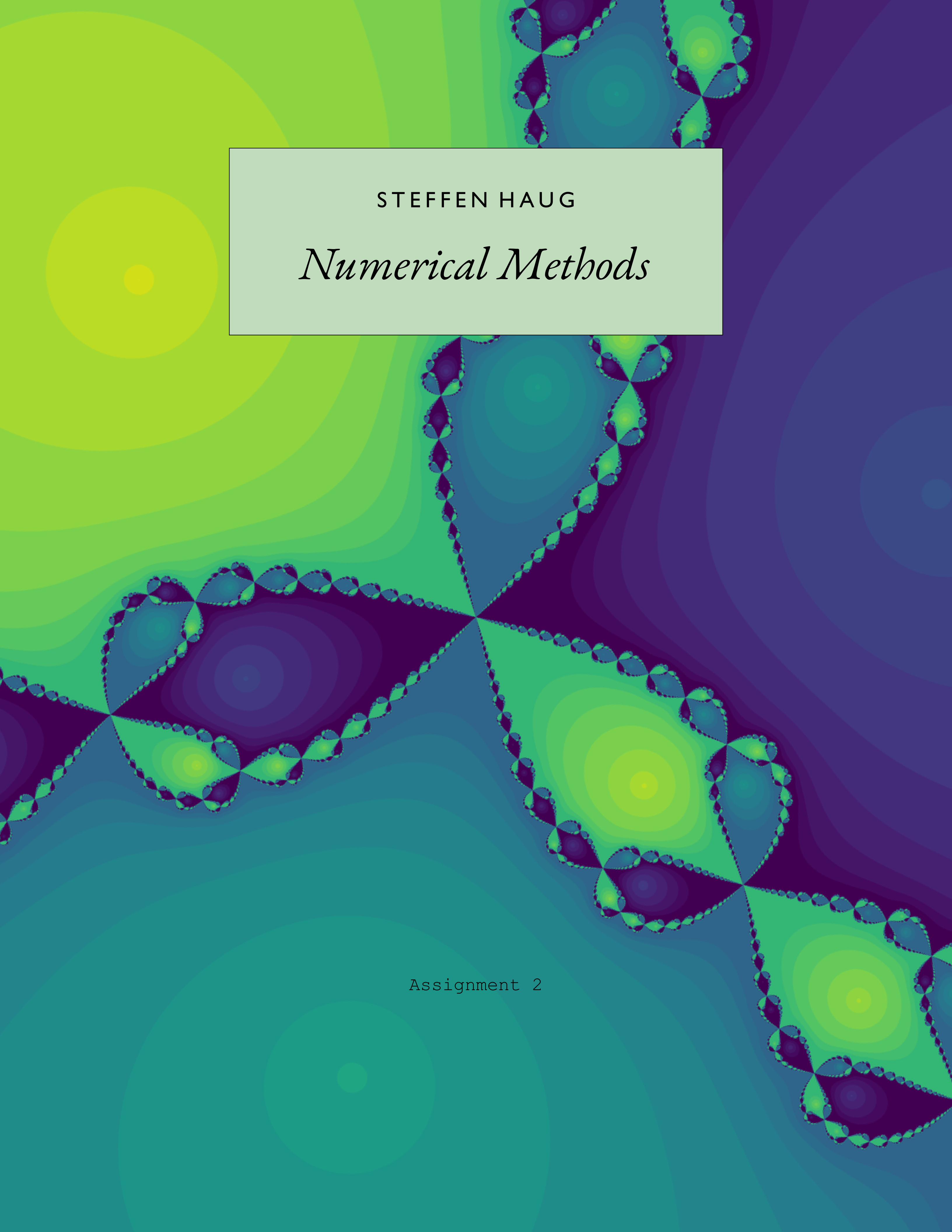

STEFFEN HAUG

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. I have attempted to abstract away the `sympy`-code, because it is just boilerplate that pollutes the interesting part.

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
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

    J = symbolic_jac(F)

    # singular jacobian means trouble
    Jfn = callable_fn(J)
    assert LA.det(Jfn(x, y)) != 0

    # function-version of the Jacobian
    Ji = callable_fn(J.inv())

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

    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.

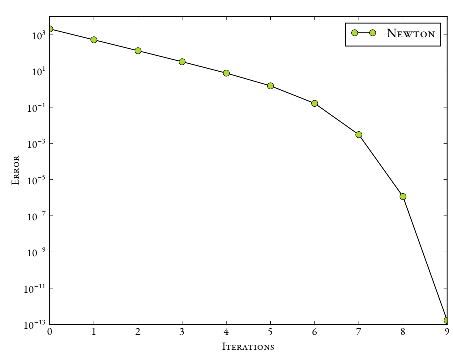

Figure 1: *Convergence of Newton's method with the norm $\|\mathbf{F}(x, y)\|_2$*

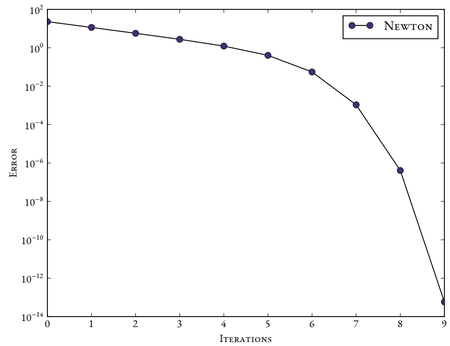

Figure 2: *Convergence of Newton's method with the norm* $\|\mathbf{x}_{k+1} - \mathbf{x}_k\|_2$

The figures indicate that the convergence is at least super-linear. It is not easy to read from an image exacly how fast the convergence is, but with the estimated μ, 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.

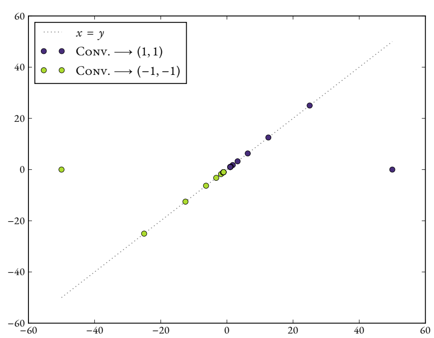

Figure 3: *Convergence of Newton's method along the line* $x = y$*, starting at* $\mathbf{x} = (-50, 0)$ *and* $\mathbf{x} = (50, 0)$.

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 $f : \mathbb{C} \longrightarrow \mathbb{C}$, we want to inspect the basins of attraction for Newton's mehod. We can view f as a function $f : \mathbb{R}^2 \longrightarrow \mathbb{R}^2$ 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.

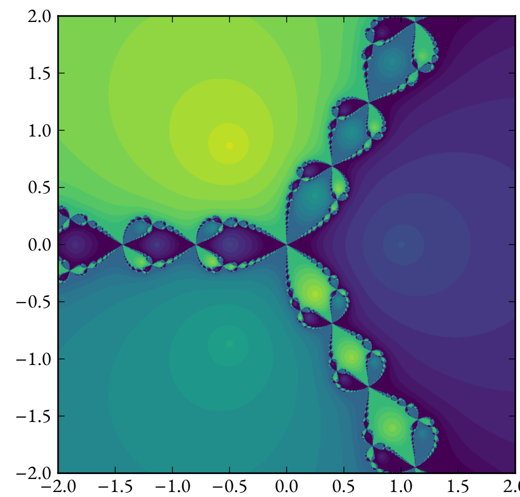

Figure 4: *Basins of attraction for Newton's method applied on* $f(z) = z^3 - 1$.

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

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

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:

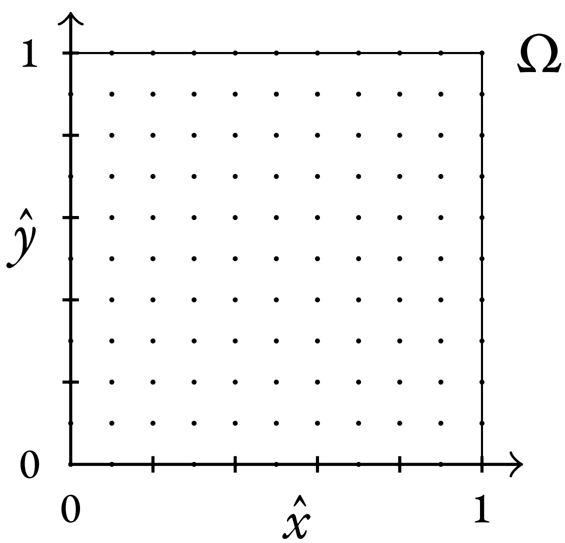

Figure 5: *Lattice points in* Ω.

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

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

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 Ω, the sampled data looks something like figure 6 for a relatively low choice of n.

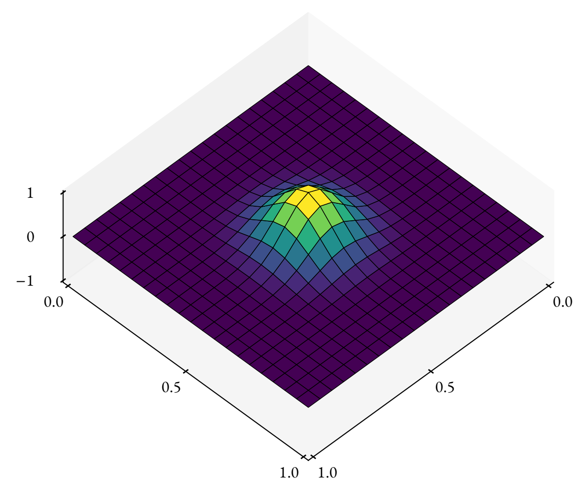

Figure 6: $f(x, y)$ *sampled across the lattice in* Ω.

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

Listing 4: *Program to sample functions over lattices*

```python
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:

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

in this equation, $M^{-1}N$ and $M^{-1}\mathbf{f}$ are constants. It is sensible (and more efficient) to compute them once upfront 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.*

```
# src/alg.py
def spectral_radius(M):
    return np.max(np.abs(LA.eigvals(M)))

def solve_nd_fpi(M, N, f):
    # solves the linear system (M - N)u = b by
    # fix-point iteration u = inv(M)(Nu + b).

    Mi = LA.inv(M)
    C  = Mi.dot(N)
    g  = Mi.dot(f)

    assert spectral_radius(C) < 1

    u = g
    for _ in range(MAX_ITER):
        u = C.dot(u) + g
        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.*

```
# src/alg.py

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

def choose_matrices(A, method="jacobi", omega=1.0):
    # pick a method based on the parameter
    # i have included some redundant parameters
    # so it is possible to write "shorthand"
    # upper-case also works.
    M, N = {
        # Jacobi method
        "jacobi":        jacobi_mat,
        "j":             jacobi_mat,
        # Gauss-Seidel method
        "gauss-seidel":  gs_mat,
        "gs":            gs_mat,
        # Successive over-relaxation
        "sor": lambda A: sor_mat(A, omega),
    }[method.lower()](A)

    return M, N
```

```
def solve_nd(A, f, method="jacobi", omega=1.0):
    # solve Ax = b.
    # returns an iterator over tuples (u, r),
    # where u is successively better solutions,
    # and r is the residual vector.
    # omega is unused unless "sor" is specified

    M, N = choose_matrices(A,
                           method=method,
                           omega=omega)

    for u in solve_nd_fpi(M, N, f):
        # compute the residual
        r = f - A.dot(u)
        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 succesively better and better approximations. It also computes a residual vector which it gives us along with each approximation as a tuple `(u, 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)

```
# interactive session
>>> from src.alg import solve_nd, last
>>> import numpy as np

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

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

>>> u, v = last(solve_nd(A, b, method="Jacobi"))
>>> u
array([ 0.25, -0.25])

>>> u, v = last(solve_nd(A, b, method="GS"))
>>> u
array([ 0.25, -0.25])

>>> u, v = last(solve_nd(A, b, method="SOR"))
>>> u
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

```
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)
```

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:

```
def right(it):
    for _, x in it: yield x

def relative_residual(appr):
    # we need a list, because
    # we can't peek iterators
    assert type(appr) == list
    a = LA.norm(appr[0][1])
    for r in right(iter(appr)):
        yield LA.norm(r) / a
```

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

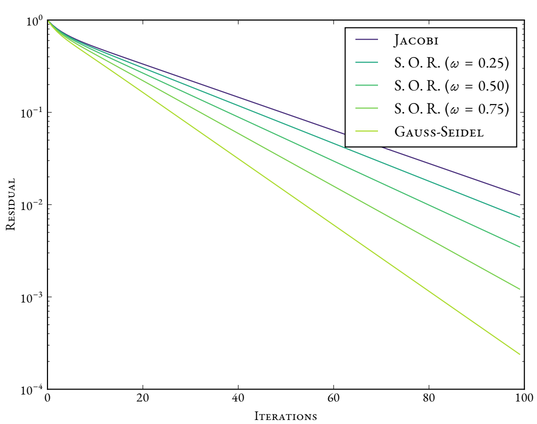

Figure 7: *The residuals of each method for a given iteration.*

ii. 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

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

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

$$C\mathbf{u} = \sum_i \lambda_i \cdot a_i \cdot v_i$$

$$\implies C^n\mathbf{u} = \sum_i \lambda_i^n \cdot a_i \cdot v_i.$$

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 radiuses of the iteration matrices.

Table 2: *The spectral radiuses of the iteration matrices for each method*

Algorithm	Spectral Radius
Jacobi	0.9594
Gauss-Seidel	0.9206
S. O. R. ($\omega = 0.25$)	0.9539
S. O. R. ($\omega = 0.50$)	0.9465
S. O. R. ($\omega = 0.75$)	0.9362

TABLE 3 reveals that they all have similar rates of convergence, but the successive over-relaxation method seems to get better as ω increases. This motivates a more systematic search for ω.

By searching by brute force in $I = [0, 100]$, it is revealed that $\omega \approx 1.6$ gives $\rho \approx 0.74$. This is a promising result, so we search in that general area ($I = [1, 2]$). We find that $\omega \approx 1.5628$ gives $\rho \approx 0.7198$, which is a much faster converging approximation than any of the originals.

iii. Comparing the relative time

Now that we know (at least we think we know) the optimal choice of ω, it makes sense to benchmark all three algorithms. The solver i presented earlier does not return early based on a tolerance-criterium, because letting them all run for exactly the same number of iterations makes nicer plots and easier (to read) code. Now that we want the algorithms to compete, we don't want to naïvely run them all for the same amount of iterations. (That would be a pretty dumb benchmark) We can use the norm of the residual vector, $\|\mathbf{f} - A\mathbf{u}\|_2$ as a measure of error, and return whenever this is close to zero. Since we want to *really* pressure the algorithms, it makes sense to use a very large value for `MAX_ITER` (i used 5000) and use a low tolerance (i used $1 \cdot 10^{-10}$)

To benchmark the algorithms, i will make them solve the same problem in a long loop. This is simple, because we already have the code to setup the problem we looked at previously, but it is not a very sophisticated way to benchmark programs.

Table 3: *The time it takes to run 1000 iterations.*

Algorithm	**Time (μs)**	**# Iterations**
Jacobi	13718915	552
Gauss-Seidel	9938928	276
S. O. R. ($\omega = 1.5628$)	5126427	77

Sure enough, the Gauss-Seidel algorithm runs faster than the Jacobi method, and the succesive over-relaxation algorithm with optimal ω is way faster than both of them. Since this is exactly what we expected, the benchmark is probably not too far off.

iv. Relation between k, Δx and convergence