[Segmentation Fault in C++ (Eigen library) - Stack Overflow](https://stackoverflow.com/questions/51380447/segmentation-fault-in-c-eigen-library)

Copilot

**Sent by Copilot:**

Sure! Here’s how you can convert the given code to use unique\_ptr and make\_unique with Eigen:

**Include necessary headers:**

#include <Eigen/Dense>

#include <memory>

**Define the matrices and vectors using**unique\_ptr**:**

using MatrixXdPtr = std::unique\_ptr<Eigen::MatrixXd>;

using VectorXdPtr = std::unique\_ptr<Eigen::VectorXd>;

MatrixXdPtr H = std::make\_unique<Eigen::MatrixXd>(/\* initialize with your data \*/);

VectorXdPtr w = std::make\_unique<Eigen::VectorXd>(/\* initialize with your data \*/);

**Perform the SVD and solve using**unique\_ptr**:**

auto svd = std::make\_unique<Eigen::JacobiSVD<Eigen::MatrixXd>>(\*H, Eigen::ComputeThinU | Eigen::ComputeThinV);

VectorXdPtr X = std::make\_unique<Eigen::VectorXd>(svd->solve(\*w));

**Access the results:**

std::cout << "Solution X:\n" << \*X << std::endl;

Here’s the complete example:

#include <Eigen/Dense>

#include <memory>

#include <iostream>

using MatrixXdPtr = std::unique\_ptr<Eigen::MatrixXd>;

using VectorXdPtr = std::unique\_ptr<Eigen::VectorXd>;

int main() {

// Initialize H and w with your data

MatrixXdPtr H = std::make\_unique<Eigen::MatrixXd>(3, 2);

\*H << 1, 2,

3, 4,

5, 6;

VectorXdPtr w = std::make\_unique<Eigen::VectorXd>(3);

\*w << 7, 8, 9;

// Perform SVD and solve

auto svd = std::make\_unique<Eigen::JacobiSVD<Eigen::MatrixXd>>(\*H, Eigen::ComputeThinU | Eigen::ComputeThinV);

VectorXdPtr X = std::make\_unique<Eigen::VectorXd>(svd->solve(\*w));

// Access the results

std::cout << "Solution X:\n" << \*X << std::endl;

return 0;

}

This example demonstrates how to use unique\_ptr and make\_unique to manage the memory of Eigen matrices and vectors while performing SVD and solving the least squares problem. Adjust the initialization of H and w with your actual data as needed. If you have any further questions or run into issues, feel free to ask!

std::make\_unique is a utility function introduced in C++14 that simplifies the creation of unique\_ptr objects. It helps manage the lifetime of dynamically allocated objects by ensuring that memory is properly allocated and deallocated, reducing the risk of

memory leaks and errors.

Here’s what std::make\_unique does:

1. **Memory Allocation**: It allocates memory for an object on the heap.
2. **Object Construction**: It constructs the object in the allocated memory.
3. **Ownership Transfer**: It returns a unique\_ptr that takes ownership of the allocated memory.

Using std::make\_unique is preferred over manually creating a unique\_ptr with new because it provides a cleaner, safer, and more concise way to manage dynamic memory. Here’s a simple example:

#include <memory>

#include <iostream>

class MyClass {

public:

MyClass(int value) : value(value) {

std::cout << "MyClass constructed with value: " << value << std::endl;

}

~MyClass() {

std::cout << "MyClass destructed" << std::endl;

}

private:

int value;

};

int main() {

// Create a unique\_ptr to MyClass using std::make\_unique

std::unique\_ptr<MyClass> myPtr = std::make\_unique<MyClass>(42);

// Use the object through the unique\_ptr

// No need to manually delete the object, it's automatically managed

return 0;

}

In this example, std::make\_unique<MyClass>(42) creates a unique\_ptr that manages an instance of MyClass initialized with the value 42. [The memory is automatically cleaned up when the unique\_ptr goes out of scope1](https://en.cppreference.com/w/cpp/memory/unique_ptr/make_unique)[2](https://thispointer.com/what-is-the-make_unique-in-c-what-is-its-benefit/).

If you have any more questions or need further clarification, feel free to ask!