

STEFFEN HAUG

*Numerical Methods*

Assignment 2

## PROBLEM I

The goal is to implement Newtons method for a generic function  $F : \mathbb{R}^2 \rightarrow \mathbb{R}^2$ . I will use a symbolic computation library to compute the Jacobian  $J_F$ , which means the function needs to consist of sympy-compatible primitives. Except for this restriction, the functions can be generic.

**LISTING 1:** Helper functions for symbolic manipulation

---

```

1 # file: src/alg.py
2 import numpy as np
3 from numpy import linalg as LA
4 import sympy as sp
5 import scipy.sparse as scsp
6
7
8 def symbolic_jac(py_fn):
9     # Computes a symbolic jacobian matrix
10    # for a function f : R^2 -> R^2.
11
12    # compute the entries of the vector by
13    # evaluating the function for sp-symbols
14    x, y = sp.symbols('x y')
15    f1, f2 = py_fn(x, y)
16    F = sp.Matrix([f1, f2])
17
18    return F.jacobian([x, y])
19
20
21 def callable_fn(symbolic):
22    # Create a function that substitutes
23    # for the symbolic values.
24    x, y = sp.symbols('x y')
25    return sp.lambdify(
26        [x, y], symbolic, 'numpy'
27    )

```

---

Armed with some auxillary functions to handle the symbolic computation, we implement the iteration using the Newton method equation

$$\mathbf{x} = \mathbf{x} - J_F^{-1}F(\mathbf{x})$$

**LISTING 2:** Newton's method

---

```

1 # file: src/alg.py
2 MAX_ITER = 100
3
4 def solve(F, x0, tol=1E-6):
5     x, y = x0
6
7     J = symbolic_jac(F)
8
9     # singular jacobian means trouble
10    Jfn = callable_fn(J)
11    assert LA.det(Jfn(x, y)) != 0
12
13    # function-version of the Jacobian
14    Ji = callable_fn(J.inv())
15
16    def step(f, Ji_f, x):
17        # computes the next iteration using the
18        # Newton method equation.
19        # r is the previous step
20        return x - Ji_f(*x).dot(f(*x))
21
22    for _ in range(MAX_ITER):
23        px, py = x, y
24        x, y = step(F, Ji, (x, y))
25        yield x, y
26
27        # check the tolerance criteria
28        if LA.norm(F(x, y)) < tol:
29            break
30        if LA.norm((x - px, y - py)) < tol:
31            break
32
33    def last(it):
34        # run an iterator to the end
35        x = None
36        for x in it: pass
37        return x

```

---

```

>>> from src.alg import solve, last
>>> def F(x, y):
...     return x**2 + y**2 - 2, x - y

>>> solve(F, (-1, 0))
<generator...
>>> last(solve(F, (-1, 0)))
(-1.000000000013107, -1.000000000013107)
>>> last(solve(F, (1, 0)))
(1.000000000013107, 1.000000000013107)

```

---

The interactive session shows how the function can be used, (it may not be so obvious since it is implemented

as a generator-function, so we can collect the error *from outside*; single responsibility principle and so on) and that it is correct at least for two points in different basins of attraction for the equation

$$\mathbf{F}(x, y) = \begin{pmatrix} x^2 + y^2 - 2 \\ x - y \end{pmatrix},$$

which has its true roots in  $(-1, -1)$  and  $(1, 1)$ .

### QUADRATIC CONVERGENCE

We want to verify that Newtons method converges quadratically, also in the multivariable case. To see this, we want to evaluate the limit

$$\mu = \lim_{n \rightarrow \infty} \frac{\|\mathbf{x}_{n+1} - \mathbf{x}_n\|_2}{\|\mathbf{x}_n - \mathbf{x}_{n-1}\|_2}.$$

Obviously, we don't have an infinite number of terms of  $\{\mathbf{x}_n\}$ . The best we can do is approximate  $\mu$  by the ratios of our finite sequence.

*LISTING 3: Computing the sequence of ratios*

---

```

1 approx = list(solve(F, (1000, 0), tol=1E-15))
2
3 def pairs(L):
4     yield from zip(L[1:], L)
5
6 def diffs(L):
7     for (xn, yn), (xm, ym) in pairs(L):
8         yield (xn - xm, yn - ym)
9
10 def norms(vs):
11     for v in vs: yield LA.norm(v)
12
13 norms_of_diffs = list(norms(diffs(approx)))
14
15 for p, q in pairs(norms_of_diffs):
16     print(p/q**2)

```

---

Which produces the following values. Note that we need to use a very low tolerance, otherwise we will not see anything resembling convergence at all.

TABLE I: *The sequence  $\{\mu_n\}$  of ratios of error.*

Iteration	$\mu_n$
1	0.001 414
2	0.002 828
3	0.005 656
4	0.011 309
5	0.022 596
6	0.045 009
7	0.088 582
8	0.166 701
9	0.272 763
10	0.341 980
11	0.353 357
12	0.353 553
13	0.354 073

It seems like the sequence settles on  $\mu \approx 0.35$ , which indicates quadratic convergence.

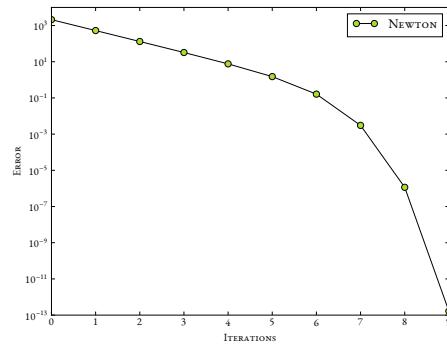


FIGURE I: *Convergence of Newton's method with the norm  $\|\mathbf{F}(x, y)\|_2$*

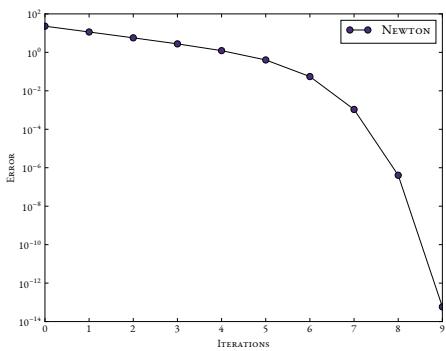


FIGURE 2: Convergence of Newton's method with the norm  $\|\mathbf{x}_{k+1} - \mathbf{x}_k\|_2$

The figures indicate that the convergence is at least superlinear. It is not easy to read from an image exactly how fast the convergence is, but with the estimated  $\mu$ , quadratic convergence seems likely.

### CONVERGENCE ALONG DIAGONALS

We want to see what happens as the method converges for two initial guesses that converges to different solutions.

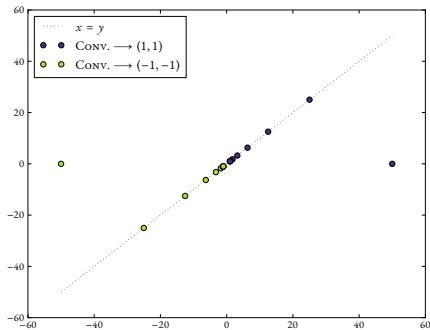


FIGURE 3: Convergence of Newton's method along the line  $x = y$ , starting at  $\mathbf{x} = (-50, 0)$  and  $\mathbf{x} = (50, 0)$ .

As we can see, the solutions instantly “jump” to the  $x = y$  diagonal, and converges along it.

### OPTIONAL PROBLEM

Given

$$f(z) = z^3 - 1$$

as a function  $f : \mathbb{C} \rightarrow \mathbb{C}$ , we want to inspect the basins of attraction for Newton's method. We can view  $f$  as a function  $f : \mathbb{R}^2 \rightarrow \mathbb{R}^2$  instead, and use the solver we already have.

For every point in the complex plane, we are interested in which of the three roots Newton's method converges to.

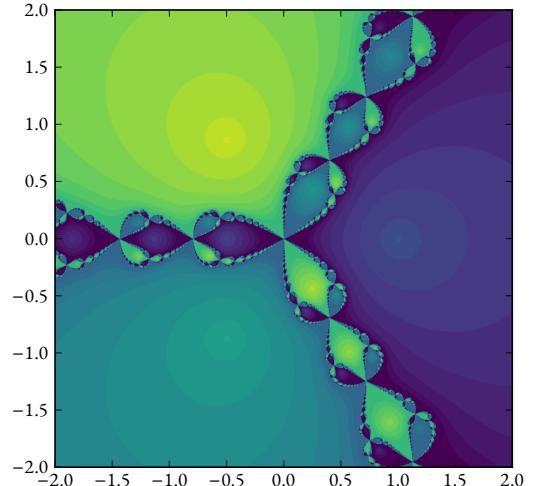


FIGURE 4: Basins of attraction for Newton's method applied on  $f(z) = z^3 - 1$ .

FIGURE 4 shows how the convergence forms three “basins” in a beautiful fractal pattern.

## PROBLEM 2

We want to consider the linear system

$$A\mathbf{u} = \mathbf{f},$$

where

$$A = (L + (\Delta x)^2 k^2 I)$$

is a matrix in  $\mathbb{R}^{n^2 \times n^2}$ , and  $\Delta x = 1/n$ . Notice that  $A$  is an operator that operates on vectors in  $\mathbb{R}^{n^2}$ , corresponding to an  $n \times n$  lattice in a domain  $\Omega$ :

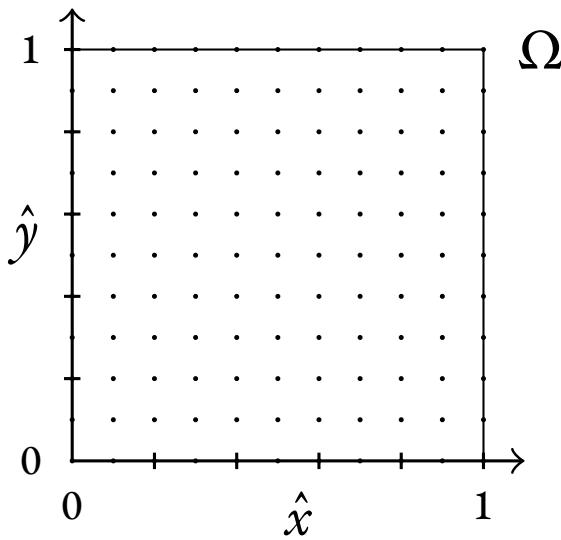


FIGURE 5: Lattice points in  $\Omega$ .

For any function  $f(x, y)$  defined on  $\Omega$ , we can let

$$\mathbf{f} = (f_1 \ f_2 \ \dots \ f_l \ \dots \ f_{n^2}) ; \quad f_l = f(x_i, y_i),$$

where

$$\begin{cases} x_i = i \cdot \Delta x \\ y_j = j \cdot \Delta x \\ l = (j - 1)n + i \end{cases}$$

which corresponds exactly to “sampling”  $f$  at the lattice points:  $x_i = x_i(i)$  and  $y_j = y_j(j)$  maps integer indeces  $i$  and  $j$  to lattice points bijectively.  $l = l(i, j)$  maps the same indeces to an index into a vector in  $\mathbb{R}^{n^2}$  bijectively. Thus, there is a bijective correspondance between lattice points and vector components. If we consider a concrete function

$$f(x, y) := \exp\left(-50\left(\left(x - \frac{1}{2}\right) + \left(y - \frac{1}{2}\right)\right)\right)$$

defined on  $\Omega$ , the sampled data looks something like FIGURE 6 for a relatively low choice of  $n$ .

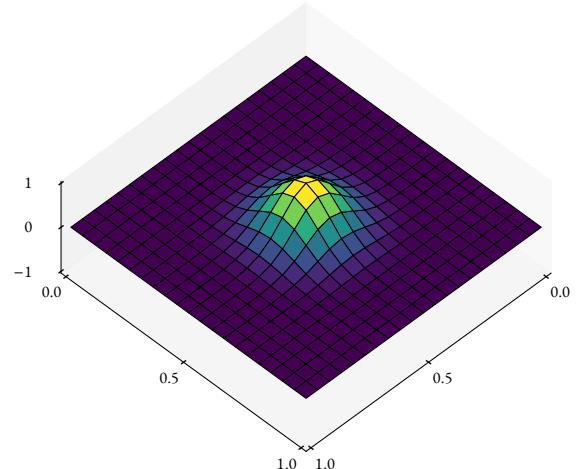


FIGURE 6:  $f(x, y)$  sampled across the lattice in  $\Omega$ .

This sampling can be done succinctly in python, using a generator function:

LISTING 4: Program to sample functions over lattices

---

```

1 def lattice(n):
2     for j in range(n):
3         for i in range(n):
4             yield i/(n - 1), j/(n - 1)
5
6 def sample(F, n):
7     # samples F over a n x n lattice.
8     return np.array(

```

```

9         [F(x,y) for x, y in lattice(n)]
10    )

```

sample( $F$ ,  $n$ ) produces exactly the vector  $\mathbf{f}$ , for a given function  $F$ , sampled over a lattice in  $\Omega$  with  $\Delta x = 1/n$ .

To solve the system  $A\mathbf{u} = \mathbf{f}$ , we want to use fix-point iteration:

$$\begin{aligned} A\mathbf{u} &= \mathbf{f} \\ (M - N)\mathbf{u} &= \mathbf{f} \\ M\mathbf{u} &= N\mathbf{u} + \mathbf{f} \\ \mathbf{u} &= M^{-1}(N\mathbf{u} + \mathbf{f}) \end{aligned}$$

in this equation,  $M^{-1}N$  and  $M^{-1}\mathbf{f}$  are constants. It is sensible (and more efficient) to compute them once up-front and give them names. This gives the more orderly iteration

$$\mathbf{u} \leftarrow C\mathbf{u} + \mathbf{g},$$

which converges by the Banach fixed-point theorem if  $\mathbf{x} \mapsto C\mathbf{u} + \mathbf{g}$  is a contraction, which is the case as long as

$$0 < \rho(C) := \max_{\lambda \text{ eig. of } C} \lambda < 1.$$

In python, this can be implemented as follows

LISTING 5: *N-dimensional solver.*

```

1  # src/alg.py
2  def spectral_radius(M):
3      return np.max(np.abs(LA.eigvals(M)))
4
5  def solve_nd_fpi(M, N, f):
6      # solves the linear system  $(M - N)\mathbf{u} = b$  by
7      # fix-point iteration  $\mathbf{u} = \text{inv}(M)(N\mathbf{u} + b)$ .
8
9      Mi = LA.inv(M)
10     C = Mi.dot(N)
11     g = Mi.dot(f)
12
13     assert spectral_radius(C) < 1
14

```

```

15     u = g
16     for _ in range(MAX_ITER):
17         u = C.dot(u) + g
18     yield u

```

provided we already have a choice of  $M$  and  $N$ . Several ways to choose these matrices are possible, and we want to be able to choose.

LISTING 6: *Argument-“parser” and choice of  $M, N$ .*

```

1  # src/alg.py
2
3  def jacobi_mat(A):
4      # Jacobi method
5      M = np.diag(A.diagonal())
6      N = M - A
7      return M, N
8
9  def gs_mat(A):
10     # Gauss-Seidel
11     M = np.tril(A)
12     N = M - A
13     return M, N
14
15 def sor_mat(A, omega):
16     # successive over-relaxation
17     D = np.diag(A.diagonal())
18     L = np.tril(A, k=-1)
19     M = D + omega*L
20     N = M - A
21     return M, N
22
23 def choose_matrices(A, method="jacobi", omega=1.0):
24     # pick a method based on the parameter
25     # i have included some redundant parameters
26     # so it is possible to write "shorthand"
27     # upper-case also works.
28     M, N = {
29         # Jacobi method
30         "jacobi": jacobi_mat,
31         "j": jacobi_mat,
32         # Gauss-Seidel method
33         "gauss-seidel": gs_mat,
34         "gs": gs_mat,
35         # Successive over-relaxation
36         "sor": lambda A: sor_mat(A, omega),
37     }[method.lower()](A)
38
39     return M, N
40
41

```

---

```

42 def solve_nd(A, f, method="jacobi", omega=1.0):
43     # solve Ax = b.
44     # returns an iterator over tuples (u, r),
45     # where u is successively better solutions,
46     # and r is the residual vector.
47     # omega is unused unless "sor" is specified
48
49     M, N = choose_matrices(A,
50                            method=method,
51                            omega=omega)
52
53     for u in solve_nd_fpi(M, N, f, tol=tol):
54         # compute the residual
55         r = f - A.dot(u)
56         yield u, r

```

---

Now, given some matrix  $M$ , and some vector  $\mathbf{b}$ , we can solve the system  $M\mathbf{u} = \mathbf{b}$ . Notice that, again, the system is implemented in such a way that it produces successively better and better approximations. It also computes a residual vector which it gives us along with each approximation as a tuple  $(\mathbf{u}, \mathbf{r})$ . This makes the API a little clunky if all we want to do is compute the solution of some system, but it makes it easy to work with the data as a sequence. If all we want is the solution, we need the left-hand element of the last tuple.

### TESTING THE SOLVER FOR A TRIVIAL PROBLEM

We want to make sure our code is correct for a simple problem, just so that we can have *some* confidence that it actually works. Solving a simple system, which we know has a solution, such as

$$\begin{cases} 3x - y = 1 \\ 2x + 2y = 0 \end{cases}$$

would give us a good hint about possible errors we might have made. (By insertion it is easy to verify that  $x = 1/4$  and  $y = -1/4$  is a solution)

---

```

1  # interactive session
2  >>> from src.alg import solve_nd, last
3  >>> import numpy as np
4
5  >>> A = np.array([[3, -1],
6  ...                  [2,  2]])
7
8  >>> b = np.array([1, 0])
9
10 >>> u, v = last(solve_nd(A, b, method="Jacobi"))
11 >>> u
12 array([ 0.25, -0.25])
13
14 >>> u, v = last(solve_nd(A, b, method="GS"))
15 >>> u
16 array([ 0.25, -0.25])
17
18 >>> u, v = last(solve_nd(A, b, method="SOR"))
19 >>> u
20 array([ 0.25, -0.25])

```

---

As we can see, all the methods give the correct solution.

### SOLVING THE ORIGINAL PROBLEM

We can find a solution to our original problem, fixing  $n = 10$  and  $k = 1/100$ , by setting up

---

```

1  from src.alg import solve_nd, sample
2  from asc.alg import A as gen_A
3  import numpy as np
4
5  n = 10
6  k = 1/100
7
8  def F(x,y):
9      return np.exp(
10          -50 * ((x - 1/2)**2 + (y - 1/2)**2)
11      )
12
13 f = sample(F, n)
14 A = gen_A(k, n)

```

---

Here, `src.alg.A` is an algorithm provided for us that generates  $A$ .  $f$  is the vector obtained when sampling  $F := f(x, y)$  across the lattice defined by  $n = 10$ . The `sample`-function is the same one we developed earlier.

Given these structures, we are interested in comparing the performance of each of the three methods when solving the system  $A\mathbf{u} = \mathbf{f}$ .

### i. COMPARING THE CONVERGENCE

We are interested in looking at the relative residual  $\|r_n\|_2 / \|r_0\|_2$  for each iteration  $n$  of the algorithms. These can be computed with the following Python-program:

```

1 def right(it):
2     for _, x in it: yield x
3
4 def relative_residual(appr):
5     # we need a list, because
6     # we can't peek iterators
7     assert type(appr) == list
8     a = LA.norm(appr[0][1])
9     for r in right(iterator(appr)):
10        yield LA.norm(r) / a

```

FIGURE 7 shows the relative residuals for each of the methods, including three different values for  $\omega$  in the case of successive over-relaxation.

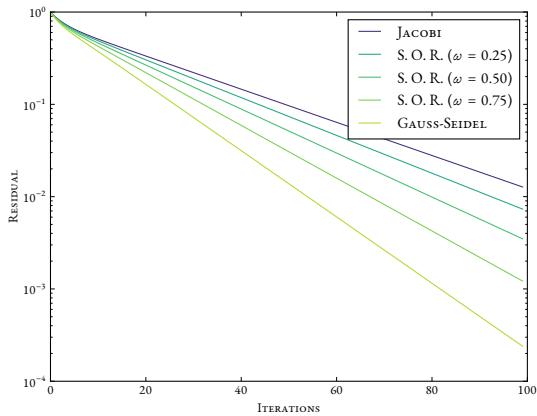


FIGURE 7: The residuals of each method for a given iteration.

### ii. COMPARING THE RELATIVE TIME

### iii. SPECTRAL RADIUS AND CONVERGENCE

Considering FIGURE 7, it is quite clear that after a certain number of iterations, the convergence is linear. In hindsight, this is not so surprising. Consider  $\mathbf{u}$  as a linear combination of an eigen-basis  $\beta = \{v_i\}$  given by the eigenvectors of  $C$ . Then the operation of  $C$  on

$$\mathbf{u} = \sum_i \alpha_i \cdot v_i$$

is to scale each component of  $\mathbf{u}$  by the corresponding eigenvalue:

$$\begin{aligned} C\mathbf{u} &= \sum_i \lambda_i \cdot \alpha_i \cdot v_i \\ \implies C^n \mathbf{u} &= \sum_i \lambda_i^n \cdot \alpha_i \cdot v_i. \end{aligned}$$

Since  $0 < \rho(C) < 1$ , each component converges by itself, so the sequence converges in its entirety, but *only as fast as the slowest component converges!* Naturally, the slowest converging component converges linearly, with  $\mu = \rho(C)$ .

Using Python, we can easily compute the spectral radiiuses of the iteration matrices.

TABLE 2: The spectral radiiuses of the iteration matrices for each method

Algorithm	Spectral Radius
Jacobi	3.999 999
Gauss-Seidel	3.999 999
S. O. R. ( $\omega = 0.25$ )	3.999 999
S. O. R. ( $\omega = 0.50$ )	3.999 999
S. O. R. ( $\omega = 0.75$ )	3.999 999

TABLE 2 reveals that they all have the same asymptotic rate of convergence.

iv. RELATION BETWEEN  $k$ ,  $\Delta x$  AND  
CONVERGENCE