

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).

We employ the method discussed in [Lecture $\rightarrow$ 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() {
3     // Setting up values, functions and jacobian
4     constexpr double a = 0.123;
5     auto f = [](double x) {
6         return atan(x) - a;
7     }; // Function which sees a as it is constexpr
8     double x_scalar = 5; // initial guess
9
10    // TODO: (8-9.c) Generate suitable terminal output to determine the
11    // order of
12    // convergence of the modified Newton method applied to the scalar
13    // equation
14    // that is given by  $f=0$ . You may want to implement the templated
15    // function
16    // sample_nonlinear_solver for the iterations.
17    // START
18    auto df = [](double x) { return 1. / (x * x + 1.); }; // Its derivative
19
20    const double x_ex = tan(a); // Exact solution
21
22    // Store solution and error at each step
23    std::vector<double> sol;
24    std::vector<double> err;
25
26    // Compute error and push back to err, used in
27    // sample_nonlinear_solver as breaking condition  $\text{errf}(x) < \text{eps}$ 
28    auto errf = [&err, x_ex](double& x) {
29        double e = std::abs(x - x_ex);
30        err.push_back(e);
31        return e;
32    };
33
34    // Perform convergence study with Modified newton for scalar
35    std::cout << std::endl
36              << "*** Modified Newton method (scalar) ***" << std::endl
37              << std::endl;
38    std::cout << "Exact: " << x_ex << std::endl;
39
40    // Initial guess and compute initial error
41    sol.push_back(x_scalar);
42    errf(x_scalar);
43
44    // Lambda performing the next step, used to define a proper
45    // function handle to be passed to sample_nonlinear_solver
46    auto newt_scalar_step = [&sol, &f, &df](double x) -> double {
47        double x_new = mod_newt_step_scalar(x, f, df);
48        sol.push_back(x_new);
49        return x_new;
50    };
51
52    // Actually perform the solution
53    sample_nonlinear_solver(newt_scalar_step, x_scalar, errf);
54
55    // Print solution (final)
56    std::cout << std::endl << "x*_scalar = " << x_scalar << std::endl;
57
58    // Print table of solutions, errors and EOOO
```

```

56 auto space = std::setw(15);
57 std::cout << space << "sol." << space << "err." << space << "order"
58     << std::endl;
59 for (unsigned i = 0; i < sol.size(); ++i) {
60     std::cout << space << sol.at(i) << space << err.at(i);
61     if (i >= 3) {
62         std::cout << space
63             << (log(err.at(i)) - log(err.at(i - 1))) /
64                 (log(err.at(i - 1)) - log(err.at(i - 2)));
65     }
66     std::cout << std::endl;
67 }
68 // END
69 }

```

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](#).