**University of Oslo** Fall, 2023
**Department of Mathematics** Mandatory assingment: 2

Author: Jonas Semprini Næss

# MAT3110/4110: Introduction to Numerical Analysis

## Problem 1.

a.) *Find a Householder transformation $H$ such that $B = HAH^T$ is a tridiagonal matrix. This involves some bookkeeping, so it may help to use the aid of a computer for calculating the linear algebra in steps that you describe.*

**Solution:**

We observe that $A \in \mathbb{R}^{3\times 3}$, meaning there is only a single Householder iteration needed to obtain our Householder matrix $H$. This is due to the Householder method only requiring $n-2$ reflection transformations to obtain the designated tridiagonal matrix, where $n$ naturally is the dimension of $A$.

Now let $\boldsymbol{v} = \boldsymbol{x} + \text{sign}(x_{21})\boldsymbol{e}_1$ where $\boldsymbol{x} = [0, 1/\sqrt{2}, -1/\sqrt{2}]^T$, which yields

$$\boldsymbol{v} = \left[0, \frac{1+\sqrt{2}}{\sqrt{2}}, -\frac{1}{\sqrt{2}}\right]^T.$$

Hence our Householder transformation looks accordingly

$$H = I_3 - 2\frac{\boldsymbol{v}\boldsymbol{v}^T}{\boldsymbol{v}^T\boldsymbol{v}}.$$

By calculation $\boldsymbol{v}^T\boldsymbol{v} = 0^2 + (\frac{1+\sqrt{2}}{\sqrt{2}})^2 + -(\frac{1}{\sqrt{2}})^2 = 2+\sqrt{2}$, meaning the expression now becomes

$$H = I_3 - \frac{2}{2+\sqrt{2}}\boldsymbol{v}\boldsymbol{v}^T$$

whereby the outer product $\boldsymbol{v}\boldsymbol{v}^T$ then gives

$$H = I_3 - \begin{pmatrix} 0 & 0 & 0 \\ 0 & \frac{\sqrt{2}+1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ 0 & -\frac{1}{\sqrt{2}} & \frac{\sqrt{2}-1}{\sqrt{2}} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{pmatrix}.$$

Which is then our Householder matrix.

To really verify that $H$ is the transformation we are looking for, we can further calculate

the product $B = HAH^T$.

$$
\begin{aligned}
B = HAH^T &= HAH \qquad \text{(H is symmetric)} \\
&= \begin{pmatrix} 1 & 0 & 0 \\ 0 & -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{pmatrix} \begin{pmatrix} 5 & \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{5}{2} & \frac{7}{2} \\ -\frac{1}{\sqrt{2}} & \frac{7}{2} & \frac{5}{2} \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{pmatrix} \\
&= \begin{pmatrix} 5 & -1 & 0 \\ -1 & -1 & 0 \\ 0 & 0 & 6 \end{pmatrix}
\end{aligned}
$$

where $B$ is a tridiagonal matrix. ■

b.) *Use Gershgorin's second theorem in combination with a similarity transformation to estimate the spectrum of A. Show that all eigenvalues are distinct and describe the location of the eigenvalues as accurately as you can.*

**Solution:**

By Gershgorin's Circle Theorem we know that when approximating the eigenvalues of $A$, the respective eigenvalues has to lay within at least one of the discs

$$D(a_{ii}, R_i) \subseteq \mathbb{C}.$$

Where $a_{ii}$ are the diagonal elements of $A$ and $R_i = \sum_{j \neq i} |a_{ij}|$.

By this result we can hence elaborate further on the eigenvalues of $A$ by observing the discs

$$D_1(5, \sqrt{2}) \tag{1}$$

$$D_2\left(\frac{5}{2}, \frac{7+\sqrt{2}}{2}\right). \tag{2}$$

Since $D_1 \subseteq D_2 \Rightarrow D_1 \cap D_2 \neq \emptyset$ it becomes beneficial to dissect these discs even farther seeing we wish to have as many disjont discs possible [1] To do so it is advantageous to perform a similarity transformation (looking into the same input space of $A$, but from the viewpoint of a different basis). Let

$$T = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \alpha & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

[1] Direct consequence of Greshgorin's second theorem stating that: "If the union of $k$ discs is disjoint from the union of the other $n-k$ discs then the former union contains exactly k and the latter $n-k$ eigenvalues of A, when the eigenvalues are counted with their algebraic multiplicities."

for $\alpha > 0$ and notice that since $B$ from (a.) is simply a reflection transformation of $A$ the spectrum of $A$ is conserved. This means that

$$\begin{aligned}\sigma(A) &= \sigma(HAH^T) \\ &= \sigma(B) \\ &= \sigma(TBT^{-1}).\end{aligned}$$

Thus, by taking the product $TBT^{-1}$ we have that

$$TBT^{-1} = \begin{pmatrix}1 & 0 & 0\\ 0 & \alpha & 0\\ 0 & 0 & 1\end{pmatrix}\begin{pmatrix}5 & -1 & 0\\ -1 & -1 & 0\\ 0 & 0 & 6\end{pmatrix}\begin{pmatrix}1 & 0 & 0\\ 0 & \frac{1}{\alpha} & 0\\ 0 & 0 & 1\end{pmatrix}$$

$$= \begin{pmatrix}5 & -\frac{1}{\alpha} & 0\\ -\alpha & -1 & 0\\ 0 & 0 & 6\end{pmatrix}$$

. Applying Greshgorin's Circle Theorem again yields the following discs

$$\tilde{D}_1\left(5, \frac{1}{\alpha}\right) \tag{3}$$

$$\tilde{D}_2\left(-1, \alpha\right). \tag{4}$$

Seeing that (from row 3) $|k-6| \leq 0,\ k \in \mathbb{R}$ we can conclude with $\lambda_1 = 6$ being one of the eigenvalues of $A$. Moreover, to hence determine the potential other eigenvalues we must demand that

$$\tilde{D}_1 \cap \tilde{D}_2 = \emptyset \Rightarrow 5 - \frac{1}{\alpha} = -1 + \alpha$$

$$\Downarrow$$

$$-\alpha^2 + 6\alpha - 1 = 0.$$

This systems has solutions $\alpha_1 \approx 5.828427,\ \alpha_2 \approx 0.1715728$, which yields $\lambda_2 \in \tilde{D}_1(\alpha \approx 5.828427) = B((5, 0.1715716))$ and $\lambda_3 \in \tilde{D}_2(\alpha \approx 0.1715728) = B((-1, 0.1715728))$, telling us that

$$\lambda_2 \in [5 - 0.1715716, 5 + 0.1715716]$$

$$\lambda_3 \in [-1 - 0.1715728, -1 + 0.1715728].$$

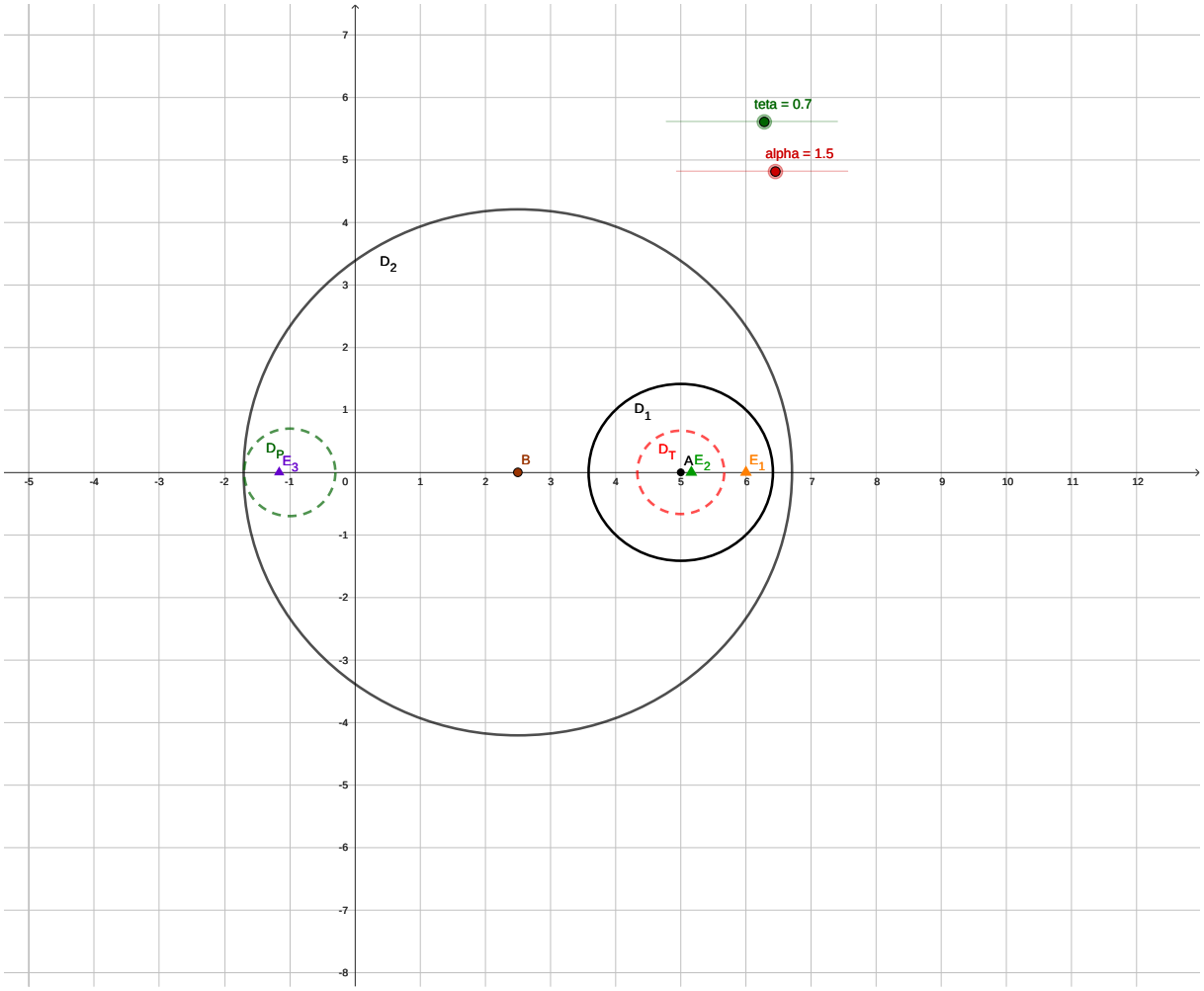


**Figure 1:** Visual of the Greshgorin circles. $D_1, D_2$ (Black) are the discs corresponding to $A$, $E_1, E_2, E_3$ are the actual eigenvalues of $A$, $D_T, D_P$ the discs corresponding to $TBT^{-1}$ and $\alpha_1 = 1.5,\ \alpha_2 = 0.7$

c.) *Computer exercise: Approximate the largest eigenvalue of A using power iteration and approximate the other two eigenvalues with inverse iteration (a shift is needed for the middle eigenvalue). Draw a random vector $x^{(0)}$ as start vector (for instance, with the components being independent, standard normal distributed). Include your implementation code and the output of 10 iterations for each approximation.*

**Solution:**

By running the implemented power iteration over 10 iterations through script 1 the following ouput is given

| Approx Eigenvalue | Actual Eigenvalue | Error |
|---|---|---|
| 6.0800585 | 6.0 | 0.0800585 |
| 5.6356824 | 6.0 | 0.3643176 |
| 5.7005412 | 6.0 | 0.2994588 |
| 5.7557640 | 6.0 | 0.2442360 |
| 5.8043507 | 6.0 | 0.1956493 |
| 5.8457834 | 6.0 | 0.1542166 |
| 5.8800852 | 6.0 | 0.1199148 |
| 5.9077903 | 6.0 | 0.0922097 |
| 5.9297235 | 6.0 | 0.0702765 |
| 5.9468134 | 6.0 | 0.0531866 |

**Table 1:** Output through 10 iterations of the power iteration.

For inverse iteration the following is given

| Approx Eigenvalue | Actual Eigenvalue | Error |
|---|---|---|
| 1.0000000 | -1.1622777 | 2.1622777 |
| 4.6485161 | -1.1622777 | 5.8107938 |
| -6.1464352 | -1.1622777 | 4.9841576 |
| -0.7073727 | -1.1622777 | 0.4549050 |
| -1.2235930 | -1.1622777 | 0.0613154 |
| -1.1575017 | -1.1622777 | 0.0047760 |
| -1.1616153 | -1.1622777 | 0.0006624 |
| -1.1627635 | -1.1622777 | 0.0004858 |
| -1.1621031 | -1.1622777 | 0.0001746 |
| -1.1623296 | -1.1622777 | 0.0000519 |
| -1.1622635 | -1.1622777 | 0.0000141 |

**Table 2:** Output through 10 iterations of the inverse iteration.

and lastly for the shifted inverse iteration with $\mu = 5$.

| Approx Eigenvalue | Actual Eigenvalue | Error |
|---|---|---|
| 6.0000000 | 5.1622777 | 8.3772234e-01 |
| 5.3504813 | 5.1622777 | 1.8820363e-01 |
| 5.2036234 | 5.1622777 | 4.1345701e-02 |
| 5.1691899 | 5.1622777 | 6.9122765e-03 |
| 5.1634094 | 5.1622777 | 1.1317319e-03 |
| 5.1624615 | 5.1622777 | 1.8380479e-04 |
| 5.1623075 | 5.1622777 | 2.9834424e-05 |
| 5.1622825 | 5.1622777 | 4.8415647e-06 |
| 5.1622784 | 5.1622777 | 7.8568265e-07 |
| 5.1622778 | 5.1622777 | 1.2749881e-07 |
| 5.1622777 | 5.1622777 | 2.0690213e-08 |

**Table 3:** Output through 10 iterations of the inverse iteration with shift ($\mu = 5$).

# Problem 2.

a.) *Describe the function $R(\beta)$ for the nonlinear least squares problem for determining this circle and compute its Jacobian $J_R(\beta) \in \mathbb{R}^{20\times 3}$.*

**Solution:**

The function $R(\beta)$ is simply what is defined as the cost function. By taking the difference of the model(estimation) against the true measurement, you will acheive what is called a residual, or error margin, which describes how much the model deviates at that certain point. Therefore, can $R(\beta)$ look as follows

$$R(\beta) = \begin{bmatrix} f(x_1, y_1; \beta) - b_1 \\ f(x_2, y_2; \beta) - b_2 \\ \vdots \\ f(x_n, y_n; \beta) - b_n \end{bmatrix}$$

which relative to this sample data gives

$$R(\beta) = \begin{bmatrix} f(x_1, y_1; \beta) - b_1 \\ f(x_2, y_2; \beta) - b_2 \\ \vdots \\ f(x_{20}, y_{20}; \beta) - b_{20} \end{bmatrix}.$$

For the case of the Jacobian we remember that a general Jacobian is defined as

$$J_f = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

where $f : \mathbb{R}^n \mapsto \mathbb{R}^m$ and $x \in \mathbb{R}^n$. Hence will the Jacobian for $f(x, y; \beta)$ look as follows

$$J_R(\beta) = \begin{bmatrix} \frac{\partial f_1}{\partial \beta_1} & \cdots & \frac{\partial f_1}{\partial \beta_3} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_{20}}{\partial x_1} & \cdots & \frac{\partial f_{20}}{\partial \beta_3} \end{bmatrix}$$

$$= \begin{bmatrix} -2(x_1 - \beta_1) & -2(y_1 - \beta_2) & -2\beta_3 \\ \vdots & \vdots & \vdots \\ -2(x_{20} - \beta_1) & -2(y_{20} - \beta_2) & -2\beta_3 \end{bmatrix}$$

b.) *Find a reasonable guess for $\beta^0$ and compute the circle $\beta^*$ that best describes the measurements in least squares sense using the Gauss-Newton method (3) on a computer. Include one plot of containing the circle $\beta^*$ you obtain and the measurement points.*

**Solution:**

Code can be found in the Appendix at 4, and produces the following plot.

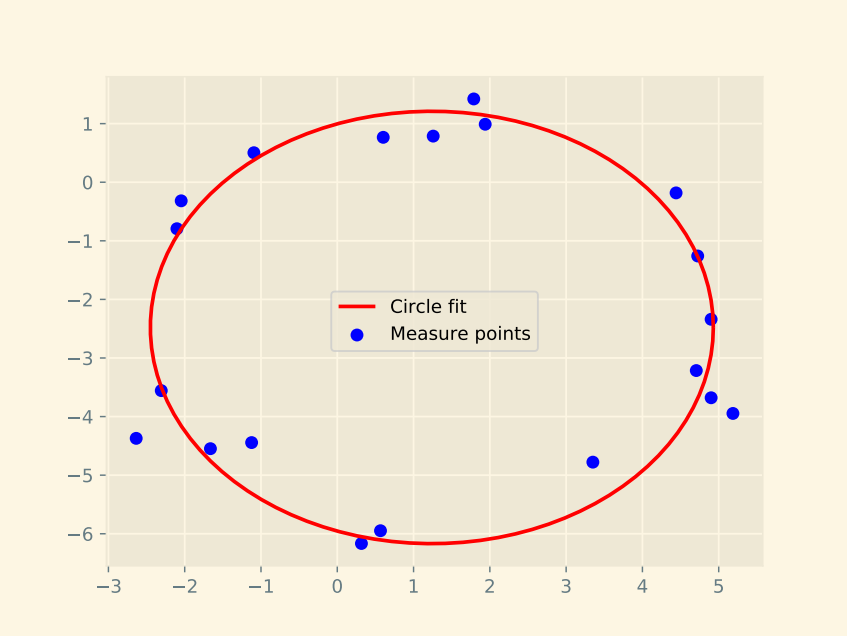


**Figure 2:** Plot of the circle $\beta^*$

with the correspoding $\beta$ values.

| $\beta$ | Estimate: |
|---|---|
| $\beta_1$ | 1.23942346 |
| $\beta_2$ | -2.47892999 |
| $\beta_3$ | 3.69002715 |

**Table 4:** Values of the approximated integral of $I$

c.) *Solve the linear least squares problem with unknowns $(x_c, y_c, w)$ on a computer and use the equation for $w$ to determine the circle's radius, $r$. How does the solution of this linear least squares problem compare to what you obtained in part b)?*

**Solution:**

By rewriting the equation we can implement the linear least squares as

$$2xx_c + 2yy_c + -x_i^2 - y_i^2 = 0$$

which by the following code (5) gives $r \approx 3.690027147759961$, and aligns well with the results of $\beta_3$ from Table (4).

# Problem 3.

a.) *Describe $L_k(x)$ and $\hat{L}_p(y)$ and show that $p(x, y)$ in (7) with the functions $L_k(x)$ and $\hat{L}_p(y)$ that you define indeed is a solution of (6).*

**Solution:**

We know by the ridgid definition of the Lagrange Polynomial in one dimension that

$$L_k(x) = \prod_{\substack{i=0 \\ i \neq k}}^{n} \frac{x - x_i}{x_k - x_i} = \frac{(x - x_0)(x - x_1)\dots(x - x_{i-1})\dots(x - x_n)}{(x_i - x_0)(x_i - x_1)\dots(x_i - x_{i-1})(x_i - x_{i+1})\dots(x_i - x_n)}$$

which can also be generalised for $y$, in the same manner, implying that

$$\hat{L}_p(y) = \prod_{\substack{j=0 \\ j \neq p}}^{n} \frac{y - y_j}{y_p - y_j} = \frac{(y - y_0)(y - y_1)\dots(y - y_{j-1})\dots(y - y_n)}{(y_j - y_0)(y_j - y_1)\dots(y_j - y_{j-1})(y_j - y_{j+1})\dots(x_j - y_n)}$$

Now observe for equation (6) that it becomes

$$\begin{aligned} p(x_i, y_j) = \sum_{k=0}^{n}\sum_{p=0}^{n} f(x_k, y_p) L_k(x_i)\hat{L}_p(y_j) &= \sum_{k=0}^{n}\sum_{p=0}^{n} f(x_k, y_p)\delta_{ik}\delta_{jp}, \quad i \neq k \wedge j \neq p. \\ &= f(x_i, y_j) \end{aligned}$$

which means that it interpolates the data, since each basis polynomial has degree $n$ and hence must the sum (or linear combinations) be of degree $\leq n$. Conclusively must $p(x, y)$ be a solution of $p(x_i.y_j) = f(x_i, y_j),\ \forall\ 0 \leq i, j \leq n$. ■

b.) *Leaning on the fact that any univariate polynomial $p$ of degree $\leq n$ that has $n+1$ or more zeros is equal to the 0 function, meaning $p \equiv 0$, show that the solution to the 2 dimensional interpolation problem (6) is unique.*

**Solution:**

Suppose that $p$ is not unique and there exists another solution to the interplolation problem $q \in \mathcal{P}_{n,n}$. Now define $r = p - q$, implying that $r \in \mathcal{P}_{n,n}$, meaning we can write it as

$$\sum_{i=0}^{n}\sum_{j=0}^{n} c_{i,j}x^i y^j, \ c_{i,j} \in \mathbb{R}.$$

Without loss of generality, we can fix any $0 \leq p \leq n$ such that the function $r(x, y_p)$ is now a univariate polyomial in $x$ of degree $\leq n$ and can be written as

$$\sum_{i=0}^{n} a_i(y_p)x^i$$

where $a_i(y_p) = \sum_{j=0}^{n} c_{i,j}y_p^j$. This polynomial has $n+1$ distinct root, namely $x_i, \ i = 0, 1, \ldots, n$. But a polynomial that has $n+1$ distinct roots and is of degree $\leq n$ can only happen to exist as long its $\equiv 0$ , which tells us that the only solution to $r(x, y_p) = 0$ is if all $a_i(y_p) = 0 \iff c_{i,j} = 0, \ \forall i, j$ meaning $r(x, y) \equiv 0$, which contradicts our assumption and $p \in \mathcal{P}_{n,n}$ must therefore be unique [2] ■

c.) *Determine $p_{i,j}(x, y)$ and verify that the 2D square-mesh trapezoidal rule is given by*

$$T(n) = \frac{h^2}{4}\sum_{i=1}^{n}\sum_{j=1}^{n}\Big(f(x_{i-1}, y_{j-1}) + f(x_i, y_{j-1}) + f(x_{i-1}, y_{j-1}) + f(x_i, y_j)\Big)$$

**Solution:**

We recall from (a.) that a general $p(x, y) = \sum_{k=0}^{n}\sum_{p=0}^{n} f(x, y)L_k(x)\hat{L}_p(y)$, which if we restrict to only be of degree $\leq 1$ yields

$$p_{i,j}(x, y) = \sum_{k=0}^{1}\sum_{p=0}^{1} f(x_{i-k}, y_{j-p})L_k(x)\hat{L}_p(y)$$

which written out in full is

$$\begin{aligned}
\sum_{k=0}^{1}\sum_{p=0}^{1} f(x_{i-k}, y_{j-p})L_k(x)\hat{L}_p(y) &= \sum_{k=0}^{1} L_k(x)\left[\hat{L}_0(y)(f(x_{i-k}, y_j) + \hat{L}_1(y)(f(x_{i-k}, y_{j-1})\right] \\
&= L_0(x)\left[\hat{L}_0(y)(f(x_i, y_j) + \hat{L}_1(y)(f(x_i, y_{j-1})\right] \\
&+ L_1(x)\left[\hat{L}_0(y)(f(x_{i-1}, y_j) + \hat{L}_1(y)(f(x_i.y_j)\right] \\
&= L_0(x)\hat{L}_0(y)f(x_i, y_j) + L_0(x)\hat{L}_1(y)f(x_i, y_{j-1}) \\
&+ L_1(x)\hat{L}_0(y)f(x_{i-1}, y_j) + L_1(x)\hat{L}_1(y)f(x_{i-1}, y_{j-1}).
\end{aligned}$$

[2] I originally had an idea of trying to show that the Vandermonde matrix for the linear map $R : \mathcal{P}_{n,n} \mapsto \mathcal{P}_{n,n}$ where $R(p) = p - q$, for $p, q \in \mathcal{P}_{n,n}$ is non-singular, hence invertible and thus must $\ker(R) = 0$ (I think..) meaning the only solution has to be 0, but encountered that to be rather difficult. I would more than glady recieve some direction on that if there is.

With $p_{i,j}(x,y)$ explicitly defined we can then turn to the integral

$$\int_{x_{i-1}}^{x_i}\int_{y_{j-1}}^{y_j} p_{i,j}(x,y) = \int_{x_{i-1}}^{x_i} L_0(x)\frac{y_j - y_{j-1}}{2}\Big[f(x_i,y_j) + f(x_i,y_{j-1})\Big] +$$

$$L_1(x)\frac{y_j - y_{j-1}}{2}\Big[f(x_{i-1},y_j) + f(x_{i-1},y_{j-1})\Big]$$

$$= \frac{x_i - x_{i-1}}{2}\frac{y_j - y_{j-1}}{2}\Big[f(x_i,y_j) + f(x_i,y_{j-1}) + f(x_{i-1},y_j) + f(x_{i-1},y_{j-1})\Big]$$

Where we remember that the problem is enclosed wihin a 2D square mesh with $(x_i, y_j) = (ih, jh)$, hence infering that

$$\frac{x_i - x_{i-1}}{2} = \frac{y_j - y_{j-1}}{2} = \frac{h}{2} \iff \frac{x_i - x_{i-1}}{2}\frac{y_j - y_{j-1}}{2} = \frac{h^2}{4}.$$

Which in application of the trapezoidal rule conclusively yields

$$T(n) = \sum_{i=1}^{n}\sum_{j=1}^{n}\frac{h^2}{4}\Big(f(x_{i-1},y_{j-1}) + f(x_i,y_{j-1}) + f(x_{i-1},y_{j-1}) + f(x_i,y_j)\Big)$$

$$= \frac{h^2}{4}\sum_{i=1}^{n}\sum_{j=1}^{n}\Big(f(x_{i-1},y_{j-1}) + f(x_i,y_{j-1}) + f(x_{i-1},y_{j-1}) + f(x_i,y_j)\Big)$$

and we are done. ■

d.) *For $n^s = 2^{1+s}$, $s = 1, 2, \ldots, 7$ compute $T(n_s)$ to estimate the integral*

$$I = \int_0^1\int_0^1 \exp(-(x - \sin(y^2))^3)\, dx\, dy$$

*Estimate numerically the order of convergence $r$ in*

$$E(n) = cn^{-r} + \mathcal{O}(n^{-(r+1)})$$

**Solution:**

One can compute $T(n_s)$ through the following script (6). Which gives that

| Iterations: $n_s$ | Approximation: |
|---|---|
| 4.0 | 0.9401276 |
| 8.0 | 0.9369460 |
| 16.0 | 0.9362333 |
| 32.0 | 0.9360607 |
| 64.0 | 0.9360179 |
| 128.0 | 0.9360072 |
| 256.0 | 0.9360045 |
| 512.0 | 0.9360038 |

**Table 5:** Values of the approximated integral of $I$

When so approximating the error margin $E(s) = \left|\tilde{I} - T(n_s)\right|$ we apply the pseudo-refrence solution $\tilde{I}$ through the following script (7), to analyse $r$.

| $r \approx -\frac{\log(E(n_s)) - log(E(n_{s-1}))}{\log(n_s) - \log(n_{s-1})}$ |
|---|
| 2.1297178200861935 |
| 2.0371047381023133 |
| 2.0105389924909276 |
| 2.0066241517597554 |
| 2.0176713454124955 |
| 2.0705388506214772 |
| 2.3219654771450973 |

**Table 6:** Values of the approximated integral of $I$

Which tells us that $r$ lays in the region of 2. It is though important to realise that when $T(n_s)$ gets sufficiently close to the pseudo-refrence solution, then $E(s)$ will experience round-off errors on a computer since it is dealing with very small numbers that are difficult to represent exactly (i.e 2.3219...).

Conclusively we then have that

$$E(n) \approx cn^{-2} + \mathcal{O}(n^{-3})$$

# Appendix

### Source code

**Listing 1:** Power Iteration

```
import numpy as np
import pandas as pd
import random

matrix = np.array(
    [
        [5, 1 / np.sqrt(2), -1 / np.sqrt(2)],
        [1 / np.sqrt(2), 5 / 2, 7 / 2],
        [-1 / np.sqrt(2), 7 / 2, 5 / 2],
    ]
)

result = pd.DataFrame(columns=["Approx Eigenvalue", "Actual Eigenvalue", "Error"])

num_iterations = 10
actual_eigenvalues, _ = np.linalg.eig(matrix)
print(actual_eigenvalues)
actual_largest_eigenvalue = max(actual_eigenvalues)

```

```
n = matrix.shape[0]
b = np.random.rand(n)

for i in range(num_iterations):
    # Power iteration
    b = np.dot(matrix, b)
    # Normalize the vector
    eigenvalue = np.linalg.norm(b)
    error = abs(eigenvalue - actual_largest_eigenvalue)
    b /= eigenvalue
    result.loc[i] = [eigenvalue, actual_largest_eigenvalue, error]


with pd.option_context(
    "display.max_rows",
    None,
    "display.max_columns",
    None
```

**Listing 2:** Inverse Iteration

```
import numpy as np
import pandas as pd
import random

matrix = np.array(
    [
        [5, 1 / np.sqrt(2), -1 / np.sqrt(2)],
        [1 / np.sqrt(2), 5 / 2, 7 / 2],
        [-1 / np.sqrt(2), 7 / 2, 5 / 2],
    ]
)

result = pd.DataFrame(columns=["Approx Eigenvalue", "Actual Eigenvalue", "Error"])

num_iterations = 10
actual_eigenvalues, _ = np.linalg.eig(matrix)


def inverse_power_method(matrix, iter, tol=1e-15):
    A = matrix
    b = np.zeros((len(A), iter + 1))
    b[:, 0] = np.random.normal(0, 0.1, A.shape[0])
    rn = np.ones((iter + 1,))
    for k in range(num_iterations):
        b[:, k] = b[:, k] / np.linalg.norm(b[:, k])
        b[:, k + 1] = np.linalg.solve(A, b[:, k])
        rn[k + 1] = np.sum(b[:, k + 1]) / np.sum(b[:, k])
        if abs(rn[k + 1] - rn[k]) < tol:
            break
```

```
    if k < iter:
        rn[k + 2 :] = rn[k + 1]
    inv_pow = 1.0 / rn
    for i in range(0, len(inv_pow)):
        eigenvalue = inv_pow[i]
        error = abs(eigenvalue - actual_eigenvalues[1])
        result.loc[i] = [eigenvalue, actual_eigenvalues[1], error]



inverse_power_method(matrix, iter=num_iterations)

with pd.option_context(
    "display.max_rows",
    None,
    "display.max_columns",
    None,
    "display.precision",
    7,
):
    print(result)
```

**Listing 3:** Inverse Iteration with shift

```
import numpy as np
import pandas as pd
import random

matrix = np.array(
    [
        [5, 1 / np.sqrt(2), -1 / np.sqrt(2)],
        [1 / np.sqrt(2), 5 / 2, 7 / 2],
        [-1 / np.sqrt(2), 7 / 2, 5 / 2],
    ]
)

result = pd.DataFrame(columns=["Approx Eigenvalue", "Actual Eigenvalue", "Error"])

num_iterations = 10
actual_eigenvalues, _ = np.linalg.eig(matrix)

def inverse_power_method(matrix, mu, iter, tol=1e-15):
    Ashift = matrix - mu * np.identity(matrix.shape[0])
    b = np.zeros((len(Ashift), iter + 1))
    b[:, 0] = np.random.normal(0.5, 0.3, Ashift.shape[0])
    rn = np.ones((iter + 1,))
    for k in range(num_iterations):
        b[:, k] = b[:, k] / np.linalg.norm(b[:, k])
        b[:, k + 1] = np.linalg.solve(Ashift, b[:, k])
        rn[k + 1] = np.sum(b[:, k + 1]) / np.sum(b[:, k])
```

```
        if abs(rn[k + 1] - rn[k]) < tol:
            break
    if k < iter:
        rn[k + 2 :] = rn[k + 1]
    inv_pow = 1.0 / rn + mu
    for i in range(0, len(inv_pow)):
        eigenvalue = inv_pow[i]
        error = abs(eigenvalue - actual_eigenvalues[0])
        result.loc[i] = [eigenvalue, actual_eigenvalues[0], error]


mu = 5
inverse_power_method(matrix, mu, iter=num_iterations)

with pd.option_context(
    "display.max_rows",
    None,
    "display.max_columns",
    None,
    "display.precision",
    7,
):
    print(result)
```

**Listing 4:** Least Squares fit Circle Fit

```
import scipy.io
import numpy as np
import matplotlib.pyplot as plt

measurementData = scipy.io.loadmat("circle-measurements.mat")
x = measurementData["x"].reshape(-1)
y = measurementData["y"].reshape(-1)

phi = np.linspace(0, 1, 100)

b_1 = (max(x) + min(x)) / 2
b_2 = (max(y) + min(y)) / 2
b_3 = (abs(max(x)) + abs(min(x)) + abs((max(y)) + abs(min(x)))) / 4
beta = [b_1, b_2, b_3]

n = 3
matrix = np.zeros((len(x), n))


def circle_model(beta, x, y):
    return (x - beta[0]) ** 2 + (y - beta[1]) ** 2 - beta[2] ** 2


def Jacobian(matrix, beta):
```

```
    jac = matrix
    for i in range(0, len(x)):
        jac[i][0] = -2 * (x[i] - beta[0])
        jac[i][1] = -2 * (y[i] - beta[1])
        jac[i][2] = -2 * (beta[2])
    return jac


def gauss_newton(beta, max_iter=100, tolerance=1e-6):
    params = beta
    for _ in range(max_iter):
        J = Jacobian(matrix, params)
        r = circle_model(params, x, y)
        J_T = np.transpose(J)
        S = np.linalg.pinv(np.matmul(J_T, J))
        delta_params = np.matmul(S, np.matmul(J_T, r))

        # print(delta_params.shape, r.shape)
        params = params - delta_params

        if np.linalg.norm(delta_params) < tolerance:
            break

    return params


beta_n = gauss_newton(beta)
print(beta_n)

s1 = beta_n[0] + beta_n[2] * np.cos(2 * np.pi * phi)
s2 = beta_n[1] + beta_n[2] * np.sin(2 * np.pi * phi)
plt.scatter(x, y, marker="o", color="green", label="Measure points")
plt.plot(s1, s2, color="red")
plt.xlim(min(x), max(x))
plt.ylim(min(y) - 1, max(y) + 2)
plt.legend()
 plt.show()
```

**Listing 5:** Linear Least Squares fit Circle Fit

```
import scipy.io
import numpy as np
import matplotlib.pyplot as plt

measurementData = scipy.io.loadmat("circle-measurements.mat")
x = measurementData["x"].reshape(-1)
y = measurementData["y"].reshape(-1)

A = np.c_[2 * x, 2 * y, np.ones_like(x)]

```

```
r = np.linalg.solve(A.T @ A, A.T @ (-(x**2) - (y**2)))

r_fit = np.sqrt(r[0] ** 2 + r[1] ** 2 - r[2])

print(r_fit)
```

**Listing 6:** Numerical Integral

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

results = pd.DataFrame(columns=["Iterations (n)", "Approx Integral"])

P_s = 2 ** (10)

n = [int(2 ** (1 + i)) for i in np.linspace(1, 8, 8)]


# Define the function to be integrated
def f(x, y):
    return np.exp(-((x - np.sin(y**2)) ** 3))


# Define the integration limits
x_min, x_max = 0, 1
y_min, y_max = 0, 1


integral_results = []


# Composite Trapezoidal Rule function
def composite_trapezoidal_rule(f, n):
    x_values = np.linspace(0, 1, n + 1)
    y_values = np.linspace(0, 1, n + 1)
    h = 1 / n
    X, Y = np.meshgrid(x_values, y_values)
    z = f(X, Y)
    sum = np.sum(z[1:, 1:] + z[:-1, 1:] + z[1:, :-1] + z[:-1, :-1])
    comp_sum = ((h**2) / 4) * sum
    return comp_sum


# Calculate the integral for different values of n
for s in n:
    result = composite_trapezoidal_rule(f, s)
    integral_results.append(result)


```

```
for i, s in enumerate(n):
    result = integral_results[i]
    results.loc[i] = [s, result]

with pd.option_context(
    "display.max_rows",
    None,
    "display.max_columns",
    None,
    "display.precision",
    7,
):
    print(results)
```

**Listing 7:** Error Estimation

```
pseudo = composite_trapezoidal_rule(f, P_s)

error = [abs(pseudo - app) for app in integral_results]

print(len(error), len(n))
for i in range(1, len(n)):
    num = np.log(error[i]) - np.log(error[i - 1])
    den = np.log(n[i]) - np.log(n[i - 1])
    r = -(num / den)
    print(r)
```