

ETH Lectures 401-0663-00L Numerical Methods for Computer Science
401-2673-00L Numerical Methods for CSE

Homework Problems and Projects

Prof. R. Hiptmair, SAM, ETH Zurich

Autumn Term 2021, Version of October 2, 2022
(C) Seminar für Angewandte Mathematik, ETH Zürich

[Link](#) to the current version of this homework collection

0.1 General Information

0.1.1 Weekly Homework Assignments

All problems will be published in this single “.pdf” file. Every week, we publish a list of problem numbers on the lecture Moodle page. These are the current assignments for that week. Details are given on the lecture Moodle page.

0.1.2 Importance of Homework

Homework assignments are **not** mandatory. However, it is **very important** that you constantly exercise with the material you learn. “Solving” the homework assignments one week before the main exam by looking at the solutions will likely result in failure.

We provide hints and solutions from the beginning. It is your responsibility to look at the solutions only if you are stuck with a difficult problem and you tried to find a solution for a sufficient amount of time.

Most homework problems come with an estimate for the time required for its solution. These times are what we expect a bright and well-prepared student to require for that problem during an exam.

0.1.3 Corrections and Grading of Assignments

Homework assignments will not be graded.

However, you may submit your handwritten solutions by handing them to your tutor directly or uploading them via the Moodle upload interface, see the lecture Moodle page for instructions. The assistants will

Problem 8-9: Modified Newton method

The following problem consists in EIGEN implementation of a modified version of the Newton method (in one dimension [Lecture → Section 8.4.2.1] and many dimensions [Lecture → Section 8.5]) for the solution of a nonlinear system.

Involves coding in C++. Refresh your knowledge of stopping criteria for iterative methods [Lecture → Section 8.2.3].

For the solution of the non-linear system of equations $\mathbf{F}(\mathbf{x}) = \mathbf{0}$ (with $\mathbf{F} : \mathbb{R}^n \rightarrow \mathbb{R}^n$), the following iterative method has been proposed:

$$\begin{aligned} \mathbf{y}^{(k)} &= \mathbf{x}^{(k)} + \mathbf{D}\mathbf{F}(\mathbf{x}^{(k)})^{-1} \mathbf{F}(\mathbf{x}^{(k)}) , \\ \mathbf{x}^{(k+1)} &= \mathbf{y}^{(k)} - \mathbf{D}\mathbf{F}(\mathbf{x}^{(k)})^{-1} \mathbf{F}(\mathbf{y}^{(k)}) , \end{aligned} \quad (8.9.1)$$

where $\mathbf{D}\mathbf{F}(\mathbf{x}) \in \mathbb{R}^{n,n}$ is the Jacobian matrix of $\mathbf{F}$ evaluated in the point $\mathbf{x}$.

(8-9.a) 🕒 (15 min.) Assume that $\mathbf{F}$ is **injective**,

$$\mathbf{x}, \mathbf{y} \in \mathbb{R}^n, \quad \mathbf{F}(\mathbf{x}) = \mathbf{F}(\mathbf{y}) \Rightarrow \mathbf{x} = \mathbf{y} . \quad (8.9.2)$$

Show that under this assumption the iteration (8.9.1) is **consistent** with $\mathbf{F}(\mathbf{x}) = \mathbf{0}$ in the sense of [Lecture → Def. 8.3.1.1], that is, show that, under the assumption that $\mathbf{D}\mathbf{F}(\mathbf{x}^*)$ is regular for every fixed point $\mathbf{x}^*$ of the iteration, $\mathbf{x}^{(k)} = \mathbf{x}^{(0)}$ for every $k \in \mathbb{N}$ if and only if $\mathbf{F}(\mathbf{x}^{(0)}) = \mathbf{0}$.

SOLUTION of (8-9.a):

- ❶ 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 .
- ❷ Conversely, if $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} = \dots = \mathbf{x}^{(0)}$, then, by the recursion of the Newton method,

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

Then we plug the first equation into the second and infer

$$\begin{aligned} \mathbf{D}\mathbf{F}(\mathbf{x}^{(k)})^{-1} \mathbf{F}(\mathbf{x}^{(k)}) - \mathbf{D}\mathbf{F}(\mathbf{x}^{(k)})^{-1} \mathbf{F}(\mathbf{x}^{(k)} + \mathbf{D}\mathbf{F}(\mathbf{x}^{(k)}) \mathbf{F}(\mathbf{x}^{(k)})) &= \mathbf{0} , \\ [\mathbf{D}\mathbf{F}(\mathbf{x}^{(k)}) \text{ regular}] \quad \blacktriangleright \quad \mathbf{F}(\mathbf{x}^{(k)}) - \mathbf{F}(\mathbf{x}^{(k)} + \mathbf{D}\mathbf{F}(\mathbf{x}^{(k)}) \mathbf{F}(\mathbf{x}^{(k)})) &= \mathbf{0} . \end{aligned}$$

Finally, we exploit the injectivity (8.9.2) of $\mathbf{F}$ and again the regularity of $\mathbf{D}\mathbf{F}(\mathbf{x}^{(k)})$:

$$\mathbf{F}(\mathbf{x}^{(k)}) = \mathbf{F}(\mathbf{x}^{(k)} + \mathbf{D}\mathbf{F}(\mathbf{x}^{(k)})^{-1} \mathbf{F}(\mathbf{x}^{(k)})) \Leftrightarrow \mathbf{D}\mathbf{F}(\mathbf{x}^{(k)})^{-1} \mathbf{F}(\mathbf{x}^{(k)}) = \mathbf{0} \Leftrightarrow \mathbf{F}(\mathbf{x}^{(k)}) = \mathbf{0} .$$



(8-9.b) 🕒 (20 min.) Implement a C++ function

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

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

Here, `f` is a functor object of type `Function` passing the function $F : \mathbb{R} \mapsto \mathbb{R}$ and `df` a functor object of type `Jacobian` passing the derivative $F' : \mathbb{R} \mapsto \mathbb{R}$. Both require an appropriate evaluation operator `operator()` and have to conform with `std::function<double(double)>`.

SOLUTION of (8-9.b):

C++ code 8.9.3: Implementation of `mod_newt_step_scalar`.

```

2  template <typename Scalar, class Function, class Jacobian>
3  Scalar mod_newt_step_scalar(const Scalar& x, Function&& f, Jacobian&& df) {
4      double return_value = 0.;
5      // TODO: (8-9.b) Compute a step of the modified Newton method for a
6      // scalar
7      // function
8      // START
9      Scalar y = x + f(x) / df(x);
10     return_value = y - f(y) / df(x);
11     // END
12     return return_value;
13 }
```

Get it on  GitLab ([modnewt.hpp](#)).

(8-9.c)  (30 min.) What is the order of convergence of the method described in (8.9.1)?

To investigate it, write a C++ function

```
void mod_newt_ord(void);
```

that

- uses the function `mod_newt_step` to the following scalar equation: $\arctan(x) - 0.123 = 0$,
- generates a suitable terminal output that makes it possible to empirically determine the order of convergence, in the sense of [Lecture → Rem. 8.2.2.12],
- implements meaningful stopping criteria, see [Lecture → Section 8.2.3].

Use $x_0 = 5$ as initial guess and tabulate data from which you can read off the order.

HINT 1 for (8-9.c): The exact solution is $x = \tan(a) = 0.123624065869274 \dots$

HINT 2 for (8-9.c): Remember that $\arctan'(x) = \frac{1}{1+x^2}$.

SOLUTION of (8-9.c):

We implement a generic function `sample_nonlinear_solver()` 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 initial guess, and termination parameters.

C++ code 8.9.4: Implementation of sample_nonlinear_solver.

```

2  template <class StepFunction, class Vector, class ErrorFunction>
3  bool sample_nonlinear_solver(const StepFunction& step, Vector& x,
4                             const ErrorFunction& errf, double eps = 1e-8,
5                             unsigned int max_itr = 100) {
6      // TODO: (8-9.c,f) Implement this generic function that iteratively
7      // solves a
8      // nonlinear equation. The function should return whether the
9      // iteration
10     // converged in the maximum number of iterations
11     // START.
12     // Temporary where to store new step
13     Vector x_new = x;
14     double r = 1;
15
16     for (unsigned itr = 0; itr < max_itr; ++itr) {
17         // Compute error (or residual)
18         r = errf(x);
19
20         std::cout << "[Step " << itr << "] Error: " << r << std::endl;
21
22         // Advance to next step, x_new becomes x_{k+1}
23         x_new = step(x);
24
25         // Termination conditions
26         // If tol reached, then we have convergence
27         if (r < eps * norm(x)) {
28             std::cout << "[CONVERGED] in " << itr << " it. due to err. err = " << r
29             << " < " << eps << "." << std::endl;
30             return true;
31         }
32         x = x_new;
33     }
34
35     // If max it reached
36     std::cout << "[NOT CONVERGED] due to MAX it. = " << max_itr
37     << " reached, err = " << r << "." << std::endl;
38
39     // END
40     return false;
41 }

```

Get it on  [GitLab \(modnewt.hpp\)](https://gitlab.com/modnewt/modnewt.hpp).

We employ the method discussed in [Lecture → Rem. 8.2.2.12] to estimate the order of convergence of the iteration. The estimator is

$$\text{estord} := \frac{\log \epsilon_{k+1} - \log \epsilon_k}{\log \epsilon_k - \log \epsilon_{k-1}}, \quad \epsilon_k := \left\| \mathbf{x}^{(k)} - \mathbf{x}^* \right\|_2. \quad (8.9.5)$$

C++ code 8.9.6: Implementation of mod_newt_ord.

```

2  void mod_newt_ord(void) {
3      // Setting up values, functions and jacobian
4      constexpr double a = 0.123;
5      auto f = [](double x) {
6          return atan(x) - a;
7      };
8      // Function which sees a as it is constexpr
9      double x_scalar = 5; // initial guess
10
11     // TODO: (8-9.c) Generate suitable terminal output to determine the
12     // order of

```

```

11 // convergence of the modified Newton method applied to the scalar
12 // equation
13 // that is given by  $f=0$ . You may want to implement the templated
14 // function
15 // sample_nonlinear_solver for the iterations.
16 // START
17 auto df = [](double x) { return 1. / (x * x + 1.); }; // Its derivative
18
19 const double x_ex = tan(a); // Exact solution
20
21 // Store solution and error at each step
22 std::vector<double> sol;
23 std::vector<double> err;
24
25 // Compute error and push back to err, used in
26 // sample_nonlinear_solver as breaking condition  $\text{errf}(x) < \text{eps}$ 
27 auto errf = [&err, x_ex](double& x) {
28     double e = std::abs(x - x_ex);
29     err.push_back(e);
30     return e;
31 };
32
33 // Perform convergence study with Modified newton for scalar
34 std::cout << std::endl
35     << "*** Modified Newton method (scalar) ***" << std::endl
36     << std::endl;
37 std::cout << "Exact: " << x_ex << std::endl;
38
39 // Initial guess and compute initial error
40 sol.push_back(x_scalar);
41 errf(x_scalar);
42
43 // Lambda performing the next step, used to define a proper
44 // function handle to be passed to sample_nonlinear_solver
45 auto newt_scalar_step = [&sol, &f, &df](double x) -> double {
46     double x_new = mod_newt_step_scalar(x, f, df);
47     sol.push_back(x_new);
48     return x_new;
49 };
50
51 // Actually perform the solution
52 sample_nonlinear_solver(newt_scalar_step, x_scalar, errf);
53
54 // Print solution (final)
55 std::cout << std::endl << "x^*_scalar = " << x_scalar << std::endl;
56
57 // Print table of solutions, errors and EOOO
58 auto space = std::setw(15);
59 std::cout << space << "sol." << space << "err." << space << "order"
60     << std::endl;
61 for (unsigned i = 0; i < sol.size(); ++i) {
62     std::cout << space << sol.at(i) << space << err.at(i);
63     if (i >= 3) {
64         std::cout << space
65             << (log(err.at(i)) - log(err.at(i - 1))) /
66             (log(err.at(i - 1)) - log(err.at(i - 2)));
67     }
68     std::cout << std::endl;
69 }
70 // END
71 }

```

Get it on  GitLab (modnewt.hpp).

solution	error	estord
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

The method converges very fast so that the iteration error quickly drops to the level of machine precision, where it is polluted by roundoff errors. So it is difficult to estimate the rate of convergence from the data.

We can just infer that the rate seems to be between 2 and 3.



(8-9.d)  (20 min.) Implement a templated C++ function

```
template <typename Vector, typename Function, typename Jacobian>
Vector mod_newt_step_system(const Vector &x,
                           Function &&f, Jacobian& df);
```

that *efficiently* realizes a step of the iteration (8.9.1) for a function $F : \mathbb{R}^n \rightarrow \mathbb{R}^n$ passed through the functor `f`, which complies with `std::function<Vector(const Vector &)>`; The **Vector** type can be assumed to have the capabilities of **Eigen::VectorXd**, whereas the type **Jacobian** must have an evaluation operator that returns an object compatible with **Eigen::MatrixXd**.

HINT 1 for (8-9.d): Do not forget to perform a single LU-decomposition of the Jacobian and reuse it. See [Lecture → Code 8.5.3.6] for an example.

SOLUTION of (8-9.d):

C++ code 8.9.7: Implementation of mod_newt_step_system.

```
2 template <typename Vector, class Function, class Jacobian>
3 Vector mod_newt_step_system(const Vector& x, Function&& f, Jacobian& df) {
4     Vector return_value = x; // this is just a dummy, you should overwrite
5     // it
6     // TODO: (8-9.d) Efficiently perform one step of the modified Newton
7     // method
8     // for a system of equations.
9     // START
10    auto lu = df(x).lu();
11    // Reusing LU decomposition
12    Vector y = x + lu.solve(f(x));
13    return_value = y - lu.solve(f(y));
14    // END
15    return return_value;
16 }
```

Get it on  GitLab (modnewt.hpp).



In the sequel we consider the non-linear system of equations

$$F(x) := Ax + \begin{bmatrix} c_1 e^{x_1} \\ \vdots \\ c_n e^{x_n} \end{bmatrix} = 0, \quad (8.9.8)$$

where $\mathbf{A} \in \mathbb{R}^{n,n}$ is symmetric positive definite, $\mathbf{x} = [x_j]_{j=1}^n \in \mathbb{R}^n$ is the solution vector, and $c_i \geq 0$, $i = 1, \dots, n$.

(8-9.e)  Compute the Jacobian matrix $\mathbf{D}\mathbf{F}(\mathbf{x})$ for $\mathbf{F}$ as given in (8.9.8).

SOLUTION of (8-9.e):

We can simply rely on component-wise partial derivatives to derive

$$\mathbf{D}\mathbf{F}(\mathbf{x}) = \mathbf{A} + \begin{bmatrix} c_1 e^{x_1} & & & \\ & c_2 e^{x_2} & & \\ & & \ddots & \\ & & & c_n e^{x_n} \end{bmatrix} \in \mathbb{R}^{n,n}. \quad (8.9.9)$$

▲

(8-9.f)  (30 min.) [depends on Sub-problem (8-9.d)]

Based on your implementation of `mod_newt_step_system()`, create a C++ function

```
Eigen::VectorXd mod_newt_sys(const Eigen::Matrix &A,
                             const Eigen::VectorXd& c,
                             double tol 1.0E-6, int maxit = 100);
```

that uses the modified Newton iteration (8.9.1) to solve (8.9.8). The argument `A` passes the matrix $\mathbf{A}$ and `c` supplies the values c_j .

Stop the iteration and return the approximate solution when the Euclidean norm of the increment $\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}$ relative to the norm of $\mathbf{x}^{(k+1)}$ is smaller than the tolerance passed in `tol` or if `maxit` steps of the iterations have been carried out. Use the zero vector as initial guess.

SOLUTION of (8-9.f):

C++ code 8.9.10: Implementation of `mod_newt_sys`.

```
2  Eigen::VectorXd mod_newt_sys(const Eigen::MatrixXd& A, const Eigen::VectorXd& c,
3                               double tol = 1.0e-6, int maxit = 100) {
4      Eigen::VectorXd x_system = Eigen::VectorXd::Zero(c.size()); // Initial guess
5
6      // TODO: (8-9.f) Solve the non-linear system of equations (9.9.6)
7      // using the
8      // modified Newton method. You may want to use your implementation
9      // of
10     // sample_nonlinear_solver.
11     // START
12
13     // Handler for function and jacobian in standard format. Must see
14     // matrix A
15     // and vector c.
16     auto F = [&A, &c](const Eigen::VectorXd& x) {
17         Eigen::VectorXd tmp =
18             A * x + c.cwiseProduct(x.array().exp().matrix()).eval();
19         return tmp;
20     };
21     auto dF = [&A, &c](const Eigen::VectorXd& x) {
```

```

19 Eigen::MatrixXd C = A;
20 Eigen::VectorXd temp = c.cwiseProduct(x.array().exp().matrix());
21 C += temp.asDiagonal();
22 return C;
23 };
24
25 // Container for errors
26 std::vector<double> err;
27 // Define lambda for breaking condition, which also stores
28 // the error of the previous step. Must see 'err'.
29 auto rerr = [&err](Eigen::VectorXd& x) {
30     if (err.size() > 0)
31         return err.back();
32     else
33         return 1.;
34 };
35
36 // Refactor (i.e. wrap) the step such that it is compatible with
37 // sample_nonlinear_solver. Must see F, dF and 'err'. We also
38 // store the
39 // error.
40 auto newt_system_step = [&F, &dF,
41                         &err](const Eigen::VectorXd& x) -> Eigen::VectorXd {
42     Eigen::VectorXd x_new = mod_newt_step_system(x, F, dF);
43     double e = (x - x_new).norm() / x_new.norm();
44     err.push_back(e);
45     return x_new;
46 };
47
48 // Actually performs computations
49 sample_nonlinear_solver(newt_system_step, x_system, rerr, tol, maxit);
50 // END
51 return x_system;
52 }

```

Get it on  GitLab (modnewt.hpp).



(8-9.g)  (30 min.) [depends on Sub-problem (8-9.f)]

Write a C++ function

```
void mod_newt_sys_test(void);
```

that tests your implementation of `mod_newt_sys()` for

$$\mathbf{A} = \begin{bmatrix} 2 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 2 \end{bmatrix}, \quad \mathbf{c} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix},$$

and determines the order of convergence from suitable tabulated values.

HINT 1 for (8-9.g): Determine a reference solution by calling `mod_newt_sys()` with a very low tolerance. The invoke that function with `tol = 0` and small and increasing values of `maxit`. 

SOLUTION of (8-9.g):

We implement a “stepping” function and reuse the previous codes.

We iterate the Modified Newton 5 times and print out the error for each step with corresponding order of convergence.

The order of convergence is estimate as recommended in [Lecture → Rem. 8.2.2.12], see (8.9.5).

C++ code 8.9.11: Implementation of `mod_newt_sys_test()`.

```

2  void mod_newt_sys_test(void) {
3      Eigen::MatrixX<double> A(4, 4);
4      A << 2., -1., 0., 0., -1., 2., -1., 0., 0., -1., 2., -1., 0., 0., -1., 2.;
5      Eigen::VectorX<double> c(4);
6      c << 1., 2., 3., 4.;
7
8      // TODO: (8-9.g) Determine the order of convergence from suitable
9      // tabulated
10     // values.
11     // determine a reference solution with very low tolerance
12     // (machine precision)
13     // START
14     constexpr double reference_tol = std::numeric_limits<double>::epsilon();
15     Eigen::VectorX<double> reference_sol = mod_newt_sys(A, c, reference_tol);
16
17     std::vector<double> err;
18     std::vector<Eigen::VectorX<double>> sol;
19
20     // seperate calculation and printing to not pollute the table from
21     // output of
22     // iteration function increase the number of maximal iterations
23     sol.push_back(Eigen::VectorX<double>::Zero(4));
24     err.push_back(reference_sol.norm());
25     for (unsigned k = 1; k < 5; ++k) {
26         sol.push_back(mod_newt_sys(A, c, 0., k));
27         err.push_back((reference_sol - sol.at(k)).norm());
28     }
29
30     // Perform convergence study with Modified newton for system of
31     // equations
32     std::cout << std::endl
33               << "*** Modified Newton method (system) ***" << std::endl
34               << std::endl;
35     std::cout << "Exact: " << reference_sol.transpose() << std::endl;
36
37     auto space = std::setw(15);
38     std::cout << space << "k" << space << "err." << space << "order" << std::endl;
39
40     // print iteration number, error and order
41     for (unsigned k = 0; k < 5; ++k) {
42         std::cout << space << k << space << err.at(k);
43         if (k >= 1 && k < 4) {
44             std::cout << space
45                       << (log(err.at(k + 1)) - log(err.at(k))) /
46                          (log(err.at(k)) - log(err.at(k - 1)));
47         }
48         std::cout << std::endl;
49     }
50     // END
51 }

```

Get it on  GitLab (modnewt.hpp).

k	error	estord
0	2.87469	
1	0.359054	3.39385
2	0.000308353	2.74505
3	1.18145e-12	0.442692
4	2.22045e-16	

◁ Error table as printed by `mod_newt_sys()`.

As in Sub-problem (8-9.c) the method converges very fast and it is difficult to tease out a convergence order, because only few meaningful data points are generated before the iteration error becomes polluted by roundoff error.

The order of convergence seems to be > 2 consistent with what we observed in Sub-problem (8-9.c).



End Problem 8-9, 145 min.