STEFFEN HAUG

Numerical Methods

Assignment 2

Problem 1

The goal is to implement Newtons method for a generic function $\mathbf{F} : \mathbb{R}^2 \longrightarrow \mathbb{R}^2$. I will use a symbolic computation library to compute the Jacobian $\mathbf{J_F}$, 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*

```python
# file: src/alg.py
import numpy as np
from numpy import linalg as LA
import sympy as sp

from numpy import linalg as la
import scipy.sparse as scsp


def symbolic_jac(py_fn):
    # Computes a symbolic jacobian matrix
    # for a function f : R^2 -> R^2.

    # compute the entries of the vector by
    # evaluating the function for sp-symbols
    x, y = sp.symbols('x y')
    f1, f2 = py_fn(x, y)
    F  = sp.Matrix([f1, f2])

    return F.jacobian([x, y])


def callable_fn(symbolic):
    # Create a function that substitutes
    # for the symbolic values.
    x, y = sp.symbols('x y')
    return sp.lambdify(
        [x, y], symbolic, 'numpy'
    )
```

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

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

Listing 2: *Newton's method*

```python
# file: src/alg.py
MAX_ITER = 100

def solve(F, x0, tol=1E-6):
    x, y = x0
    assert x + y != 0, "Singular Jacobian!"

    # Compute the jacobian symbolically
    J  = symbolic_jac(F)
    # Make a "callable" inverse
    Ji = callable_fn(J.inv())

    def step(f, Ji_f, r):
        # computes the next step using
        # the Newton method equation.
        return r - Ji_f(*r).dot(f(*r))


    for _ in range(MAX_ITER):
        px, py = x, y
        x,  y  = step(F, Ji, (x, y))
        yield x, y

        # check the tolerance criteria
        if la.norm(F(x, y)) < tol:
            break
        if la.norm((x - px, y - py)) < tol:
            break

def last(it):
    # run an iterator to the end
    x = None
    for x in it: pass
    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 resposibility 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

$$\mu = \lim_{n\longrightarrow\infty} \frac{\|\mathbf{x}_{n+1} - \mathbf{x}_n\|_2}{\|\mathbf{x}_n - \mathbf{x}_{n-1}\|_2^2}.$$

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

Listing 3: *Computing the sequence of ratios*

```
approx = list(solve(F, (1000, 0), tol=1E-15))

def pairs(L):
    yield from zip(L[1:], L)

def diffs(L):
    for (xn, yn), (xm, ym) in pairs(L):
        yield (xn - xm, yn - ym)

def norms(vs):
        for v in vs: yield LA.norm(v)

norms_of_diffs = list(norms(diffs(approx)))

for p, q in pairs(norms_of_diffs):
    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 1: *The sequence $\{\mu_n\}$ of ratios of error.*

Iteration	μ_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.

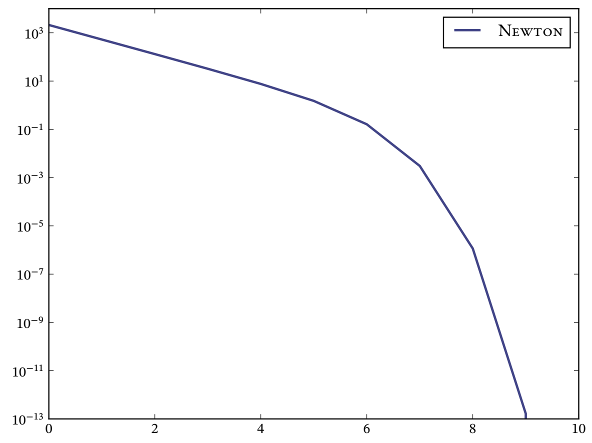

Figure 1: *Convergence of Newton's method with the norm $\|\mathbf{F}(x,y)\|_2$*

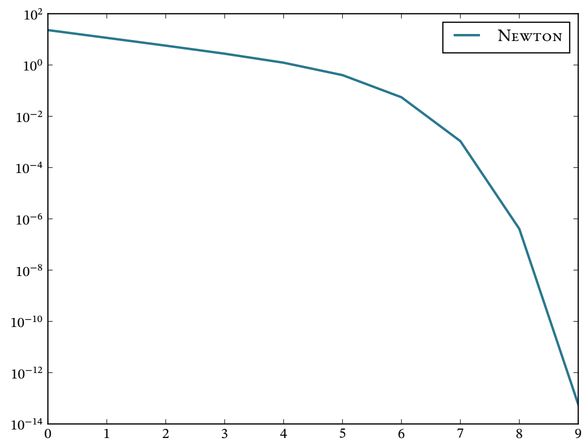

FIGURE 2: *Convergence of Newton's method with the norm* $\|\mathbf{x}_{k+1} - \mathbf{x}_k\|_2$

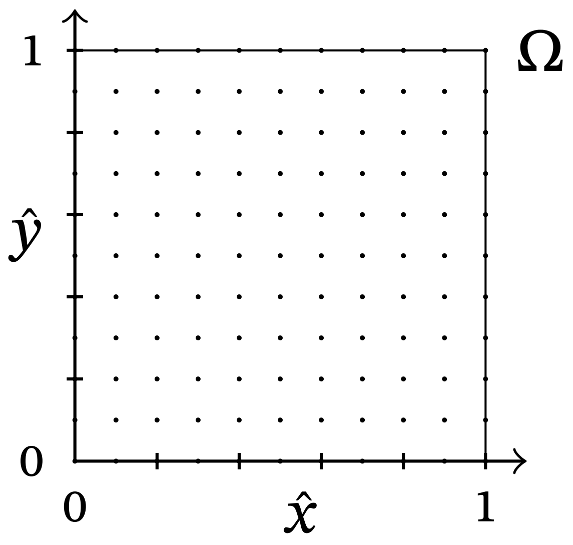

FIGURE 3: *Lattice points in* Ω.

Optional problem

Problem 2

We want to consider the linear system

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

where

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

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 Ω:

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

$$\mathbf{f} = \begin{pmatrix} f_1 & f_2 & \cdots & f_l & \cdots & f_{n^2} \end{pmatrix}; \quad f_l = f(x_i, y_i),$$

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: $x(i)$ and $y(j)$ maps integer indeces i and j to lattice points. $l = l(i, j)$ maps the same indeces to an index into a vector in $\mathbb{R}^{n^2}$. 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 Ω, the sampled data looks something like FIGURE 4 for a low choice of n.

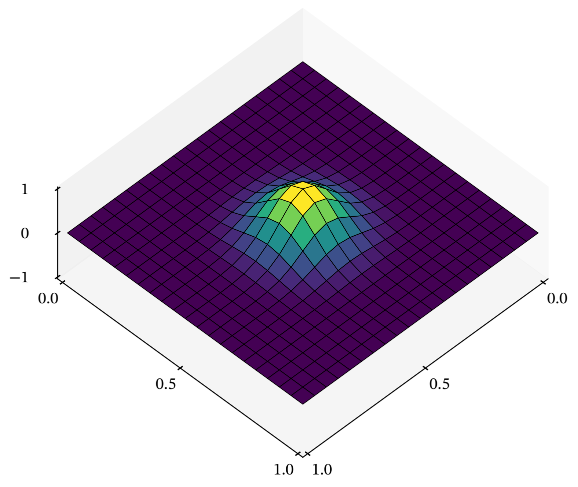

FIGURE 4: *$f(x,y)$ sampled across the lattice in Ω.*

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

LISTING 4: *Program to sample functions over lattices*

```
def lattice(n):
    for j in range(n):
        for i in range(n):
            yield i/(n - 1), j/(n - 1)

def sample(F, n):
    # samples F over a n x n lattice.
    return np.array(
        [F(x,y) for x, y in lattice(n)]
    )
```

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

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

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

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

$$\mathbf{u} = C\mathbf{u} + \mathbf{g}\,.$$

Which converges by the Banach fixed-point theorem if $\mathbf{x} \mapsto C\mathbf{x} + \mathbf{g}$ is a contraction, which is the case as long as the spectral radius of C, $\rho(C) < 1$. In python, this can be implemented as follows

LISTING 5: *N-dimensional solver.*

```
def spectral_radius(M):
    return np.max(np.abs(LA.eigvals(M)))

def solve_nd_fpi(M, N, f, tol=1E-6):
    # solves the linear system (M - N)x = f by
    # fix-point iteration x = inv(M)(Nx + b).

    # Compute inv(M), C and f
    Mi = LA.inv(M)
    C = Mi.dot(N)
    g = Mi.dot(f)

    assert spectral_radius(C) < 1, "rho(C) > 1"

    x = g
    for _ in range(MAX_ITER):
        x = C.dot(x) + b

    return x
```

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.*

```
def solve_nd(A, b, tol=1E-6,
        method="jacobi", omega=1):
    # solve Ax = b.
    # omega is only used if you choose
    # the method successive over-relaxation

    def jacobi_mat(A):
        # Jacobi method
        M = np.diag(A.diagonal())
        N = M - A
        return M, N

    def gs_mat(A):
        # Gauss-Seidel
        M = np.tril(A)
        N = M - A
        return M, N

    def sor_mat(A, omega):
        # successive over-relaxation
        D = np.diag(A.diagonal())
        L = np.tril(A, k=-1)
        M = D + omega*L
        N = M - A
        return M, N

    # pick a method based on parameter.
    # i have included some redundant parameters
    # so it is possible to write "shorthand"
    M, N = {
        "jacobi":        jacobi_mat,
        "j":             jacobi_mat,
        "gauss-seidel":  gs_mat,
        "gs":            gs_mat,
        "sor": lambda A: sor_mat(A, omega),
    }[method.lower()](A)

    x = solve_nd_fpi(M, N, b, tol=tol)

    return x
```

Now, given some matrix A, and some vector $\mathbf{b}$, we can solve the system $A\mathbf{x} = \mathbf{b}$.

Testing the solver for a trivial problem

We want to make sure our code is correct by first solving a simple system, which we know has a solution, such as

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

```
>>> from src.alg import solve_nd
>>> import numpy as np

>>> A = np.array([[3, -1],
...               [2,  2]])

>>> b = np.array([1, 0])

>>> solve_nd(A, b, method="Jacobi")
array([ 0.25, -0.25])
>>> solve_nd(A, b, method="Gauss-Seidel")
array([ 0.25, -0.25])
>>> solve_nd(A, b, method="SOR")
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

```
from src.alg import solve_nd, sample
from asc.alg import A as gen_A
import numpy as np

n = 10
k = 1/100

def F(x,y):
    return np.exp(-50 * ((x - 1/2)**2 + (y - 1/2)**2))

f = sample(F, n)
A = gen_A(k, n)

u = solve_nd(A, f)
```

Here, `src.alg.A` is an algorithm provided for us that generates A.