Eigen

Lucas Frérot

Table of contents

Here are the helper files for this section:

- CMakeLists.txt
- wave.cpp
- kinetic_energy.cpp
- animate.py
- plot.py

CMake

To use Eigen as a dependency, you can use CMake's `FetchContent` module:

```
include(FetchContent)
FetchContent_Declare(
  eigen
  GIT_REPOSITORY https://gitlab.com/libeigen/eigen.git
  GIT_TAG 5.0.1
  GIT_SHALLOW TRUE
)
FetchContent_MakeAvailable(eigen)
```

Then link your target against Eigen:

```
target_link_libraries(mytarget PRIVATE Eigen3::Eigen)
```

Kinetic energy

We want to compute, with Eigen arrays:

$$E_k = \sum_{i=1}^{N} \frac{1}{2} m_i v_i^2,$$

where $v_i^2 = \vec{v}_i \cdot \vec{v}_i = \|\vec{v}_i\|^2$. Let's start with an empty function taking as inputs a velocity array and masses array:

```
// Ref<EigenType> is how you pass a reference to an Eigen array/vector/matrix
double kinetic_energy(Ref<Array3Xd> velocities, Ref<ArrayXd> masses) {
  return 0;
}
```

You can observe that the type for `velocities` is particular: it has a fixed size of 3 rows times X columns (X being the number of atoms). Each column in `velocities` is therefore an atom's velocity vector. Since masses could be different for different atoms, we need to compute v_i in a *column wise* fashion. Let's define a `squared_velocity` array:

```
ArrayXd squared_velocities = velocities.colwise().squaredNorm();
```

That lets us define the energy per particle:

```
ArrayXd ek_per_particle = 0.5 * masses * squared_velocities;
```

And finally we can obtain the total kinetic energy with `ek_per_particle.sum()`.

Scalar wave propagation with Eigen

A very simple way to solve the scalar wave equation:

$$\frac{\partial^2 u}{\partial x^2} = \frac{\partial^2 u}{\partial t^2},$$

is to use a centered finite difference approximation for the space derivative (with x) and time derivative (with t). The latter leads to the velocity Verlet algorithm seen in class. The former follows the formula:

$$\frac{\partial^2 u}{\partial x^2}(x_i) \approx \frac{u(x_{i-1}) - 2u(x_i) + u(x_{i+1})}{\Delta x^2},$$

where Δx is the spacing between x_i and x_{i+1}. We now need boundary conditions to the wave equation. Here we select a fixed end $x = -L/2$ and free end at $x = L/2$, which translate to:

$$\partial_x^2 u(x_0) = 0,\ u(x_{N+1}) = u(x_{N-1})$$

This scheme, with the Verlet integration, can be entirely written without loops using Eigen operations like `head`, `segment` and `tail`, which we can see in wave.cpp.

You can plot (with Numpy and Matplotlib) the result of the simulation by piping the standard output to a plot script:

```
./wave | ../animate.py
```

Try benchmarking the `wave` program with and without the `std::cout` call in the time loop.

Eigen and C++11's `auto`

Eigen and the `auto` keyword (which lets the compiler infer the type of a variable) do not play well. For example, the following code fails to compile:

```
Array3Xd positions(3, 10);
auto r01_vector = positions.col(1) - positions.col(0);
auto r01 = r01_vector.norm();
```

The compilation error should look like:

```
error: ‘class Eigen::CwiseBinaryOp<Eigen::internal::scalar_difference_op<double, double>, co
1>, 3, 1, true>, const Eigen::Block<Eigen::Array<double, 3, -1>, 3, 1, true> >’ has no membe
```

The very long type name `Eigen::CwiseBinaryOp<...>` is actually the type that the compiler deduced for `r01_vector`: this is because when compilers try to deduce types they always go for the closest match to what they have. In this case, the type of the expression `positions.col(1) - positions.col(0)` is used for `r01_vector`. Due to Eigen's expression template system, the expression type does not resolve to an array or vector type, and we cannot call `.norm()` on it. Instead we should use the following code:

```
Array3Xd positions(3, 10);
Vector3d r01_vector = positions.col(1) - positions.col(0);
auto r01 = r01_vector.norm();
```

Here, because we manually specified the type for `r01_vector`, we can call `.norm()` on it because it is part of the `Vector3d` API. We do not need to specify the type on `r01` because we know `.norm()` here does not return an Eigen type but a `double`.

Note: Eigen's expression template system is a system that implements lazy evaluation of array operations. In our case, the difference between the two positions columns will be evaluated when `r01` is assigned, not when `r01_vector` is declared. This allows Eigen to find the optimal number of temporary objects needed for a computation, which can save both memory and computation time (at the expanse of compilation time to resolve the optimal operation order).