostajemy się w taki sposób:

```
MatrixXi mat(2,2);
mat(0,0) = 1;
mat(1,0) = 2;
mat(0,1) = 3;
mat(1,1) = 4;
```

inicjalizować macierz możemy także za pomocą:

```
mat << 1, 2, 3, 4;
```

Innymi podstawowymi operacjami są: rozszerzanie macierzy (**resize**) oraz wyświetlanie (przeładowana funkcja **cout**).

Macierze specjalne

Do dyspozycji mamy kilka zdefiniowanych specjalnych macierzy oraz wektorów, które są bardzo często wykorzystywane:

- Zero(),
- Identity(),
- Constant(),
- Random(),
- LinSpaced().

Tworzymy je w taki sposób:

```
Mat=Matrix3i::Random();
Wektor=VectorXi::LinSpaced(10,0,20);
    //tworzy wektor od 0 do 20 z odstępami
     równymi 2. Tożsamy ze znanym z
    Matlaba linspace(x1,x2,n)
```

Podstawowe operacje na macierzach

Biblioteka Eigen oferuje szereg podstawywych operacji na macierzach takich jak:

- dodawanie, odejmowanie, mnożenie przez skalar,
- mnożenie macierzy przez macierz,
- transpozycja **transpose(), transposeInPlace()**,
- wyznaczanie macierzy sprzężonej **conjugate()**, odwrotnej **inverse()** oraz dopełniającej **addjoint()**, **addjointInPlace()**.

Pozostałe operacje

Ponadto użytkownik może:

- wyznaczyć sumę elementów **sum()**,
- wymnożyć wszystkie elementy **prod()**,
- obliczyć średnią **mean()**,
- wyznaczyć minimum, maksimum **maxCoeff(), minCoeff()**,
- wymnożyć elementy na przekątnej **trace()**,
- szukanie na jakiej pozycji znajduje się maksimum bądź minimum **minCoeff(&i,&j)**.

Aliasing

Jest to bardzo niekorzystne zjawisko, które może doprowadzić do błędnych wyników obliczeń. Ma ono związek z tym, że niektóre obliczenia przeprowadzane są "w locie".

Metody zapobiegania: **eval()**, **noalias()**, **...inPlace()**.

Operacje arytmetyczne nie grożą aliasingiem ponieważ w ich przypadku zawsze tworzona jest lokalna kopia macierzy.

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}^T = \begin{bmatrix} 1 & & \\ & & \\ & & \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}^T = \begin{bmatrix} 1 & & \\ 2 & & \\ & & \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}^T = \begin{bmatrix} 1 & & \\ 2 & & \\ 3 & & \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}^T = \begin{bmatrix} 1 & 2 & \\ 2 & & \\ 3 & & \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}^T = \begin{bmatrix} 1 & 2 \\ 2 & 5 \\ 3 & \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}^T = \begin{bmatrix} 1 & 2 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}^T = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 5 & \\ 3 & 6 & \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}^T = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 5 & 6 \\ 3 & 6 & \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}^T = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 5 & 6 \\ 3 & 6 & 9 \end{bmatrix}$$

```
mat = mat.transpose();//aliasing
mat = mat.transposeinPlace();//zapobiegamy

mat.bottomRightCorner(2,2) = mat.
    topLeftCorner(2,2).eval();//zapobiegamy

mat.noalias() = mat * mat;//zapobiegamy
```

Operacje blokowe na macierzach

Kiedy zachodzi konieczność dostępu do wybranego fragmentu macierzy, wtedy z pomocą przychodzi szereg poleceń ułatwiających tą czynność z poleceniem **block** na czele:

```
mat.block<2,2>(1,3);
```

Pozostałe operacje ułatwiające pracę to na przykład:

- **row(i)**,
- **col(j)**,
- **leftCols(i)**,
- **bottomRows(j)**.

Corner-related operations

Eigen also provides special methods for blocks that are flushed against one of the corners or sides of a matrix or array. For instance, **.topLeftCorner()** can be used to refer to a block in the top-left corner of a matrix.

The different possibilities are summarized in the following table:

Block operation	Version constructing a dynamic-size block expression	Version constructing a fixed-size block expression
Top-left p by q block *	`matrix.topLeftCorner(p,q);`	`matrix.topLeftCorner<p,q>();`
Bottom-left p by q block *	`matrix.bottomLeftCorner(p,q);`	`matrix.bottomLeftCorner<p,q>();`
Top-right p by q block *	`matrix.topRightCorner(p,q);`	`matrix.topRightCorner<p,q>();`
Bottom-right p by q block *	`matrix.bottomRightCorner(p,q);`	`matrix.bottomRightCorner<p,q>();`
Block containing the first q rows *	`matrix.topRows(q);`	`matrix.topRows<q>();`
Block containing the last q rows *	`matrix.bottomRows(q);`	`matrix.bottomRows<q>();`
Block containing the first p columns *	`matrix.leftCols(p);`	`matrix.leftCols<p>();`
Block containing the last q columns *	`matrix.rightCols(q);`	`matrix.rightCols<q>();`

Operacje na wektorach

Operacje na wektorach zasadniczo nie różnią się od tych wykonywanych na macierzach. Możemy zatem między innymi wymnożyć wektory czy też znaleźć ich minima, maksima.

Operacje charakterystyczne dla wektora

Wyróżniamy jednak kilka charakterystycznych operacji, które możemy wykonać jedynie na wektorze. Są to:

- pobranie pierwszych n elementów **head(n)**,
- pobranie ostatnich n elementów **tail(n)**,
- Pobranie bloku ze środka wektora **segment(i,n)**.

Możemy także oczywiście pomnożyć macierz przez wektor.

Inne ciekawe funkcje

Czasami zachodzi konieczność poszukania maksimum na przykład w każdym wierszu macierzy. Wtedy możemy wykorzystać funkcję **colwise()** albo **rowwise()**.

Możemy także wykorzystać funkcje warunkowe: **any()**, **all()**, **count()**.

```
(mat > 2).count();
```

Reprezentacja macierzy w pamięci

Macierz:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix},$$

będzie reprezentowana w pamięci w sposób jednowymiarowy. W taki sposób:

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{bmatrix},$$

bądź w taki:

$$\begin{bmatrix} 1 & 4 & 7 & 2 & 5 & 8 & 3 & 6 & 9 \end{bmatrix},$$

Możemy określić w jaki sposób przechować macierz w pamięci za pomocą poleceń **RowMajor** oraz **ColMajor** w konstruktorze macierzy. Warto się zastanowić, które rozwiązanie będzie korzystniejsze.