NumCSE

Autumn Semester 2017

Prof. Rima Alaifari

Exercise sheet 11
Iterative Methods for Non-Linear Systems of Equations

P. Bansal

Submission deadline - 20th December.

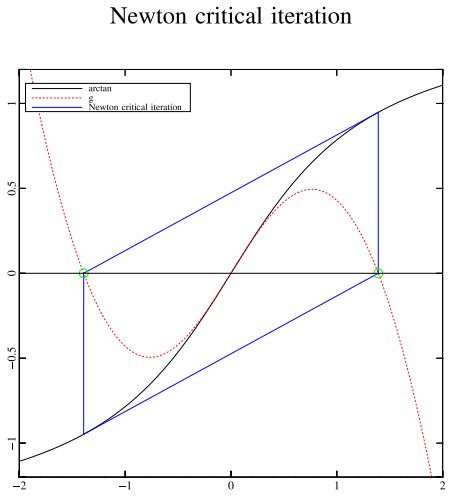
Problem 11.1: Newton's method for $F(x) := \arctan x$

The merely local convergence of Newton's method can be problematic. In Example 8.4.54 in the lecture notes, the consequences of this local convergence are discussed. One of them is the possible failure to converge because of the Newton correction overshoot. We look at this issue in this problem.

(a) Find an equation satisfied by the smallest positive initial guess $x^{(0)}$ for which Newton's method **does not converge** when applied to $F(x) = \arctan x$.

Hint: Find out when the Newton method oscillates between two values, consider the function graph to get insight.

SOLUTION:



The function $\arctan(x)$ is **positive, increasing and concave** for positive x. This implies that the first iterations of Newton's method with initial points $0 < x^{(0)} < y^{(0)}$ satisfy $y^{(1)} < x^{(1)} < 0$, visualise through the figure above. The function is **odd**, i.e. $\arctan(-x) = -\arctan(x)\ \forall\ x \in \mathbb{R}$. So, analogously for initial negative values $y^{(0)} < x^{(0)} < 0$ gives $0 < x^{(1)} < y^{(1)}$. Moreover, opposite initial values give opposite iterations: if $y^{(0)} = -x^{(0)}$ then $y^{(n)} = -x^{(n)}$ for every $n \in \mathbb{N}$.

All these facts imply that, if $|x^{(1)}| < |x^{(0)}|$, then the absolute values of the following iterations will converge monotonically to zero. Vice versa, if $|x^{(1)}| > |x^{(0)}|$, then the absolute values of the Newton's iterations will diverge monotonically. Moreover, the iterations change sign at each step, i.e., $x^{(n)} \cdot x^{(n+1)} < 0$.

It follows that the smallest positive initial guess $x^{(0)}$ for which Newton's method does not converge satisfies $x^{(1)} = -x^{(0)}$. This can be written as $x^{(1)} = x^{(0)} - \frac{f(x^{(0)})}{f'(x^{(0)})} = x^{(0)} - (1 + (x^{(0)})^2)\arctan x^{(0)} = -x^{(0)}$. Therefore, $x^{(0)}$ is a zero of the function $g(x) = 2x - (1 + x^2)\arctan x$ with $g'(x) = 1 - 2x\arctan x$.

(b) Implement a C++ function:

```
double newton_arctan(double x0_);
```

which uses Newton's method to find an approximation of such $x^{(0)}$. Here `x0_` is the initial guess.

SOLUTION:

Newton's iteration to find the smallest positive initial guess reads

$$x^{(n+1)} = x^{(n)} - \frac{2x^{(n)} - (1 + (x^{(n)})^2) \arctan x^{(n)}}{1 - 2x \arctan x^{(n)}} .$$

C++-code 11.1: Solution of (b)

```
2  double newton_arctan(double x0_) {
3
4      double x0 = x0_;
5      double upd = 1;
6      double eps = std::numeric_limits<double>::epsilon();
7
8      while(std::abs(upd) > eps) {
9
10         double x1 = x0 - (2*x0-(1+x0*x0)*std::atan(x0)) /
               (1-2*x0*std::atan(x0));
11         upd = (x1 - x0) / x1;
12         x0 =  x1;
13
14         std::cout << "x0 = " << x1 << ", accuracy = " << upd <<
               std::endl;
15     }
16
17     return x0;
18 }
```

Problem 11.2: Modified Newton method

In this problem, we look at a modified version of the Newton method.

Given $\mathbf{F} : \mathbb{R}^n \to \mathbb{R}^n$, such that $\mathcal{J}_\mathbf{F}(\mathbf{x}^{(0)})$ is regular, the non-linear system of equations $\mathbf{F}(\mathbf{x}) = \mathbf{0}$ can be solved using the following iterative method:

$$\begin{aligned}\mathbf{y}^{(k)} &= \mathbf{x}^{(k)} + \mathcal{J}_\mathbf{F}^{-1}(\mathbf{x}^{(k)})\ \mathbf{F}(\mathbf{x}^{(k)})\ ,\\ \mathbf{x}^{(k+1)} &= \mathbf{y}^{(k)} - \mathcal{J}_\mathbf{F}^{-1}(\mathbf{x}^{(k)})\ \mathbf{F}(\mathbf{y}^{(k)})\ ,\end{aligned}$$

where $\mathcal{J}_\mathbf{F}(\mathbf{x}) \in \mathbb{R}^{n,n}$ is the Jacobian matrix of $\mathbf{F}$ evaluated at $\mathbf{x}$.

(a) Show that the iteration is consistent with $\mathbf{F}(\mathbf{x}) = \mathbf{0}$, i.e. show that $\mathbf{x}^{(k)} = \mathbf{x}^{(0)}$, $\forall k \in \mathbb{N}$, if and only if $\mathbf{F}(\mathbf{x}^{(0)}) = \mathbf{0}$.

SOLUTION:

If $\mathbf{F}(\mathbf{x}^{(k)}) = \mathbf{0}$ then $\mathbf{y}^{(k)} = \mathbf{x}^{(k)} + \mathbf{0} = \mathbf{x}^{(k)}$ and $\mathbf{x}^{(k+1)} = \mathbf{y}^{(k)} - \mathbf{0} = \mathbf{x}^{(k)}$.
So, by induction, if $\mathbf{F}(\mathbf{x}^{(0)}) = \mathbf{0}$ then $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} = \mathbf{x}^{(0)}$ for every k.

Conversely, if $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)}$, then, by the recursion of the Newton method:

$$\mathbf{F}(x^{(k)}) = \mathbf{F}(x^{(k)} + \mathcal{J}_\mathbf{F}^{-1}(\mathbf{x}^{(k)})\ \mathbf{F}(\mathbf{x}^{(k)}))$$

As $\mathcal{J}_\mathbf{F}(\mathbf{x}^{(0)})$ is regular, then $\mathbf{F}(x)$ is injective for $|\mathbf{x} - \mathbf{x}^{(0)}| < \epsilon,\ \epsilon > 0$.

$$\begin{aligned}&\Rightarrow \mathcal{J}_\mathbf{F}^{-1}(\mathbf{x}^{(k)})\ \mathbf{F}(\mathbf{x}^{(k)}) = 0\\ &\Rightarrow \mathbf{F}(\mathbf{x}^{(k)}) = 0.\end{aligned}$$

(b) Implement a C++ function

```
template <typename Scalar, class Function, class Jacobian>
Scalar mod_newt_step_scalar(const Scalar& x,
                            const Function& f,
                            const Jacobian& df);
```

that computes a step of the modified Newton method for a *scalar* function $\mathbf{F}$, that is, for the case $n = 1$.

Here, `f` is a function object of type `Function` passing the function $F : \mathbb{R} \mapsto \mathbb{R}$ and `df` a function object of type `Jacobian` passing the derivative $F' : \mathbb{R} \mapsto \mathbb{R}$. Both require an appropriate lambda function.

SOLUTION:

C++11-code 11.2: Implementation of `mod_newt_step`.

```
2 template <typename Scalar, class Function, class Jacobian>
3 Scalar mod_newt_step_scalar(const Scalar& x,
4                             const Function& f,
5                             const Jacobian& df) {
```

```
    Scalar y = x + f(x) / df(x);
    return y - f(y) / df(x);
}
```

(c) Implement a C++ function `void mod_newt_ord()` which uses the function `mod_newt_step` and a good termination criteria to solve:

$$\arctan(x) - 0.123 = 0\,;$$

Use $x_0 = 5$ as initial guess. Determine empirically the order of convergence.

SOLUTION:

We implement a generic function that performs as many steps of a generic iteration and terminates if the error is sufficiently small, or if the maximum number of iterations has been reached. To this function, we pass the stepping function, the error function, the intial guess, and termination parameters.

C++11-code 11.3: Implementation of `sample_nonlinear_solver`.

```
template <class StepFunction, class Vector, class ErrorFunction>
bool sample_nonlinear_solver(const StepFunction& step,
                             Vector & x,
                             const ErrorFunction& errf,
                             double eps = 1e-8, int max_itr = 100) {
    // Temporary where to store new step
    Vector x_new = x;
    double r = 1;

    for(int itr = 0; itr < max_itr; ++itr) {

        r = errf(x);// Compute error (or residual)
        std::cout<<"[Step " <<itr<< "] Error: "<< r <<std::endl;

        x_new = step(x);// Advance to next step, x_new becomes x_{k+1}

        // Termination criteria
        if (r < eps * norm(x)) {
            std::cout << "[CONVERGED] in " << itr << " it. due to
                err. err = "
                      << r << " < " << eps << "." << std::endl;
            return true;
        }
        x = x_new;
    }

    // If max itr reached
    std::cout << "[NOT CONVERGED] due to MAX it. = " << max_itr
              << " reached, err = " << r << "." << std::endl;
    return false;
```

```
}
```

C++11-code 11.4: Implementation of `mod_newt_ord`.

```
void mod_newt_ord() {
    // Setting up values, functions and jacobian
    const double a = 0.123;
    auto f = [&a] (double x) { return atan(x) - a; };
    auto df = [] (double x) { return 1. / (x*x+1.); };

    const double x_ex = tan(a); // Exact solution

    // Store solution and error at each time step
    std::vector<double> sol;
    std::vector<double> err;

    // Compute error and push back to err, used in
    // general_nonlinear_solver as breaking condition errf(x) < eps
    auto errf = [&err, x_ex] (double & x) {
        double e =  std::abs(x - x_ex);
        err.push_back(e);
        return e;
    };

    // Perform convergence study with Modified newton for scalar
    std::cout << std::endl
              << "*** Modified Newton method (scalar) ***"
              << std::endl << std::endl;
    std::cout << "Exact: " << x_ex << std::endl;

    // Initial guess and compute initial error
    double x_scalar = 5;
    sol.push_back(x_scalar);
    errf(x_scalar);

    // Lambda performing the next step, used to define a proper
    // function handle to be passed to general_nonlinear_solver
    auto newt_scalar_step = [&sol, &f, &df] (double x) -> double {
        double x_new = mod_newt_step_scalar(x, f, df);
        sol.push_back(x_new);
        return x_new;
    };

    // Actually perform the solution
    sample_nonlinear_solver(newt_scalar_step, x_scalar, errf);

    // Print solution (final)
    std::cout << std::endl
              << "x^*_scalar = " << x_scalar
```

```
47                  << std::endl;
48
49      // Print table of solutions, errors and EOOC
50      auto space = std::setw(15);
51      std::cout << space << "sol."
52                << space << "err."
53                << space << "order"
54                << std::endl;
55      for(unsigned i = 0; i < sol.size(); ++ i) {
56          std::cout << space << sol.at(i)
57                    << space << err.at(i);
58          if(i >= 3) {
59              std::cout << space << (log(err.at(i)) -
                    log(err.at(i-1))) / (log(err.at(i-1)) -
                    log(err.at(i-2)));
60          }
61          std::cout << std::endl;
62      }
63  }
```

The order of convergence is approximately 3:

solution	error	order
5	4.87638	
0.560692	4.87638	
0.155197	0.437068	
0.123644	0.0315725	1.08944
0.123624	2.00333e-05	2.80183
0.123624	5.19029e-15	2.99809

Problem 11.3: Solving a quasi-linear system

In this problem, we implement a simple fixed point iteration and Newton's method to solve a so-called quasi-linear system of equations. **Template:** `quasilin.cpp`

Consider the *nonlinear* (quasi-linear) system:

$$\mathbf{A}(\mathbf{x})\mathbf{x} = \mathbf{b}\,, \tag{11.1}$$

where $\mathbf{b} \in \mathbb{R}^n$ and $\mathbf{A} : \mathbb{R}^n \to \mathbb{R}^{n,n}$ is a matrix-valued function:

$$\mathbf{A}(\mathbf{x}) := \begin{bmatrix} \gamma(\mathbf{x}) & 1 & & & & \\ 1 & \gamma(\mathbf{x}) & 1 & & & \\ & \ddots & \ddots & \ddots & & \\ & & \ddots & \ddots & \ddots & \\ & & & 1 & \gamma(\mathbf{x}) & 1 \\ & & & & 1 & \gamma(\mathbf{x}) \end{bmatrix}, \quad \gamma(\mathbf{x}) := 3 + \|\mathbf{x}\|_2.$$

Here $\|\cdot\|_2$ is the Euclidean norm.

A fixed point iteration can be obtained from Eq. (11.1) by the "frozen argument technique": for the step '$k+1$', evaluate the matrix valued function from the previous step 'k' and solve a linear system for the '$(k+1)$-th' iterate.

(a) Derive the fixed point iteration for Eq. (11.1) and implement a C++ function;

```
template <class func, class Vector>
void fixed_point_step(const func& A, const Vector & b,
                      const Vector & x, Vector & x_new);
```

to advance from $\mathbf{x}^{(k)}$ to $\mathbf{x}^{(k+1)}$.

Important Remark: The lambda function for `func A` is provided in the template.

SOLUTION:

C++11-code 11.5: Iterate computation.

```
2  template <class func, class Vector>
3  void fixed_point_step(const func& A, const Vector & b, const Vector
       & x, Vector & x_new) {
4    // Next step
5    auto T = A(x);
6    Eigen::SparseLU<Eigen::SparseMatrix<double>> Ax_lu;
7    Ax_lu.analyzePattern(T);
8    Ax_lu.factorize(T);
9    x_new = Ax_lu.solve(b);
10 }
```

(b) Implement a C++ function:

```
template <class func, class Vector>
void fixed_point_method(const func& A, const Vector& b,
    const double atol, const double rtol, const int max_itr);
```

which computes the solution $\mathbf{x}^*$ using `fixed_point_step`. The input arguments are the absolute tolerance `atol`, relative tolerance `rtol` and maximum iterations `max_itr`. Use $\mathbf{x}^{(0)} = \mathbf{b}$ as initial guess.

SOLUTION:

C++11-code 11.6: Solution of fixed point.

```
template <class func, class Vector>
void fixed_point_method(const func& A, const Vector& b, const double
    atol, const double rtol, const int max_itr) {

          auto fix_step = [&A, &b] (const Eigen::VectorXd & x,
             Eigen::VectorXd & x_new) { fixed_point_step(A, b, x,
             x_new); };
     auto x = b;
    auto x_new = x;

    for( int itr = 0; itr<max_itr; itr++) { // till max itr

        fix_step(x, x_new); // Advance to next step
        double r = (x - x_new).norm();// Compute residual

        std::cout << "[Step " << itr << "] Error: " << r <<
           std::endl;

        // atol, termination
        if (r < atol) {
            std::cout << "[CONVERGED] in " << itr << " it. due to
               atol. err = " << r << " < " << atol << "." <<
               std::endl;
            break;
        }
        // rtol, termination
        if (r < rtol*x_new.norm()) {
            std::cout << "[CONVERGED] in " << itr << " it. due to
               rtol. err = " << r << " < " << rtol*x_new.norm() <<
               "." << std::endl;
            break;
        }
        x = x_new;
    }
    std::cout << std::endl << "x^*_fix = " << std::endl << x_new <<
       std::endl;
}
```

(c) Derive the Newton iteration for Eq. (11.1) and implement a C++ function as in subproblem(a):

```
template <class func, class Vector>
void newton_step(const func& A, const Vector & b,
                 const Vector & x, Vector & x_new);
```

Note that the Jacobian will be a rank 1 modification. So, ideally, you can use the Sherman-Morrison-Woodbury formula to compute the inverse of the Jacobian optimally.

SOLUTION:

The Newton iteration, as provided in the lecture notes, reads:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \left(\mathbf{A}(\mathbf{x}^{(k)}) + \frac{\mathbf{x}^{(k)}(\mathbf{x}^{(k)})^\top}{\|\mathbf{x}^{(k)}\|_2} \right)^{-1} (\mathbf{A}(\mathbf{x}^{(k)})\mathbf{x}^{(k)} - \mathbf{b}) \tag{11.2}$$

We replace the inversion with the SMW formula:

$$\begin{aligned}
\mathbf{x}^{(k+1)} &= \mathbf{x}^{(k)} - \left(\mathbf{A}(\mathbf{x}^{(k)}) + \frac{\mathbf{x}^{(k)}(\mathbf{x}^{(k)})^\top}{\|\mathbf{x}^{(k)}\|_2} \right)^{-1} (\mathbf{A}(\mathbf{x}^{(k)})\mathbf{x}^{(k)} - \mathbf{b}) \\
&= \mathbf{x}^{(k)} - \mathbf{A}(\mathbf{x}^{(k)})^{-1} \left(\mathbf{I} - \frac{1}{\|\mathbf{x}^{(k)}\|_2} \frac{\mathbf{x}^{(k)}(\mathbf{x}^{(k)})^\top \mathbf{A}(\mathbf{x}^{(k)})^{-1}}{1 + \frac{(\mathbf{x}^{(k)})^\top \mathbf{A}(\mathbf{x}^{(k)})^{-1}\mathbf{x}^{(k)}}{\|\mathbf{x}^{(k)}\|_2}} \right) (\mathbf{A}(\mathbf{x}^{(k)})\mathbf{x}^{(k)} - \mathbf{b}) \\
&= \mathbf{A}(\mathbf{x}^{(k)})^{-1}\mathbf{b} + \mathbf{A}(\mathbf{x}^{(k)})^{-1} \frac{\mathbf{x}^{(k)}(\mathbf{x}^{(k)})^\top \mathbf{A}(\mathbf{x}^{(k)})^{-1}}{\|\mathbf{x}^{(k)}\|_2 + (\mathbf{x}^{(k)})^\top \mathbf{A}(\mathbf{x}^{(k)})^{-1}\mathbf{x}^{(k)}} (\mathbf{A}(\mathbf{x}^{(k)})\mathbf{x}^{(k)} - \mathbf{b}) \\
&= \mathbf{A}(\mathbf{x}^{(k)})^{-1} \left(\mathbf{b} + \frac{\mathbf{x}^{(k)}(\mathbf{x}^{(k)})^\top (\mathbf{x}^{(k)} - \mathbf{A}(\mathbf{x}^{(k)})^{-1}\mathbf{b})}{\|\mathbf{x}^{(k)}\|_2 + (\mathbf{x}^{(k)})^\top \mathbf{A}(\mathbf{x}^{(k)})^{-1}\mathbf{x}^{(k)}} \right)
\end{aligned}$$

C++11-code 11.7: Newton step.

```
2  template <class func, class Vector>
3  void newton_step(const func& A, const Vector & b, const Vector & x,
       Vector & x_new) {
4    // Reuse LU decomposition with SMW
5    auto T = A(x);
6    Eigen::SparseLU<Eigen::SparseMatrix<double>> Ax_lu;
7    Ax_lu.analyzePattern(T);
8    Ax_lu.factorize(T);
9    // Solve a bunch of systems
10   auto Axinv_b = Ax_lu.solve(b);
11   auto Axinv_x = Ax_lu.solve(x);
12   // Next step
13   x_new = Axinv_b + Ax_lu.solve(x*x.transpose()*(x-Axinv_b)) /
         (x.norm() + x.dot(Axinv_x) );
14 }
```

(d) Implement a C++ function similar to `fixed_point_method`:

```
template <class func, class Vector>
void newton_method(const func& A, const Vector& b,
    const double atol, const double rtol, const int max_itr);
```

SOLUTION:

C++11-code 11.8: Solve Newton.

```
template <class func, class Vector>
void newton_method(const func& A, const Vector& b, const double
    atol, const double rtol, const int max_itr) {

        auto newt_step = [&A, &b] (const Eigen::VectorXd & x,
            Eigen::VectorXd & x_new) { newton_step(A, b, x, x_new);
            };
    auto x = b;
    auto x_new = x;

    for( int itr = 0; itr < max_itr; itr++) { // till max itr

        newt_step(x, x_new);// Advance to next step
        double r = (x - x_new).norm();// Compute residual

        std::cout << "[Step " << itr << "] Error: " << r <<
            std::endl;

        // atol, termination
        if (r < atol) {
            std::cout << "[CONVERGED] in " << itr << " it. due to
                atol. err = " << r << " < " << atol << "." <<
                std::endl;
            break;
        }
        // rtol, termination
        if (r < rtol*x_new.norm()) {
            std::cout << "[CONVERGED] in " << itr << " it. due to
                rtol. err = " << r << " < " << rtol*x_new.norm() <<
                "." << std::endl;
            break;
        }
        x = x_new;
    }
    std::cout << std::endl << "x^*_newt = " << std::endl << x_new <<
        std::endl;
}
```
