Assignment 5: Finite Element Method

Handout date: 29.05.2019
Submission deadline: 01.07.2019 at 23:59

In this exercise you will implement

- gradient descent and Newton's method,
- a spring system simulation,

1. Optimization

1.1. **Gradient Descent and Line Search.** Your task is to implement gradient descent and line search, and then test it on the Rosenbrock function. The Rosenbrock function is a banana-shaped function:

$$f(x,\ y) = (a - x)^2 + b(y - x^2)^2$$

In the file `optLib/RosenbrockFunction.h` the class `RosenbrockFunction` derives from `ObjectiveFunction`. You need to implement:

(1) the method `computeValue(const VectorXd& x)`, which returns $f(x)$,
(2) the method `addGradientTo(VectorXd& grad, const VectorXd& x)`, which adds the gradient $\nabla f(x)$ to grad.

In the file `optLib/GradientDescentFunctionMinimizer.h`, a skeleton of the gradient-descent algorithm is provided to you. The method `minimize(ObjectiveFunction* function, VectorXd& x)` returns true if a minimum of the function function was successfully found and writes the solution in x. You need to implement:

(3) the method `computeSearchDirection`, which writes the search direction in `dx`, and
(4) the method `doLineSearch`, which performs a line search on the function function in the direction `dx` and writes the solution to x.

Use the variable `maxLineSearchIterations` for the maximum number of line search iterations. Set the initial step length to $\alpha = 1$ and the scaling factor to $\beta = 0.5$.

Report. Test the gradient descent method on the Rosenbrock function by compiling and executing `test_rosenbrock.cpp`. Report the number of iterations it took to converge, the solution $x_{\min}$ and the minimum value $f(x_{\min})$. (This will all be reported to the console.)

1.2. **Newton's method.** As you probably noticed, gradient descent converges rather slowly. To improve performance, you will implement Newton's method and test it on the Rosenbrock function. Newton's method computes the search direction using second order information , stored in the Hessian. To get the Hessian of the Rosenbrock function , you need to:

(1) implement the method `addHessianEntriesTo` in the class `RosenbrockFunction`, which adds its second order derivates as `Eigen::Triplets` to `hessianEntries`. The class `NewtonFunctionMinimizer` (located at `optLib/NewtonFunctionMinimizer.h`) overloads the method `computeSearchDirection(...)` from the class `GradientDescentFunctionMinimizer` used in the previous two tasks.
(2) Implement the method `computeSearchDirection`, which computes the search direction `dx` given an objective function function and a current state `x`. Use the solver `Eigen::SimplicialLDLT` to solve for the search direction.

Report. In `test_rosenbrock.cpp` line 26, change the 0 to 1, compile and run. Report the number of iterations, the solution $x_{\min}$ and the minimum $f(x_{\min})$.

2. Spring system simulation

Let's first introduce the spring simulation. Our spring system consists of:

- n point masses, each spring $i \in [1,\ n]$ with mass m_i, rest/undeformed position X_i and current/deformed position x_i.
- springs, where each spring connects two point masses i and j, has a spring stiffness k, rest length $L = ||X_i - X_j||$ and current length $l = ||x_i - x_j||$.

We collect all point masses, rest positions and current positions in the vectors $\mathrm{m} = \{m_1,\ m_n\}$, $\mathrm{X} = \{X_1,\ X_n\}$ and $x = \{x_1,\ x_n\}$. The springs are modeled using Hooke's law. It states, that the force needed to change the length of a spring k scales linearly: $f_j(x) = -k(\frac{l}{L} - 1)\frac{x_i - x_j}{||x_i - x_j||}$. Static equilibrium conditions require that all forces sum up to zero: $f_{\text{int}} + f_{\text{ext}} = 0$, where $f_{\text{int}} = \sum_{k=1} f_k$ is the sum over all internal forces and f_{ext} are external forces, such as gravitational forces. We can solve this equation by transforming it into an energy minimization problem $\min_x E_{\text{ext}} + E_{\text{int}}$, where the total internal energy $E_{\text{int}} = \sum_k^N E_k$ is the sum of all spring energies $E_k = \frac{1}{2}k(\frac{l}{L} - 1)^2 L$. We will solve this energy minimization problem first with the Gradient Descent method and then with Newton's method. Before using either of the methods to simulate a mass-spring system, you will first verify both of the optimization methods on the Rosenbrock function.

2.1. **Spring simulation with gradient descent.** Now that we have a working gradient-descent algorithm, we can use it to simulate a mass-spring system. The objective function we want to minimize is the total potential energy, located in `TotalEnergyFunction`. The class `TotalEnergyFunction` derives from `ObjectiveFunction`, thus we can use it in our gradient-descent implementation. It is responsible for summing up the energy and gradient of all energy elements (class Element), which in this case are springs. Your task is to implement the computation of the spring energy and gradient. The class Spring in the file `femLib/Spring.h` is derived from the base class Element and connects two nodes (= mass points). To get the current position of node i, you can use the function `getNodePos(i,`

`x)` and `getNodePos(i, X)` to get the rest position, respectively. To get the global index of node i, use the function `getNodeIndex(int i)`. The spring stiffness is stored in the member k.

You need to implement:

(1) the method `getEnergy(const VectorXd& x, const VectorXd& X)`, which returns the spring energy given the current state `x` and rest state `X`,
(2) the method `addEnergyGradientTo(const VectorXd& x, const VectorXd& X, VectorXd& grad)`, which adds the spring energy gradient to grad given `x` and `X`.

Report. Test your implementation by compiling and running `main.cpp`. Make sure "Gradient Descent" is chosen as optimization strategy and "Spring" as element type. Hit "Test" and report the total energy, number of iterations and minimum and maximum energy values, plus a screenshot of the deformed mesh - this will take a few seconds.

2.2. **Spring simulation with Newton's method.** With Newton's method in place, we can make the mass-spring simulation a lot faster. Implement the method `addHessianEntriesTo` in the class Spring, which adds the second order derivatives of the spring energy to the Hessian (`hesEntries`).

Report. Test your implementation by compiling and running `main.cpp`. Make sure "Newton's method" is chosen as optimization strategy and "Spring" as element type. Hit "Test" and report the total energy, number of iterations and minimum and maximum energy values, plus a screenshot of the deformed mesh

3. FEM Simulation

The finite element method provides a more sophisticated simulation framework, especially when coupled with a non-linear material model , such as the Neo-Hookean material model. In this exercise, you will implement the key components of a static Neo-Hookean FEM simulation. The Neo-Hookean material model in two dimensions defines the energy density as:

$$\Psi_{\mathrm{NH}} = \frac{\mu}{2}(tr(\mathrm{C}) - 2) - \mu \ln(J) + \frac{\lambda}{2}\ln(J)^2$$

where $\mathrm{C} = \mathrm{F}^T\mathrm{F}$ is the right Cauchy deformation tensor, $J = \det(\mathrm{F})$, and λ are the shear and bulk modulus. F is the deformation gradient.

3.1. **FEM simulation with gradient descent.** In this task you are going to simulate an elastic two-dimensional body using the Neo-Hookean material model and gradient descent. Because computing the gradient of an Neo-Hookean energy is rather tedious, we will approximate it using finite differences. As with the spring, the `FEM` element `femLib/FEMElement.h` derives from the Element base class. In the class `FEMElementFD`, implement the following:

(1) the function `getEnergy`, which returns the Neo-Hookean deformation energy. You can use the function `computeDeformationGradient`, which writes the deformation gradient F into `F` given an array `x`, which contains the coordinates of the three nodes. The shear modulus μ and bulk modulus λ are stored in the members `shearModulus` and `bulkModulus`. The area of the

element in undeformed state is stored in `restAreaShape`. To approximate the gradient of the Neo-hookean energy, we will use a centered finite difference scheme.

(2) In the class `FEMElementFD`, implement the function `addEnergyGradientTo`, which computes the local gradient of the element using finite differences and adds it to the global gradient grad.

Hint. An issue which didn't occur using springs, is that of inverted elements. One way to deal with this issue, is to add a safe-guard to the line search. In the line search algorithm, when searching for the step length that will decrease the total energy, also check if the new energy value is finite, e.g. with `std::isfinite`, and continue searching, if it is not.

Report. Test your implementation by compiling and running `main.cpp`. Choose "FEM FD" as the element type and make sure "Gradient Descent" is selected as the optimization strategy. To play around with the simulation click "Play". You can pin nodes and apply forces by changing the mouse mode. For the report:

(1) Run a fresh instance of the simulation,
(2) choose "FEM FD" as the element type and make sure "Gradient Descent" is selected as the optimization strategy
(3) click "Test" (this will take some time)
(4) and report the total energy, the number of iterations and the minimum and maximum deformation energy, as well as a screenshot of the deformed mesh.

3.2. **FEM simulation with Newton's method.** As with the spring simulation , Newton's method will hopefully give a performance boost. To use Newton's method , we need to compute the Hessian. Compute the Hessian $\nabla^2 E$ of the Neo-Hookean energy in the function `addEnergyHessianTo` in the class `FEMElementFD`.

Report. Test your implementation by compiling and running `main.cpp`. Choose "FEM FD" as the element type and make sure "Newton's Method" is selected as the optimization strategy. To play around with the simulation click "Play". You can pin nodes and apply forces by changing the mouse mode. For the report:

(1) Run a fresh instance of the simulation,
(2) choose "FEM FD" as the element type and make sure "Newton's Method" is selected as the optimization strategy
(3) click "Test" (this should be faster now)
(4) and report the total energy, the number of iterations and the minimum and maximum deformation energy, as well as a screenshot of the deformed mesh.

3.3. **Exact derivatives and Automatic Differentiation (Bonus).** Finite difference approximations allow us to compute derivatives without finding the analytical gradient or Hessian. While easy to implement, analytical solutions of gradients and Hessians are not only faster, but also more accurate. For the Neo-Hookean energy:

(1) compute the exact gradient and Hessian using Automatic Differentiation and put the result in the class `FEMElementAutoDiff`.

(2) compute the exact gradient and Hessian by deriving the analytic solution by hand and use it in the class `FEMElementAnalytic`.

Report.. Mention in the report if you implemented (1) and/or (2). For (2) , report the analytical solution for the gradient and the Hessian.

Hint. You can always check and debug derivatives with finite differences!