Problem Sheet 6

Exercise 1.1

To derive the numeric form of the equation on the grid, we use the example given in the script with Dirichlet boundary condition:

$$-D\frac{T_{i+1}-2T_i+T_{i-1}}{h^2}=\epsilon \tag{1}$$

In matrix notation, this reads:

$$\begin{pmatrix} 1 & & & & \\ 1 & -2 & 1 & & \\ & & \ddots & & \\ & & 1 & -2 & 1 \\ & & & & 1 \end{pmatrix}\begin{pmatrix} T_1 \\ \vdots \\ T_i \\ \vdots \\ T_N \end{pmatrix}=\begin{pmatrix} T_0 \\ -\dfrac{\epsilon h^2}{D} \\ \vdots \\ -\dfrac{\epsilon h^2}{D} \\ T_0 \end{pmatrix} \tag{2}$$

Th following equation initializes the elements of the sparse matrix from (2) using three 1D arrays with N elements:

```python
def initialize_matrix(N):
  diagonal = np.ones((N, 1))
  upper = np.ones((N, 1))
  lower = np.ones((N, 1))

  diagonal = np.multiply(diagonal, -2)
  diagonal[0] = 1
  diagonal[N-1] = 1
  lower[N-1] = 0
  upper[0] = 0
  return lower, diagonal, upper
```

Function to initialize the 3D array. Notice that the array for the diagonal part contains one element more than the off diagonal lower and upper part. The corresponding entry in the array is set to one but ignored in any further computation.

The matrix multiplication with the sparse matrix is realized through:

```python
# Function to multiply a trigonal matrix and a vector
def trigonal_multiplication(lower, diagonal, upper, vector):
  result = np.zeros((N, 1))
  result[0] = diagonal[0] * vector[0]
  result[N-1] = diagonal[N-1] * vector[N-1]
  for i in range(1, N-1):
    result[i] = lower[i] * vector[i-1] + diagonal[i] * vector[i] + upper[i]
        * vector[i+1]
  return result
```

Function to multiply the given sparse matrix with a vector.

Testing this algorithm with an input vector equal to one yields the output:

```
[[1.][0.][0.][0.][0.][0.][0.][0.][1.]]
```

Listing:

This is exactly what we would expect. Thus, we are fine to assume that the algorithm works correctly. We now turn to the implementation of the forward-elimination backward-substitution algorithm. We splitted this into two parts:

```python
# Function that implements the forward-elimination, backward-substitution
    algorithm
def forward_elimination(lower, diagonal, upper, b):
  for i in range(0, N-2):
    if diagonal[i] > 0:
      m = (-1) * np.abs(lower[i+1]/diagonal[i])
    else:
      m = np.abs(lower[i + 1] / diagonal[i])
    lower[i+1] = lower[i+1] + m * diagonal[i]
    diagonal[i+1] = diagonal[i+1] + m * upper[i]
    b[i+1] = b[i+1] + m * b[i]
  return diagonal, b
```

Function for the forward elimination step.

This function only returns the diagonal and the b-part. The lower will be equal to zero after performing the procedure while the upper part will still have the same shape. Thus they are not returned by the algorithm. Now we turn to the backward-substitution step:

```python
# Function that implements the backward-substitution step
# The output vector contains the values for T_i
def backward_substitution(diagonal, b):
  result = np.zeros((N, 1))
  result[N-1] = b[N-1]
  result[0] = b[0]
  for i in range(2, N):
    result[N-i] = (b[N-i] - result[N-i+1])/diagonal[N-i]
  return result
```

Function for the backward substitution step.

To run the algorithm, we use:

```python
# Function to run the full algorithm
def run_matrix_algorithm():
  l, d, u = initialize_matrix(N)
  b = initialize_b(T_0)
  a, b = forward_elimination(l, d, u, b)
  y = backward_substitution(a, b)

  # Verify solution and calculate residual
  l, d, u = initialize_matrix(N)
  b = initialize_b(T_0)
  print('Residual:', trigonal_multiplication(l, d, u, y) - b)
```

```
x = create_x_array()
plt.plot(x, y, 'k--')
```

Runs the forward-backward algorithm and plots the result and also calculates the residual.

The plot of the solution looks like: This looks quite similar to a parabola, which is what

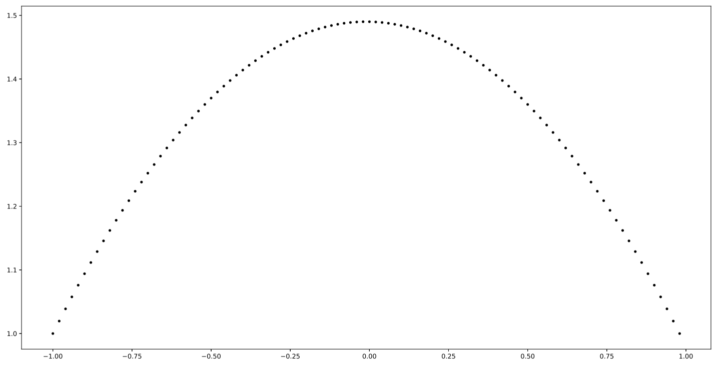

Figure 1: Plot of the solution for $N = 100$

we would expect as a solution to the partial differential equation. The residual does not vanish for the the given problem. Instead it is of order 10^{-16} to 10^{-17}. This is probably due to finite machine precision when multiplying large numbers with small numbers. Here we can see the same result for $N = 1000$:

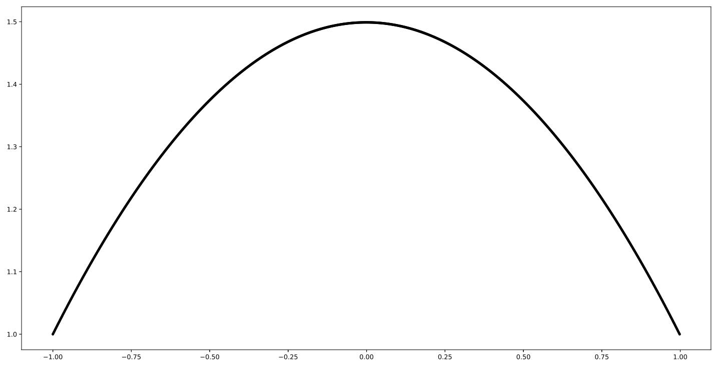

Figure 2: Plot of the solution for $N = 1000$

We see that the solution is much smoother for higher numbers of N.

Next, we implement the Jacobi iteration. To do this we start by writing a function that does one Jacobi step per call:

```python
# Function that implements a Jacobi iteration
def jacobi_step(lower, diagonal, upper, x, b):
  zero = np.zeros((N, 1))
  p = trigonal_multiplication(lower, zero, upper, x)
  x = trigonal_multiplication(zero, 1./diagonal, zero, b + p)
  return x
```

Function that performs one Jacobi step.

To run this function, we use the following piece of code:

```python
# Function to run the full jacobi algorithm
def run_jacobi_algorithm(steps):
  x = create_x_array()
  l, d, u = initialize_matrix(N)
  b = initialize_b(T_0)
  y = np.zeros((N, 1))
  for i in range(0, steps):
    y = jacobi_step((-1) * l, d, (-1) * u, y, b)
    plt.plot(x, y)
```

Function to perform several Jacobi steps and plots the result for each step.

Result for $N = 8$ grid points and 30 steps:

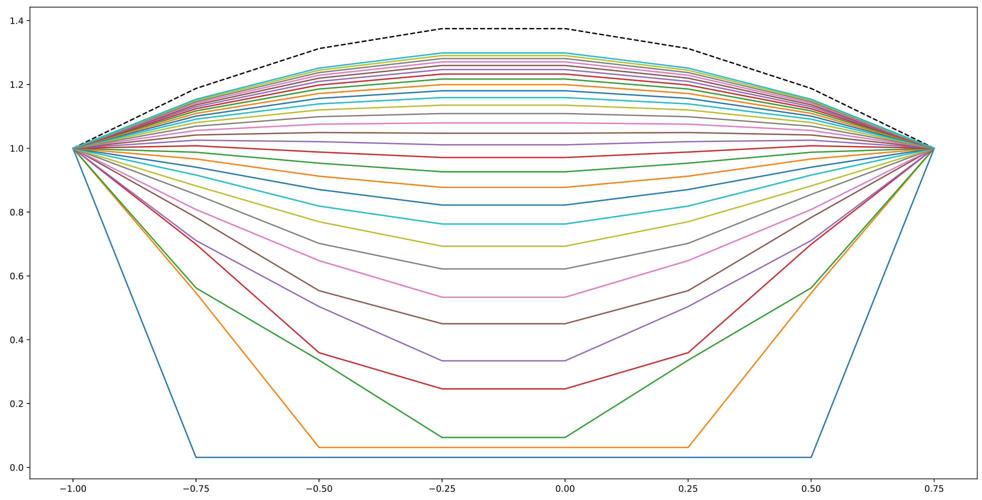

Figure 3: Jacobi iteration for 30 steps and $N = 100$. The black, dotted line indicates the solution obtained form he previous algorithm.

We see that in the first few steps, that the information is propagating from the boundaries to the middle. After some time a parabola arises. We now compute the same for $N = 100$:

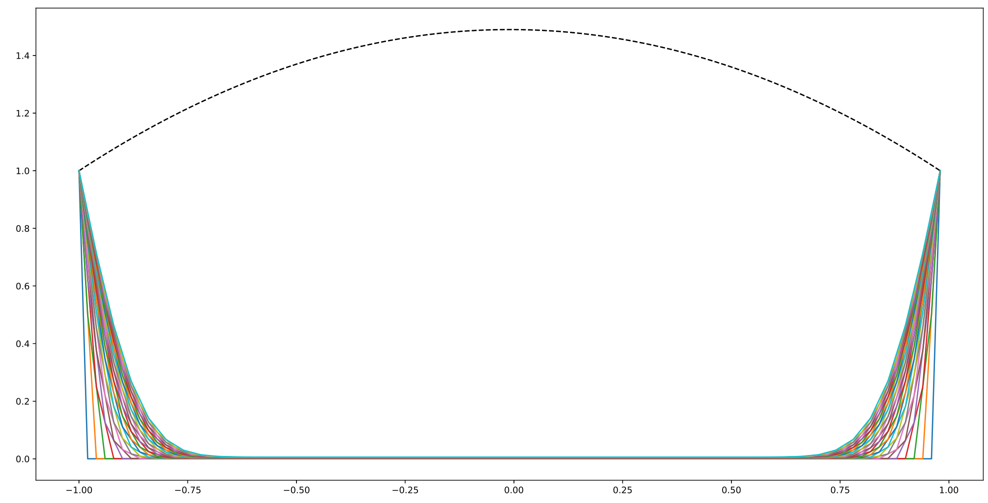

Figure 4: Jacobi iteration for 30 steps and $N = 100$. The black, dotted line indicates the solution obtained form he previous algorithm.

Here, the solution does not converge to the expected parabola. This is due to the fact that the information from the boundaries can only travel 30 grid points in 30 steps because we have nearest neighbor coupling only. On a grid with 100 gridpoints the stepsize is thus not sufficiently large.

Exercise 1.2

We begin with the construction of the restricted matrices. We employ a restriction scheme where the nearest neighbors are used to compute the coarser grid:

$$x_i^{2h} = \frac{1}{4}x_{i-1}^h + \frac{1}{2}x_i^h + \frac{1}{4}x_{i+1}^h \tag{3}$$

Writing this as a matrix for the 9×5, the 5×3 and the 3×2 case, we get:

$$R_1 = \frac{1}{4}\begin{pmatrix} 2 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 2 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 2 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 2 \end{pmatrix} \tag{4}$$

$$R_2 = \frac{1}{4}\begin{pmatrix} 2 & 1 & 0 & 0 & 0 \\ 0 & 1 & 2 & 1 & 0 \\ 0 & 0 & 0 & 1 & 2 \end{pmatrix} \tag{5}$$

$$R_3 = \frac{1}{4}\begin{pmatrix} 2 & 1 & 0 \\ 0 & 1 & 2 \end{pmatrix} \tag{6}$$

We can do the same thing for the prolongation matrices with the mapping pattern that calculates the intermediate points as averages of the next points to the left and the right and maps the static points directly onto their counterpart on the finer grid.

$$x_i^h = x_i 2h \text{ if static point}$$
$$x_i^h = \frac{1}{2}(x_{i+1}^{2h} + x_{i-1}^{2h}) \text{ if intermediate point}$$

$$P_0 = \frac{1}{2}\begin{pmatrix} 2 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 2 \end{pmatrix} \tag{7}$$

$$P_1 = \begin{pmatrix} 2 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 2 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 2 \end{pmatrix} \tag{8}$$

$$P_2 = \begin{pmatrix} 2 & 0 \\ 1 & 1 \\ 0 & 2 \end{pmatrix} \tag{9}$$

We immediately see that the matrices are the transposed of the restriction matrices multiplied by a factor of 2.