Assignment 1

Unconstrained Optimization & IK

Please fill in your name and student id:

First name:	Last name:	Student ID:

For a more readable version of this document, see Readme.pdf.

Deadline: March 12th 2020, 12pm. Only commits ***pushed*** to github before this date and time will be considered for grading.

Issues and questions: Please use the issue tracker of the main repository if you have any general issues or questions about this assignment. You can also use the issue tracker of your personal repository or email Moritz.

Assignment

In this assignment, we will implement derivative-based numerical methods to solve unconstrained optimization problems (Part 1). We will then use the implemented methods to solve Inverse Kinematics for a two-link kinematic chain, aka. the two-bar linkage (Part 2).

Part 1 - Unconstrained Optimization

We will implement and compare various strategies to solve the unconstrained minimization problem

$$x = \text{argmin}_{\tilde{x}} f(\tilde{x}).$$

To test your implementations, you can run the "opt-app".

1.1 Random minimization

A naive way to find a minimum of $f(x)$ is to randomly sample the objective function $f(x)$ over a prescribed search domain.

Task: Implement an optimization routine, that samples a search domain Ω_s and saves the best candidate x_B and its corresponding function value $f(x_B)$.

Relevant code:

In the file `src/optLib/RandomMinimzier.h`, implement the method `minimize(...)` of class `RandomMinimizer`.

The members `searchDomainMin/Max` define the minimum and maximum values of the search domain, $\Omega_s = [\text{searchDomainMin}, \text{searchDomainMax},]$, for each dimension. E.g. `searchDomainMin[i]` is the lower bound of the search domain for dimension i. The variables `xBest` and `fBest` should store the best candidate x_B and function value $f(x_B)$ that has been found so far.

The method runs in a for loop. To compare it to other optimization techniques, `iterations` has been set to 1.

Hints: [`std::uniform_real_distribution`] can generate random numbers.

1.2 Gradient Descent

Task: Implement Gradient Descent with fixed step size and with variable step size using the Line Search method.

Relevant code:

The file `src/optLib/GradientDescentMinimizer.h` contains three classes:

- `GradientDescentFixedStep`: implement the method `step(...)`. It shall update `x` to take a step of size `stepSize` in the direction of the search direction `dx`. The search direction was previously computed in the method `computeSearchDirection(...)`, which calls `obj->addGradient(...)` to compute $\nabla f(x)$.
- `GradientDescentLineSearch`: implement the method `step(...)`. Instead of fixed step size, use the Line Search method to iteratively find the best step size. `maxLineSearchIterations` defines the maximum number of iterations to perform.

1.3 Newton's method

Task: Implement Newton's method with global Hessian regularization.

Relevant code:

In the file `src/optLib/NewtonFunctionMinimizer.h` , implement the method `computeSearchDirection(...)` to compute the search direction according to Newton's method. The `ObjectiveFunction` has a method `getHessian(...)` that computes the Hessian.

Hint: The Eigen library provides various solvers to solve a linear system $Ax = b$, where A is a sparse matrix. Take a look at the example in the documentation here. Thus, to solve $Ax = b$, you could use e.g. this:

```
Eigen::SimplicialLDLT<SparseMatrix<double>, Eigen::Lower> solver;
solver.compute(A);
x = solver.solve(b);
```

Part 2 - Inverse Kinematics

In the second part, we want to use the numerical methods implemented in Part 1 to solve the IK problem for a two-bar linkage.

2.1 Forward Kinematics and Objective

Task: Implement forward kinematics and the objective function $f(x) = \frac{1}{2}(e(x) - x_t)^T(e(x) - x_t)$, where x are the joint angles, $e(x)$ computes the end-effector position (end-position of 2nd bar) given joint angles and x_t the target position.

Relevant code:

Forward Kinematics: In the file `src/app/Linkage.h`, implement the method `Linakge::forwardKinematics`. It takes as input the joint angles and outputs the end-positions of the links. For two bars it thus returns three end-positions.

Objective function: In the same file, implement the method `InverseKinematics::evaluate`. The input vector `x` are the joint angles. The method should compute $f(x)$ and return it.

Once you've implemented the above methods, you can run the "ik-app" and choose joint angles by clicking on the left side of the app, and change the target position x_t by clicking on the right side of the app.

2.2 Jacobian and Gradient

Task: Implement the Jacobian $J = \frac{\partial e}{\partial x}$ and the gradient $\nabla_x f$.

Relevant code:

Jacobian: Implement the method `Linkage::dfk_dangles`, which computes the Jacobian J. To test your implementation with finite differences, run the app `test-app`. The first test should pass!

Gradient: Implement the method `InverseKinematics::addGradientTo`, which computes the gradient and adds it to `grad`. Again, you can test your implementation with the "test-app: the second test should pass. (Hint: make use of the Jacobian!)

Once the above is implemented, you can solve IK with Gradient Descent. Run the "ik-app" and hit Space.

2.3 Jacobian derivative and Hessian

Task: Implement the $\frac{\partial J}{\partial x}$ and the Hessian $\nabla_x^2 f$.

Relevant code:

Jacobian derivative: Implement the method `Linkage::ddfk_ddangles`. It computes the derivative of the Jacobian and returns it as a Tensor, which in the code is an `std::array` of two `Matrix2d`, where `tensor[i][j]` corresponds to $\frac{\partial^2 e}{\partial x_i \partial x_j}$.

Hessian: Implement the method `InverseKinematics::hessian`, which returns the Hessian $\nabla_x^2 f$.

Once the above is implemented, you can solve IK with Newton's method. Run the "ik-app", choose "Newton's method" and hit Space.

Setting things up

Prerequisites

Make sure you install the following:

- Git (https://git-scm.com)
    - Windows: download installer from website
    - Linux: e.g. `sudo apt install git`
    - MacOS: e.g. `brew install git`
- CMake (https://cmake.org/)
    - Windows: download installer from website
    - Linux: e.g. `sudo apt install cmake`
    - MacOS: e.g. `brew install cmake`

Building the code

On Windows, you can use Git Bash to perform the steps mentioned below.

> Note: There seems to be a github classroom bug, where the starter code is not imported. Thus, cancel the import process and import the code yourself, like so:
> After step 1:
>
> `cd comp-fab-a0-XXX`
>
> `git pull https://github.com/computational-robotics-lab/comp-fab-a0`
>
> `git submodule update --init --recursive`

1. Clone this repository and load submodules:

   ```
   git clone --recurse-submodules YOUR_GIT_URL
   ```

2. Make build folder and run cmake

   ```
   cd comp-fab-a0-XXX
   mkdir build && cd build
   cmake ..
   ```

3. Compile code and run executable
   - for MacOS and Linux:

     ```
     make
     ./src/app/app
     ```

   - for Windows:
     - open `assignement0.sln` in Visual Studio
     - in the project explorer, right-click target "app" and set as startup app.
     - Hit F5!

Some comments

- If you are new to git, there are many tutorials online, e.g. http://rogerdudler.github.io/git-guide/.
- You will not have to edit any CMake file, so no real understanding of CMake is required. You might however want to generate build files for your favorite editor/IDE: https://cmake.org/cmake/help/latest/manual/cmake-generators.7.html