Homework assignment 3

Nonlinear finite elements

Note: The submission of homework assignments 1 to 4 is mandatory to pass the course. The assignments lead from the mathematical formulation of a model problem to the numerical solution of this problem. They build on each other. You must achieve at least 50 of the achievable points on each assignment.

For all tasks, include the solution steps and intermediate results. The final result alone is not sufficient! We recommend that you use Python for the solution of numerical tasks. Please create a PDF and attach the numerical codes as separate files. Please also include a PDF if you use Jupyter notebooks.

Problem 1 Newton-Raphson-Verfahren

Problem 1.1 Scalar-valued functions

Implementation of a Newton solver

6 achievable points

Find a root for any function $f(x)$ numerically. Write a solver that locally solves $f(x) = 0$ using a simple Newton iteration method. Use the following interface for this:

```
def newton(fun, x0, jac, tol=1e-6, maxiter=200,
           callback=None):
    """
    Newton solver expects scalar function fun and scalar
    initial value x0.
```

```
6
7       Parameters
8       ----------
9       fun : callable(x)
10          Scalar function of a scalar.
11      x0 : float
12          Scalar initial value.
13      jac : callable(f, x)
14          Scalar derivative of f at position x.
15      tol : float, optional
16          Tolerance (with respect to the zero value) for
17          convergence.
18      maxiter : int, optional
19          Maximum number of Newton iterations.
20      callback : callable(x, **kwargs), optional
21          Callback function to handle logging and convergence
22          stats.
23
24      Returns
25      -------
26      x : float
27          Approximate local root.
28      """
```

Aborting the Newton iteration

The Newton method must be aborted at some point. The function signature provides two mechanisms for this: the parameter `tol` specifies a tolerance ε by which the function value may differ from zero. This means that the loop is aborted if the function value $|f(x_i)| < \varepsilon$ is fulfilled at time step i. To avoid situations in which no convergence can be achieved, the function also provides for a maximum number of steps (parameter `maxiter`). The iteration is aborted if one of these two criteria is met. Since the Newton method has not converged when the second criterion is met, an error should be reported here. This can be done, for example, using an exception with the `raise` command.

Callback

The interface mentioned above contains a so-called *callback* function. This should follow the interface

```
1 def solver_callback(x):
2     """Solver callback for logging.
3
4     Parameters
```

```
    ----------
    x : float or np.ndarray
        Current approximate solution
    """
    ...
```

and, if specified, be called once per Newton iteration with the current approximate solution x. This callback can then keep track of all intermediate solutions, for example, and output current convergence criteria, thus helping you to troubleshoot.

Implement such a callback. What exactly is displayed is up to you.

Illustration using a simple polynomial

First test your Newton solver on the simple polynomial $f(x) = x^3 + x^2 - x + 1$. Choose different starting values for this. For $x_0 = 0.9$ and a relative convergence criterion of $\varepsilon = 0.01$, the output of your callback, the approximation solutions visited and the convergence behavior could look like this, for example:

```
          X           FUN           JAC
          =           ===           ===
  9.000e-01     1.639e+00     3.230e+00
  3.926e-01     8.220e-01     2.475e-01
 -2.929e+00    -1.262e+01     1.888e+01
 -2.261e+00    -3.182e+00     9.810e+00
 -1.936e+00    -5.741e-01     6.375e+00
 -1.846e+00    -3.827e-02     5.533e+00
 -1.839e+00    -2.168e-04     5.471e+00
 -1.839e+00    -7.094e-09     5.470e+00
 -1.839e+00    -2.220e-16     5.470e+00
```

Now find the root for the same polynomial by starting at $x_0 = -1.1$. Show the steps of the Newton solver for the starting points $x_0 = 0.9$ and $x_0 = -1.1$, plotting the current position x_i against the iteration number i. What do you observe?

Note: Your Newton solver should work for arbitrary functions. To do this, you have to implement the derivative of the function by hand. For example, to find the root of the function $f(x) = x^2 - 1$, you need an implementation of the derivative $\mathrm{d}f/\mathrm{d}x = 2x$, for example in the following form:

```python
def f(x):
    return x * x - 1
def df(x):
    return 2 * x
```

You can use automatic differentiation from the library JAX to compute the derivative.

Problem 1.2 Vector-valued functions

7 achievable points

Generalize the implementation of your Newton solver to vector-valued functions $\vec{f}(\vec{x})$. The `fun` function in the function signature above must now return a vector (i.e., a `numpy` array). The `jac` function returns a matrix. It may make sense to implement a separate function for the vector-valued implementation. Use your solver to find the minimum of the function $g(x, y) = 2\exp(-10(x^2 + y^2)) + x^2 + y^2 + x$.

Please answer the following questions in the context of solving this task:

- So far, we have been talking about the Newton method for solving coupled nonlinear equations. Here, we are now asking for the minimization of a function, which is an optimization problem. How does this fit together? What special structure does the Jacobian matrix have for an optimization problem? What do you also call this matrix in this case?
- You have to consider when to abort the Newton iteration. What could be a reasonable criterion here and why?
- What do the iterations of your solver look like in a two-dimensional plot in the x-y plane, starting from the starting points $(1/2, 1/2)$ and $(5, 5)$? Show the function values $g(x, y)$ color-coded in the background. This can be done, for example, with the `matplotlib` function `pcolormesh` or `contour`.

Problem 2 Poisson-Boltzmann equation

We now turn to the solution of the one-dimensional Poisson-Boltzmann equation. In its dimensionless form, this is

$$\frac{\mathrm{d}^2\,\Phi}{\mathrm{d}x^2} = \sinh\Phi. \tag{3.1}$$

The aim of this task is to implement a solver for this nonlinear equation. You can use results from the lecture notes, such as expressions for the element matrices and code snippets.

Problem 2.1 Discretization

3 achievable points

Discretize the nonlinear Poisson equation. Where possible, solve the integrals involving shape functions and write down the element matrices.

Note: Remember that it is sufficient to express the basis functions using the shape functions. The exact expressions for the shape functions do not yet play a role here.

Problem 2.2 Numerical quadrature of the residual

2 achievable points

The final equations for the (discrete) residual contain terms of the form $(N_I, \sinh \Phi)$, where N_I is the basis function and Φ is the approximated solution of the PDE. These terms can be approximated with Gaussian quadrature. Write the scalar products/integrals that cannot be solved analytically using the appropriate type of numerical quadrature. Do not fix the number of quadrature points here, but derive these equations for an arbitrary number of quadrature points.

Problem 2.3 Tangent Matrix

3 achievable points

Derive the tangent matrix. Show what form the tangent matrix takes in the linearized form. Compare this linearized tangent matrix with the system matrix of the linear problem.

Problem 2.4 Implementation of the residual vector and the tangent matrix

4 achievable points

Implement the residual vector and the tangent matrix. We suggest the following signatures for the functions that implement the calculation:

```
def residual(potential_g,
             potential_left=0, potential_right=0,
             dx=1, nb_quad=2, linear=False):
    """
    Assemble global residual vector for a specific potential.

    Parameters
    ----------
    potential_g : np.ndarray
        Current potential on the nodes (the expansion
        coefficients); the length of the array is the number
        of nodes.
    potential_left : float
        Left Dirichlet boundary condition.
    potential_right : float
        Right Dirichlet boundary condition.
    dx : float, optional
        Grid spacing. (Default: 1)
    nb_quad : int, optional
        Number of quadrature points. (Default: 2)
    linear : bool, optional
        Linearize mass matrix. (Default: False)

    Returns
    -------
    residual_g : np.ndarray
        Residual vector (same shape as `potential_g`)
    """
    ...

def tangent(potential_g,
            potential_left=0, potential_right=0,
            dx=1, nb_quad=2, linear=False):
    """
    Assemble global tangent matrix for a specific potential.

    Parameters
    ----------
    potential_g : np.ndarray
        Current potential on the nodes (the expansion
        coefficients); the length of the array is the number
        of nodes
    potential_left : float
        Left Dirichlet boundary condition.
    potential_right : float
        Right Dirichlet boundary condition.
```

```
    dx : float, optional
        Grid spacing. (Default: 1)
    nb_quad : int, optional
        Number of quadrature points. (Default: 2)
    linear : bool, optional
        Linearize mass matrix. (Default: False)

    Returns
    -------
    tangent_gg : np.ndarray
        Tangent matrix (quadratic, number of rows and columns
        equal number of nodes)
    """
    ...
```

When implementing, you should proceed step by step: First, ignore the Dirichlet conditions and implement only the Laplace operator. Write the code for the mass matrix (the $\sinh \Phi$ term of the differential equation) only once the Laplace operator is working. Also implement a linear variant where $\sinh \Phi \approx \Phi$. This can server as a reference solution, and we ask you below to compare with the linearized solution.

Note: It is important that the `tangent` function returns the correct derivative(s) of `residual`. To make your life easy, use automatic differentiation from the library JAX.

If you still want to implement the gradient manually, you can test your implementation numerically. For example, you can calculate the derivative of the `residual` function numerically using the difference quotients and check it against the analytical calculation of `tangent`. Below is a code block that does this numerical calculation of the derivative for you:

```
def check_tangent(value_g, residual_fun, tangent_fun,
                  eps=1e-6):
    """
    Check that tangent_fun is gives the derivative_fun
    using finite differences.

    Parameters
    ----------
    value_g : numpy.nd_array
        Nodal value for which to check the derivative
    residual_fun : callable
        Function that takes the values and returns an array
        of residual values
    tangent_fun : callable
        Function that takes the values and return the
```

```
        tangent/jacobian matrix
    eps : float, optional
        Finite difference used for numeric computation of
        the derivative (Default: 1e-6)
    """
    nb_nodes = len(value_g)
    tangent_gg = tangent_fun(value_g)
    numeric_tangent_gg = np.zeros_like(tangent_gg)
    for i in range(nb_nodes):
        _value_g = value_g.copy()
        _value_g[i] += eps
        residual_plus_g = residual_fun(_value_g)
        _value_g[i] -= 2 * eps
        residual_minus_g = residual_fun(_value_g)
        numeric_tangent_gg[:, i] = (
            residual_plus_g - residual_minus_g
            ) / (2 * eps)
    np.testing.assert_array_almost_equal(
        tangent_gg, numeric_tangent_gg)


# Check if tangent is implemented correctly
for i in range(10):
    check_tangent(np.random.random(21)-0.5, residual,
   tangent)
```

Problem 2.5 Application of the Newton solver

7 achievable points

Note: If your implementation of the Newton solver does not work for multidimensional problems, you may also use the Newton solver in the `scipy` package. You can find it at `scipy.optimize.newton`.

Solve the nonlinear Poisson–Boltzmann equation with two Dirichlet boundary conditions, one each on the left and right boundaries. Show the solutions of the equation for 1, 2 and 3 Gaussian quadrature points for a system of length $L = 5\lambda$ and a potential of $-1 k_B T/|e|$ on the left electrode and $5 k_B T/|e|$ on the right electrode, where λ is the Debye length. Show a solution with approximately 10 and 100 nodes. Compare this solution with the solution of the linearized equation.

Note:

- You have to start the Newton iteration from a specific potential Φ. What is a good starting point here? If you used the above function signatures, the call to the Newton solver should look something like this:

```
nb_nodes = 21  # Number of notes
potential0_g = ...  # Initial condition
potential_g = newton(residual, potential0_g, tangent)
```

- You may also want to pass a callback function to monitor the convergence of the solver.
- In addition, you should pass the potential boundary conditions and additional parameters, such as the number of quadrature points, to the `residual` and `tangent` functions. You can do this, for example, with lambda expressions.