UNIVERSITÀ DEGLI STUDI DI PADOVA

3D Data Processing

Eigen Introduction

Alberto Pretto

(Some slides taken from https://dritchie.github.io/csci2240/)

Eigen Libraries

- Eigen is a C++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms.
- Fast and well-suited for a wide range of tasks
- You can use Eigen together with OpenCV and other libraries (e.g., Ceres, which we will present later)

CMake with Eigen and OpenCV

```
project(test_3dp)

cmake_minimum_required(VERSION 3.10)

set(CMAKE_BUILD_TYPE "Release")

find_package( OpenCV REQUIRED )
find_package( Eigen3 REQUIRED )

#Add here your source files
set(TEST_3DP_SRCS src/main.cpp)

add_executable(${PROJECT_NAME} ${TEST_3DP_SRCS})
target_include_directories( ${PROJECT_NAME} PUBLIC
                            ${OpenCV_INCLUDE_DIRS}
                            ${EIGEN3_INCLUDE_DIR})
target_link_libraries(${PROJECT_NAME} ${OpenCV_LIBS})
set_target_properties(test_3dp PROPERTIES RUNTIME_OUTPUT_DIRECTORY $
{PROJECT_SOURCE_DIR}/bin)
```

Source with OpenCV and Eigen

```
#include <iostream>

#include <opencv2/opencv.hpp>
#include <Eigen/Dense>

int main(int argc, char** argv)
{

  Eigen::Matrix3d eigen_mat;
  Eigen::Vector2f eigen_vec;

  cv::Mat_<float> cv_mat(4,4);

  return 0;
}
```

Dynamic-Size Matrices in Eigen

```
#include <iostream>

#include <opencv2/opencv.hpp>
#include <Eigen/Dense>

using namespace Eigen;
using namespace std;

int main()
{
  MatrixXd m = MatrixXd::Random(3,3);
  m = (m + MatrixXd::Constant(3,3,1.2)) * 50;

  cout << "m =" << endl << m << endl;
  VectorXd v(3);
  v << 1, 2, 3;

  cout << "v =" << endl << v << endl;
  cout << "m * v =" << endl << m * v << endl;

  return 0;
}
```

Fixed-Size Matrices in Eigen

```
#include <iostream>

#include <opencv2/opencv.hpp>
#include <Eigen/Dense>

using namespace Eigen;
using namespace std;

int main()
{
  Matrix3d m = Matrix3d::Random();
  m = (m + Matrix3d::Constant(1.2)) * 50;
  cout << "m =" << endl << m << endl;

  Vector3d v(1,2,3);

  cout << "v =" << endl << v << endl;
  cout << "m * v =" << endl << m * v << endl;
}
```

Fixed VS Dynamic Size

- Use of fixed-size matrices and vectors has two advantages.
  - The compiler emits better (faster) code because it knows the size of the matrices and vectors.
  - Specifying the size in the type also allows for more rigorous checking at compile-time.
- **Practically, always use fixed-size matrices for size 4-by-4 and smaller.**

The Matrix Type

- All Matrix and Vector types are just typedef from the basic class Matrix
- The Matrix class takes six template parameters, but for now it's enough to learn about the first three parameters.

```
typedef Matrix<float, 4, 4> Matrix4f;
typedef Matrix<float, 3, 1> Vector3f;
typedef Matrix<int, 1, 2> RowVector2i;
typedef Matrix<double, Dynamic, Dynamic> MatrixXd;
typedef Matrix<int, Dynamic, 1> VectorXi;
```

Storage Order in Eigen

- By default, Eigen stores the elemements in **column-major order**
- Algorithms that traverse a matrix row by row will go faster when the matrix is stored in row-major order because of better data locality. Similarly, column-by-column traversal is faster for column-major matrices.

```
The matrix A:

8 2 2 9
9 1 4 4
3 5 4 5

In memory (column-major):

8  9  3  2  1  5  2  4  4  9  4  5
```

Matrices Initialization

```
int main()
{
  Matrix3d m;

  double val = 1;
  // Access the individual coefficients
  for (int r = 0; r<3; r++)
    for (int c = 0; c<3; c++)
      m(r,c) = val++;

  cout << "m =" << endl << m << endl;

  // Comma initializer syntax
  m << 9, 8, 7, 6, 5, 4, 3, 2, 1;

  cout << "m =" << endl << m << endl;

  return 0;
}
```

Matrices Initialization

```
Matrix3f A;
Matrix4d B;

// Set each coefficient to a uniform random value in the range
A = Matrix3f :: Random();

// Set B to the identity matrix
B = Matrix4d :: Identity();

// Set all elements to zero
A = Matrix3f :: Zero();

// Set all elements to ones
A = Matrix3f :: Ones();

// Set all elements to a constant value
B = Matrix4d :: Constant(4.5) ;
```

Matrices Operations

```
Matrix4f M1 = Matrix4f :: Random () ;
Matrix4f M2 = Matrix4f :: Constant (2.2) ;

// Addition
// The size and the coefficient - types of the matrices must match
cout << M1 + M2 << endl ;

// Matrix multiplication
// The inner dimensions and the coefficient - types must match
cout << M1 * M2 << endl ;

// Scalar multiplication , and subtraction
cout << M2 - Matrix4f :: Ones() * 2.2 << endl ;

// two matrices are considered equal if all corresponding coefficients
// are equal.
cout << ( M2 - Matrix4f :: Ones() * 2.2 == Matrix4f :: Zero() )<< endl ;
```

Matrices Operations

```
// Transposition
cout << M1.transpose() << endl;

// Inversion ( # include < Eigen / Dense > )
// Generates NaNs if the matrix is not invertible
cout << M1.inverse() << endl;

// Square each element of the matrix
// The array() method 'converts' a Matrix into array expressions
// Array class provides an easy way to perform coefficient-wise operations,
cout << M1.array().square() << endl;

// Multiply two matrices element - wise
cout << M1.array() * Matrix4f :: Identity().array() << endl;

// All relational operators can be applied element - wise
cout << M1.array() <= M2.array() << endl << endl ;
cout << M1.array() > M2.array() << endl ;
```

Vectors Initialization

```
// Comma initialization
v << 1.0 f , 2.0 f , 3.0 f;
// Coefficient access
cout << v (2) << endl;
// Vectors of length up to four can be initialized in the constructor
Vector3f w (1.0 f , 2.0 f , 3.0 f );
// Utility functions
Vector3f v1 = Vector3f :: Ones ();
Vector3f v2 = Vector3f :: Zero ();
Vector4d v3 = Vector4d :: Random ();
Vector4d v4 = Vector4d :: Constant (1.8);
```

Vectors Operations

```
// Arithmetic operations
cout << v1 + v2 << endl << endl;
cout << v4 - v3 << endl;

// Scalar multiplication
cout << v4 * 2 << endl;
Vector4f v5 = Vector4f (1.0f , 2.0f , 3.0f , 4.0f );

// 4x4 * 4x1 - Works !
cout << Matrix4f :: Random() * v5 << endl;

// 4x1 * 4x4 - Compiler Error !
cout << v5 * Matrix4f :: Random() << endl;
```

Vectors Operations

```
// L2 norm
cout << v1.norm() << endl << endl;

// Dot product
cout << v1.dot( v2 ) << endl << endl;

// Normalization
cout << v1.normalized() << endl << endl;

// In place normalization
v1.normalize();

// Cross product and
cout << v1.cross( v2 ) << endl;

// Convert a vector to and from homogenous coordinates
Vector3f s = Vector3f :: Random();
Vector4f q = s.homogeneous();
cout << ( s == q.hnormalized() ) << endl;

// Element-wise operations
cout << v1.array() * v2.array() << endl << endl;
```

Block Operations and Casting

```
Eigen::MatrixXf m(4,4);

m <<  1, 2, 3, 4,
      5, 6, 7, 8,
      9,10,11,12,
     13,14,15,16;

cout << "Block in the middle" << endl;
cout << m.block<2,2>(1,1) << endl << endl;
for (int i = 1; i <= 3; ++i)
{
  cout << "Block of size " << i << "x" << i << endl;
  cout << m.block(0,0,i,i) << endl << endl;
}

// Bottom right 2x2 submat with casting
Eigen::Matrix2d  = m.block<2,2>(2,2).cast <double>();
```

```
Block in the middle
6  7
10 11

Block of size 1x1
1

Block of size 2x2
1 2
5 6

Block of size 3x3
 1  2  3
 5  6  7
 9 10 11
```

Interfacing with Raw Buffers

```
int array[8];
for(int i = 0; i < 8; ++i)
  array[i] = i;

cout << "Column-major:\n"
     << Map<Matrix<int,2,4> >(array) << endl
cout << "Row-major:\n"
     << Map<Matrix<int,2,4,RowMajor> >(array) << endl;
```

```
Column-major:
0 2 4 6
1 3 5 7
Row-major:
0 1 2 3
4 5 6 7
```

```
// From an OpenCV Mat to Eigen
// WARNING: OpenCV uses row-major order
cv::Mat_<float> cv_mat(2,2);
Matrix2f eigen_mat =
  Map< Matrix<float,2,2,RowMajor> >(
        reinterpret_cast<float *>(cv_mat.data));
```

Axis-angle Representation

```
Vector3f axis = Vector3f::Random();
Float angle = M_PI/6;

// Warning: The axis vector must be normalized.
Eigen::AngleAxis<float> ang_ax(angle, axis.normalized());

// Get the corresponding 3x3 rotation matrix
Eigen::Matrix3f r_mat = ang_ax.toRotationMatrix();

// Construct a rotation matrix from axis-angle
Matrix3f m = AngleAxisf(0.25*M_PI, Vector3f::UnitX());

// Construct the correspoinding axis-angle from a rotation matrix
Eigen::AngleAxis<float> res(r_mat*m);

// Set the correspoinding axis-angle from a rotation matrix
ang_ax.fromRotationMatrix(r_mat*m);
```

Using STL Containers with Eigen

- Using STL containers on fixed-size Eigen types requires the use of an over-aligned allocator

```
// For example, instead of (BUG)
std::vector<Eigen::Vector4f>

// You need to use
std::vector<Eigen::Vector4f,Eigen::aligned_allocator<Eigen::Vector4f> >
```

- After C++17 everything is taken care by the compiler and you can stop worrying

Further Reading

The Eigen Quick Reference Guide provides a handy reference to most matrix and vector operations:

https://eigen.tuxfamily.org/dox/group__QuickRefPage.html