Recitation13

CS100 TA Group

Outline

- Introduction to Eigen
    - What is Eigen?
    - Matrix template class in Eigen
    - Initialization
    - Matrix operations
        - Arithmetic operations
        - Block operations
    - Example
- Introduction to Concurrency

Introduction to Eigen

What is Eigen?

- Eigen is an open-source linear algebra library implemented in C++. It’s fast and well-suited for a wide range of tasks, from heavy numerical computation, to simple vector arithmetic.

Matrix template class in Eigen

- All matrices and vectors are objects of the Matrix template class, and Vectors are just a special case of matrices.
- You can create matrices with Eigen simply by:

```
Matrix3f A;
Matrix4d B;
```

where A is a 3×3 matrix of float numbers, while B is a 4×4 matrix of double numbers.

- Keep in mind, that not all combinations of size and type are re-defined by typedef in Eigen. For example, if you want to define a 5×5 matrix of short numbers, you require to use a slightly different syntax:

```
// 5x5 matrix of type short
Matrix<short, 5, 5> M1;
```

Initialization

- To initialize a matrix or vector variable, you can simply use comma initializer syntax:

```
// Initialize A
A << 1.0f, 0.0f, 0.0f,
     0.0f, 1.0f, 0.0f,
     0.0f, 0.0f, 1.0f;
```

- Besides, some utility functions can be used to initialize matrices to predefined values:

```
// Set each coefficient to a uniform random value in the range
   [-1, 1]
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();
```

Matrix Operations

- Arithmetic operations: common arithmetic operators are overloaded to work with matrices, such as addition, subtraction, multiplication and scalar division
- Transposition, inversion, square can also be implemented by using methods provided by Eigen efficiently

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

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

Matrix Operations

- Block operations: you will find that making operations on a sub-block of matrices or vectors is desired
  - `matrix.block<p,q>(i,j)` returns the block starting at `i,j` with size `p,q`
  - `matrix.row(i)` returns the i-th row of matrix
  - `matrix.col(i)` returns the i-th column of matrix
  - `vector.head(n)` returns the first n elemets of vector
  - `vector.tail(n)` returns the last n elements of vector
  - `vector.segment(i,n)` returns the segment containing n elements of vector with starting point at i

Examples(operations on Vector3d)

```
Eigen::Vector3d pos;
pos << 0.0,-2.0,1.0;
Eigen::Vector3d pos2 = Eigen::Vector3d(1,-3,0.5);
pos = pos+pos2;

Eigen::Vector3d pos3 =  Eigen::Vector3d::Zero();

pos.normalize();
pos.inverse();

//Dot and Cross Product
Eigen::Vector3d v(1, 2, 3);
Eigen::Vector3d w(0, 1, 2);

double vDotw = v.dot(w); // dot product of two vectors
Eigen::Vector3d vCrossw = v.cross(w); // cross product of two vectors
```

Examples(operations on Quaterniond)

```
Eigen::Quaterniond rot;

rot.setFromTwoVectors(Eigen::Vector3d(0,1,0), pos);

Eigen::Matrix<double,3,3> rotationMatrix;

rotationMatrix = rot.toRotationMatrix();

Eigen::Quaterniond q(2, 0, 1, -3);
std::cout << "This quaternion consists of a scalar " << q.w()
<< " and a vector " << std::endl << q.vec() << std::endl;

q.normalize();

std::cout << "To represent rotation, we need to normalize it such
that its length is " << q.norm() << std::endl;

Eigen::Vector3d vec(1, 2, -1);
Eigen::Quaterniond p;
p.w() = 0;
p.vec() = vec;
Eigen::Quaterniond rotatedP = q * p * q.inverse();
Eigen::Vector3d rotatedV = rotatedP.vec();
std::cout << "We can now use it to rotate a vector " << std::endl
<< vec << " to " << std::endl << rotatedV << std::endl;
```

```
// convert a quaternion to a 3x3 rotation matrix:
Eigen::Matrix3d R = q.toRotationMatrix();

std::cout << "Compare with the result using an rotation matrix "
<< std::endl << R * vec << std::endl;

Eigen::Quaterniond a = Eigen::Quaterniond::Identity();
Eigen::Quaterniond b = Eigen::Quaterniond::Identity();
Eigen::Quaterniond c;
// Adding two quaternion as two 4x1 vectors is not supported by the Eigen API.
//That is, c = a + b is not allowed. The solution is to add each element:

c.w() = a.w() + b.w();
c.x() = a.x() + b.x();
c.y() = a.y() + b.y();
c.z() = a.z() + b.z();
```

Examples(operations on MatrixXd)

```
//Transpose and inverse:
Eigen::MatrixXd A(3, 2);
A << 1, 2,
     2, 3,
     3, 4;

Eigen::MatrixXd B = A.transpose();// the transpose of A is a 2x3 matrix
// computer the inverse of BA, which is a 2x2 matrix:
Eigen::MatrixXd C = (B * A).inverse();
C.determinant(); //compute determinant
Eigen::Matrix3d D = Eigen::Matrix3d::Identity();

Eigen::Matrix3d m = Eigen::Matrix3d::Random();
m = (m + Eigen::Matrix3d::Constant(1.2)) * 50;
Eigen::Vector3d v2(1,2,3);

cout << "m =" << endl << m << endl;
cout << "m * v2 =" << endl << m * v2 << endl;
```

```
//Accessing matrices:
Eigen::MatrixXd A2 = Eigen::MatrixXd::Random(7, 9);
std::cout << "The fourth row and 7th column element is " << A2(3, 6) << std::endl;

Eigen::MatrixXd B2 = A2.block(1, 2, 3, 3);
std::cout << "Take sub-matrix whose upper left corner is A(1, 2)"
<< std::endl << B2 << std::endl;

Eigen::VectorXd a2 = A2.col(1); // take the second column of A
Eigen::VectorXd b2 = B2.row(0); // take the first row of B2

Eigen::VectorXd c2 = a2.head(3);// take the first three elements of a2
Eigen::VectorXd d2 = b2.tail(2);// take the last two elements of b2
```

Introduction to Concurrency

Today’s learning objectives

- Asynchronous tasks and threads
- Promises and tasks
- More on mutexes and condition variables
- More on std::call_once
- Example: Ping-Pong threads

Spawning asynchronous tasks

- Two ways: **`std::async`** and **`std::thread`**
- It's all about things that are callable:
  - Functions and Member functions.
  - Objects with **`operator()`** and Lambda functions

Hello World with `std::async`

```
#include <future> // for std::async
#include <iostream>

void write_message(std::string const& message) {
    std::cout<<message;
}

int main() {
    auto f = std::async(write_message,
          “hello world from std::async\n");
    write_message("hello world from main\n");
    f.wait();
}
```

Hello World with `std::thread`

```
#include <thread> // for std::thread
#include <iostream>
void write_message(std::string const& message) {
    std::cout<<message;
}

int main() {
    std::thread t(write_message,
                  "hello world from std::thread\n");
    write_message("hello world from main\n");
    t.join();
}
```

Missing `join` with `std::thread`

```
#include <thread>
#include <iostream>
void write_message(std::string const& message) {
    std::cout<<message;
}
int main() {
    std::thread t(write_message,
                  "hello world from std::thread\n");
    write_message("hello world from main\n");
    // oops no join
}
```

Missing `wait` with `std::async`

```
#include <future>
#include <iostream>
void write_message(std::string const& message) {
    std::cout<<message;
}
int main() {
    auto f = std::async(write_message,
                        "hello world from std::async\n");
    write_message("hello world from main\n");
    // oops no wait
}
```

Async Launch Policies

- The standard launch policies are the members of the `std::launch` scoped enum.
- They can be used individually or together.

Async Launch Policies

- `std::launch::async =>` "as if" in a new thread.
- `std::launch::deferred =>` executed on demand.
- `std::launch::async | std::launch::deferred =>`

  implementation chooses (default).

std::launch::async

```
#include <future>
#include <iostream>
#include <stdio.h>
void write_message(std::string const& message) {
    std::cout<<message;
}

int main() {
    auto f=std::async(
        std::launch::async, write_message,
         "hello world from std::async\n");
    write_message("hello world from main\n");
    getchar();
    f.wait();
}
```

std::launch::deferred

```
#include <future>
#include <iostream>
#include <stdio.h>
void write_message(std::string const& message) {
    std::cout<<message;
}

int main() {
    auto f=std::async(
        std::launch::deferred, write_message,
        "hello world from std::async\n");
    write_message("hello world from main\n");
    getchar();
    f.wait();
}
```

Returning values with `std::async`

```
#include <future>
#include <iostream>
int find_the_answer() {
    return 42;
}

int main() {
    auto f = std::async(find_the_answer);
    std::cout<<"the answer is "<<f.get()<<"\n";
}
```

Passing parameters

```
#include <future>
#include <iostream>
std::string copy_string(std::string const&s) {
    return s;
}

int main() {
    std::string s="hello";
    auto f=std::async(std::launch::deferred,
        copy_string,s);
    s="goodbye";
    std::cout<<f.get()<<" world!\n";
}
```

Passing parameters with `std::ref`

```
#include <future>
#include <iostream>
std::string copy_string(std::string const&s) {
    return s;
}

int main() {
    std::string s="hello";
    auto f=std::async(std::launch::deferred,
        copy_string,std::ref(s));
    s="goodbye";
    std::cout<<f.get()<<" world!\n";
}
```

Passing parameters with a lambda

```
std::string copy_string(std::string const&s) {
    return s;
}

int main() {
    std::string s="hello";
    auto f=std::async(std::launch::deferred,
        [&s](){return copy_string(s);});
    s="goodbye";
    std::cout<<f.get()<<" world!\n";
}
```

std::async passes exceptions

```
#include <future>
#include <iostream>
int find_the_answer() {
  throw std::runtime_error("Unable to find the answer”);
}

int main() {
  auto f=std::async(find_the_answer);
  try {
    std::cout<<"the answer is "<<f.get()<<"\n";
  }
  catch(std::runtime_error const& e) {
    std::cout<<"\nCaught exception: "<<e.what();
  }
}
```

Today’s learning objectives

- Asynchronous tasks and threads
- Promises and tasks
- More on mutexes and condition variables
- More on std::call_once
- Example: Ping-Pong threads

Locking multiple mutexes

```
class account {
    std::mutex m;
    currency_value balance;
public:
    friend void transfer( account& from,
                          account& to,
                          currency_value amount ) {
        std::lock_guard<std::mutex> lock_from(from.m);
        std::lock_guard<std::mutex> lock_to(to.m);
        from.balance -= amount;
        to.balance += amount;
    }
};
```

Locking multiple mutexes (II)

```
void transfer( account& from,
               account& to,
               currency_value amount) {
  std::lock(from.m,to.m);
  std::lock_guard<std::mutex> lock_from(
      from.m, std::adopt_lock);
  std::lock_guard<std::mutex> lock_to(
      to.m, std::adopt_lock);
  from.balance -= amount;
  to.balance += amount;
}
```

Waiting for events without futures

- Repeatedly poll in a loop (busy-wait)
- Wait using a condition variable

Synchronization between threads

- Apart from just protecting data, sometimes we may wish for one thread to **wait** until another thread has something done
- In C++:
  - Conditional variables
  - Futures

Example: Waiting for an item

- If all we've got is `try_pop()`, the only way to wait is to poll:

```
std::queue<my_class> the_queue;
std::mutex the_mutex;
void wait_and_pop(my_class& data){
  for(;;){
    std::lock_guard<std::mutex> guard(the_mutex);
    if(!the_queue.empty()) {
      data=the_queue.front();
      the_queue.pop();
      return;
    }
  }
}
```

- This is not ideal.

std::condition_variable

- A synchronization primitive that can be used to block a thread or multiple threads at the same time, until
  - A notification is received from another thread
  - A time-out expires

std::condition_variable

- A thread that intends to wait on std::condition_variable has to acquire a std::unique_lock first
- The wait operations atomically release the mutex and suspend the execution of the thread
- When the condition variable is notified, the thread is awakened, and the mutex is reacquired

Performing a blocking wait

- We want to wait for a particular condition to be true (there is an item in the queue).
- This is a job for **`std::condition_variable`**:

```
std::condition_variable the_cv;

void wait_and_pop(my_class& data) {
     std::unique_lock<std::mutex> lk(the_mutex);
     the_cv.wait(lk,
                 [](）
                 {return !the_queue.empty();});
     data = the_queue.front();
     the_queue.pop();
}
```

Signalling a waiting thread

- To signal a waiting thread, we need to notify the condition variable when we push an item on the queue:

```
void push(Data const& data)
{
    {
        std::lock_guard<std::mutex> lk(the_mutex);
        the_queue.push(data);
    }
    the_cv.notify_one();
}
```

Example

```
std::mutex mut;
std::queue<data_chunk> data_queue;
std::condition_variable data_cond;

void data_preparation_thread() {
  while( more_data_to_prepare() ) {
    data_chunk data = prepare_data();
    std::lock_guard<std::mutex> lk(mut);
    data_queue.push(data);
    data_cond.notify_one();
  }
}

void data_processing_thread() {
  while(true) {
    std::unique_lock<std::mutex> lk(mut);
    data_cond.wait(lk,[]{return !data_queue.empty();});
    data_chunk data = data_queue.front();
    data_queue.pop();
    lk.unlock();
    process(data);
    if(is_last_chunk(data))
      break;
  }
}
```

Mutex to protect resource

Example

```
std::mutex mut;
std::queue<data_chunk> data_queue;
std::condition_variable data_cond;

void data_preparation_thread() {
  while( more_data_to_prepare() ) {
    data_chunk data = prepare_data();
    std::lock_guard<std::mutex> lk(mut);
    data_queue.push(data);
    data_cond.notify_one();
  }
}

void data_processing_thread() {
  while(true) {
    std::unique_lock<std::mutex> lk(mut);
    data_cond.wait(lk,[]{return !data_queue.empty();});
    data_chunk data = data_queue.front();
    data_queue.pop();
    lk.unlock();
    process(data);
    if(is_last_chunk(data))
      break;
  }
}
```

Queue used to pass data

Example

When data is ready, thread locks mutex, pushes data, and calls notify_one()

```
std::mutex mut;
std::queue<data_chunk> data_queue;
std::condition_variable data_cond;

void data_preparation_thread() {
  while( more_data_to_prepare() ) {
    data_chunk data = prepare_data();
    std::lock_guard<std::mutex> lk(mut);
    data_queue.push(data);
    data_cond.notify_one();
  }
}

void data_processing_thread() {
  while(true) {
    std::unique_lock<std::mutex> lk(mut);
    data_cond.wait(lk,[]{return !data_queue.empty();});
    data_chunk data = data_queue.front();
    data_queue.pop();
    lk.unlock();
    process(data);
    if(is_last_chunk(data))
      break;
  }
}
```

Example

notify_one() notifies
The waiting thread

```
std::mutex mut;
std::queue<data_chunk> data_queue;
std::condition_variable data_cond;

void data_preparation_thread() {
  while( more_data_to_prepare() ) {
    data_chunk data = prepare_data();
    std::lock_guard<std::mutex> lk(mut);
    data_queue.push(data);
    data_cond.notify_one();
  }
}

void data_processing_thread() {
  while(true) {
    std::unique_lock<std::mutex> lk(mut);
    data_cond.wait(lk,[]{return !data_queue.empty();});
    data_chunk data = data_queue.front();
    data_queue.pop();
    lk.unlock();
    process(data);
    if(is_last_chunk(data))
      break;
  }
}
```

Example

```
std::mutex mut;
std::queue<data_chunk> data_queue;
std::condition_variable data_cond;

void data_preparation_thread() {
  while( more_data_to_prepare() ) {
    data_chunk data = prepare_data();
    std::lock_guard<std::mutex> lk(mut);
    data_queue.push(data);
    data_cond.notify_one();
  }
}

void data_processing_thread() {
  while(true) {
    std::unique_lock<std::mutex> lk(mut);
    data_cond.wait(lk,[]{return !data_queue.empty();});
    data_chunk data = data_queue.front();
    data_queue.pop();
    lk.unlock();
    process(data);
    if(is_last_chunk(data))
      break;
  }
}
```

Receiver thread puts itself into waiting mode through this call (if queue is empty). It will also release the lock

Example

```
std::mutex mut;
std::queue<data_chunk> data_queue;
std::condition_variable data_cond;

void data_preparation_thread() {
  while( more_data_to_prepare() ) {
    data_chunk data = prepare_data();
    std::lock_guard<std::mutex> lk(mut);
    data_queue.push(data);
    data_cond.notify_one();
  }
}

void data_processing_thread() {
  while(true) {
    std::unique_lock<std::mutex> lk(mut);
    data_cond.wait(lk,[]{return !data_queue.empty();});
    data_chunk data = data_queue.front();
    data_queue.pop();
    lk.unlock();
    process(data);
    if(is_last_chunk(data))
      break;
  }
}
```

It also passes a wake condition that will be checked upon notify_all()

Example

```
std::mutex mut;
std::queue<data_chunk> data_queue;
std::condition_variable data_cond;

void data_preparation_thread() {
  while( more_data_to_prepare() ) {
    data_chunk data = prepare_data();
    std::lock_guard<std::mutex> lk(mut);
    data_queue.push(data);
    data_cond.notify_one();
  }
}

void data_processing_thread() {
  while(true) {
    std::unique_lock<std::mutex> lk(mut);
    data_cond.wait(lk,[]{return !data_queue.empty();});
    data_chunk data = data_queue.front();
    data_queue.pop();
    lk.unlock();
    process(data);
    if(is_last_chunk(data))
      break;
  }
}
```

The mutex will be automatically locked once the wait terminates

Example

```
std::mutex mut;
std::queue<data_chunk> data_queue;
std::condition_variable data_cond;

void data_preparation_thread() {
  while( more_data_to_prepare() ) {
    data_chunk data = prepare_data();
    std::lock_guard<std::mutex> lk(mut);
    data_queue.push(data);
    data_cond.notify_one();
  }
}

void data_processing_thread() {
  while(true) {
    std::unique_lock<std::mutex> lk(mut);
    data_cond.wait(lk,[]{return !data_queue.empty();});
    data_chunk data = data_queue.front();
    data_queue.pop();
    lk.unlock();
    process(data);
    if(is_last_chunk(data))
      break;
  }
}
```

Lock only for as long as necessary

Today's learning objectives

- Asynchronous tasks and threads
- Promises and tasks
- More on mutexes and condition variables
- More on std::call_once
- Example: Ping-Pong threads

Example: Ping-Pong threads

```
#include "stdlib.h”
#include <string>
#include <thread>
#include <mutex>
#include <iostream>
#include <unistd.h>

bool onRightSide;
std::mutex mut;
std::condition_variable data_cond;
void player( bool isRightSidePlayer, std::string message ) {
  while(1) {
    std::unique_lock<std::mutex> lk(mut);
    data_cond.wait(lk,[&isRightSidePlayer]{
       return isRightSidePlayer == onRightSide;});
    std::cout << message << "\n";
    usleep(1000000);
    onRightSide = !onRightSide;
    lk.unlock();
    data_cond.notify_one();
  }
}
```

Example: Ping-Pong threads

```
int main() {
  onRightSide = true;
  std::thread leftPlayer( player, false, std::string("Pong") );
  std::thread rightPlayer( player, true, std::string("Ping") );
  leftPlayer.join();
  rightPlayer.join();
  return 0;
}
```