A complete Newton solver using Eigen

Alberto Artoni

April 05, 2024

Newton solver

This example (an extended version of `Examples/src/NewtonSolver`) is about a set of tools that implement generic Newton or quasi-Newton methods to determine the zero of scalar non-linear equations, as well as vector systems using the `Eigen` library.

Newton solver

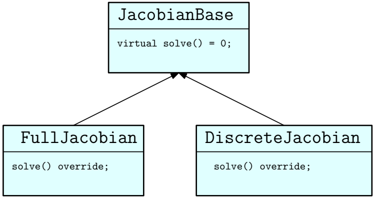
The code structure is the following:

- `NewtonTraits` contains the definition of the types used by the main classes, to guarantee uniformity.
- `JacobianBase` is a base class which implements the action of a *quasi-Jacobian*: the user may choose among `FullJacobian` where the actual Jacobian must be specified by the user, and `DiscreteJacobian`, that approximates the Jacobian via finite differences.
- `JacobianFactory` instantiates a concrete derived class of `JacobianBase` family on the fly.
- `Newton` applies the Newton method, given the non-linear system and a `JacobianBase`.
- `NewtonOptions` and `NewtonResults` bind the input options and the output results.

Abstract Class

`JacobianBase` is a base class which implements the action of a *quasi-Jacobian*:

- `FullJacobian`: the Jacobian must be specified by the user.
- `DiscreteJacobian`: the Jacobian is approximated via finite differences.



Smart Pointers

```
Newton(T::NonLinearSystemType &&     system_,
 std::unique_ptr<JacobianBase> jac_,
       const NewtonOptions &  options_ = NewtonOptions())
: system(std::forward<T::NonLinearSystemType>(system_)),
 jac(std::move(jac_)),  options(options_){}
```

The `Newton` class is initialized with a smart pointer to the `JacobianBase` class. Recall that since is an abstract class, we cannot instantiate an object. Hence we must use pointers. Move semantic will be detailed later from the professor.

Exercise

Consider the problem

$$\mathbf{f}(x, y) = \begin{bmatrix} (x-1)^2 + 0.1(y-5)^2 \\ 1.5 - x - 0.1y \end{bmatrix} = \mathbf{0}.$$

Starting from the provided solution sketch:

1. Implement the `NewtonTraits` class defining common types for homogeneity.
2. Implement the `FullJacobian` class (inheriting from `JacobianBase`) which, provided the full Jacobian matrix, solves the linear system using a direct solver with LU factorization.
3. `DiscreteJacobian` (inheriting from `JacobianBase`) which approximates the system Jacobian using finite differences and solves the linear system using a direct solver with LU factorization.
4. Implement a `JacobianFactory` method, returning an istance of `FullJacobian` or `DiscreteJacobian` depending on a parameter chosen by the user.
5. Solve the problem above using both the full and the discrete approach.