12 Interpolation and Approximation

A common problem is estimation of a function value for a value x, given a series of discrete data points with corresponding function $f(x)$ values. The data may come from experimental observations or numerical computations. There are two approaches to solving this problem: interpolation and approximation (curve fitting).

12.1 Interpolation

Interpolation is a process of finding a function $f(x)$ fulfilling the equality conditions in interpolation nodes or finding a value of a new point x between the interpolation nodes. Given a set of discrete data points, interpolation would mean finding a function going exactly through those data points. In doing so, we assume, that the data points are accurate and to not contain any noise.

Interpolation methods include:

- approximation with a piecewise linear function,
- polynomial interpolation:
  - power-base polynomial,
  - Lagrange polynomial,
  - Chebyshev polynomial,
  - Neville algorithm,
  - Newton interpolation formula,
- rational function interpolation,
- spline function interpolation,
- trigonometric function interpolation.

12.1.1 Polynomial interpolation

The simplest interpolant $F(x)$ is a polynomial. It is always possible to construct a unique polynomial of degree n that passes through $n+1$ distinct points. Given the data points $(x_i, f(x_i))$, $i = 0, 1, 2, \ldots, n$ we assume an interpolant in a form of a polynomial $F(x) = P(x)$:

$$P(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n, \tag{12.1}$$

where a_i are coefficients of the polynomial. If we substitute known pairs of values $(x_i,\ f(x_i))$ to equation 12.1, we obtain $n+1$ equations which allow to solve for coefficients a_i from a system of linear equations:

$$\mathbf{A}\mathbf{x} = \mathbf{f}(\mathbf{x}), \tag{12.2}$$

where

$$\mathbf{A} = \begin{bmatrix} 1 & x_0 & x_0^2 & \ldots & x_0^n \\ 1 & x_1 & x_1^2 & \ldots & x_1^n \\ \vdots & & & & \vdots \\ 1 & x_n & x_n^2 & \ldots & x_n^n \end{bmatrix}, \qquad \mathbf{x} = \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix}, \qquad \mathbf{f}(\mathbf{x}) \begin{bmatrix} f(x_0) \\ f(x_1) \\ f(x_2) \\ \vdots \\ f(x_n) \end{bmatrix}. \tag{12.3}$$

Since the number of equations is the same as the number of points (assuming that the data points do not contradict each other), methods for solving linear equations can be used to determine the values of polynomial coefficients. Matrix $\mathbf{A}$ is called the **Vandermonde matrix**. Functions $1, x^1, x^2, \ldots, x^n$ are called base functions, in this case – a power base. Depending on what functions are used as base functions, the polynomials may have different names.

12.1.2 Lagrange's polynomial

One of the methods of obtaining the polynomial interpolant is the Lagrange formula:

$$P(x) = f(x_0)\ell_0(x) + f(x_0)\ell_0(x) + \ldots + f(x_0)\ell_0(x) = \sum_k f(x_k)\ell_k(x), \tag{12.4}$$

where the subscription n is a degree of a polynomial and:

$$\ell_k(x) = \frac{(x-x_0)\ldots(x-x_{k-1})(x-x_{k+1})\ldots(x-x_n)}{(x_k-x_0)\ldots(x_k-x_{k-1})(x_k-x_{k+1})\ldots(x_k-x_n)} = \prod_{k\neq i}\frac{x-x_i}{x_k-x_i}. \tag{12.5}$$

are called the **cardinal functions**.

Example

Given the data points $(x,\ f(x))$ in a tabular form:

x_i	0	1	2	3
$f(x_i)$	-5	-6	-1	-16

Lagrange's polynomial of the interpolant looks as follows:

$$\begin{aligned} P(x) =& \frac{(x-1)(x-2)(x-3)}{(0-1)(0-2)(0-3)}(-5) + \frac{(x-0)(x-2)(x-3)}{(1-0)(1-2)(1-3)}(-6) + \\ &+ \frac{(x)(x-1)(x-3)}{(2-0)(2-1)(2-3)}(-1) + \frac{(x-0)(x-1)(x-2)}{(3-0)(3-1)(3-2)}(16) \end{aligned}$$

12.1.3 Newton's method (Divided Differences)

Lagrange's method is not a very efficient algorithm. Better computational results can be obtained with Newton's method, where the polynomial can be written in an alternative form. We denote by $f[x_1 \ldots x_n]$ the coefficient of the x^{n-1} term in the polynomial that interpolates the data points $((x_1, f(x_1), \ldots (x_n, f(x_n)))$. Then, the polynomial can be written in a form:

$$P_n(x) = f[x_1] + f[x_1\ x_2](x-x_1) + \ldots + f[x_1\ x_2\ \ldots x_n](x-x_1)\ldots(x\_x_{n-1}), \tag{12.6}$$

which has a more efficient evaluation procedure. The coefficients of Newton's polynomial P_n are computed by forcing the polynomial to pass through each data point $f(x_i) = P_n(x_i)$.

Using the **divided differences**:

$$f[x_k] = y_k$$

$$f[x_k\ x_{k+1}] = \frac{f[x_{k+1}] - f[x_k]}{x_{k+1} - x_k} \tag{12.7}$$

$$f[x_k\ x_{k+1}\ x_{k+2}] = \frac{f[x_{k+1}\ x_{k+2}] - f[x_k\ x_{k+1}]}{x_{k+2} - x_k} \tag{12.8}$$

$$\vdots$$

$$f[x_k\ x_{k+1}\ x_{k+2}\ x_{k+3}] = \frac{f[x_{k+1}\ x_{k+2}\ x_{k+3}] - f[x_k\ x_{k+1}\ x_{k+2}]}{x_{k+3} - x_k}. \tag{12.9}$$

The interpolating polynomial is:

$$P(x) = \sum_{i=1}^{n} f[x_1\ \ldots x_i](x-x_1)\ldots(x-x_{i-1})$$

Table 12.1. Newton's divided differences in a table form

$$\begin{array}{c|cccc} x_1 & f[x_1] & & & \\ & & f[x_1\ x_2] & & \\ x_2 & f[x_2] & & f[x_1\ x_2\ x_3] & \\ & & f[x_2\ x_3] & & \\ x_3 & f[x_3] & & & \end{array}$$

The recursive definition allows to evaluate the terms of a Newton's divided differences in a table form. For three data points, the table may look as the one in Table 12.1.

Example

Given the data points $(0,1),(2,2),(3,4)$, solve the interpolation task using the Newton's polynomial (example from (Sauer 2012)). Using equations 12.9, we get:

$$\begin{aligned} f[x_1] &= 1 \qquad f[x_2] = 2 \qquad f[x_3] = 4 \\ f[x_1 x_2] &= \frac{f[x_2] - f[x_1]}{x_2 - x_1} = \frac{2-1}{2-0} = \frac{1}{2} \\ f[x_2 x_3] &= \frac{f[x_3] - f[x_2]}{x_3 - x_2} = \frac{4-2}{3-2} = 2 \\ f[x_1 x_2 x_3] &= \frac{f[x_2\ x_3] - f[x_1\ x_2]}{x_3 - x_1} = \frac{2 - \frac{1}{2}}{3-0} = \frac{\frac{3}{2}}{3} = \frac{1}{2} \end{aligned}$$

In a table form:

$$\begin{array}{c|cccc} 0 & 1 & & & \\ & & \frac{1}{2} & & \\ 2 & 2 & & \frac{1}{2} & \\ & & 2 & & \\ 3 & 4 & & & \end{array}$$

The coefficients can be written from the top edge of the table (1, 1/2, 1/2). The interpolating polynomial can be written as:

$$P_n(x) = 1 + \frac{1}{2}(x-0) + \frac{1}{2}(x-0)(x-2).$$

Adding a fourth data point $(1,0)$, would require adding an additional row and column:

$$\begin{array}{c|cccc} 0 & 1 & & & \\ & & \frac{1}{2} & & \\ 2 & 2 & & \frac{1}{2} & \\ & & 2 & & -\frac{1}{2} \\ 3 & 4 & & 0 & \\ & & 2 & & \\ 1 & 0 & & & \end{array}$$

And a polynomial would look like the one below:

$$P_n(x) = 1 + \frac{1}{2}(x-0) + \frac{1}{2}(x-0)(x-2) - \frac{1}{2}(x-0)(x-2)(x-3).$$

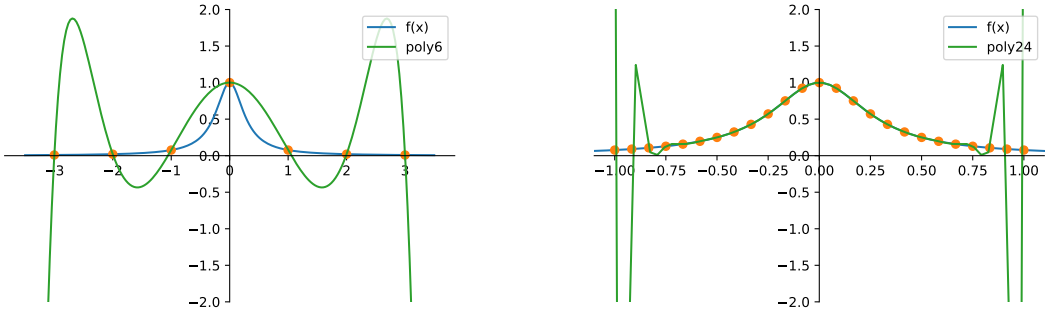

Figure 12.1. Runge's phenomenon of an interpolant for function $f(x) = \frac{1}{1+12x^2}$

Newton's method of interpolation involves two steps: computation of the coefficients and the evaluation of the polynomial. It works well if the interpolation is repeated for various values of x using the same polynomial. However, if only one point is interpolated, other method may be better suited.

12.1.4 Downsides of polynomial interpolation

Polynomial interpolation has its advantages and disadvantages. Function values are easy to determine once the interpolant is known and the function is easily differentiable and integrable. It is also useful to approximate complex functions in a less complicated way. The polynomial passes through all data points. However, linear interpolation should be carried out with fewest feasible number of points: linear interpolation using the nearest two data points is often sufficient. An interpolant based on more than six data points should be avoided – data points are far from the point of interest and do not contribute to the accuracy of an interpolated function. Polynomials of a high degree can oscillate between the data points (Figure 12.1). That is oscillation is known as the **Runge's phenomenon**.

Polynomial extrapolation – interpolation outside the range of data points is unpredictable. The interpolant may depart from the obvious trend and it is simply unknown how the function behaves outside of data points interval. Using the same interpolant leads to lower accuracy in extrapolation than interpolation. If extrapolated cannot be avoided, a low-order polynomial based on nearest-neighbour data points should be used. The results ought to be verified visually by plotting the function and extrapolation results.

12.1.5 Other types of interpolation

[1] Polynomial interpolation has many drawbacks and limits, so, other methods are also used for this numerical task:

- piecewise linear approximation,
- polynomial interpolation,
  - power base polynomial
  - Lagrange's polynomial,
  - Newton method
  - Chebyshev polynomial,
  - Neville's method,

[1] very brief intro

- rational function interpolation,
- spline function (e.g. cubic spline),
- trigonometric function interpolation;

A rational function interpolation is interpolation with a function given as:

$$R(x) = \frac{P_m(x)}{Q_n(x)}. \tag{12.10}$$

$R(x)$ is a ratio of two polynomials $P_m(x)$ and $Q_n(x)$ with degrees m and n.

Spline function interpolation:

- piecewise curve (usually cubic – **cubic spline**)
- **knot** – an interior data point that connects two subintervals:
  - coefficients for n data points and degree k: $(k+1)(n-1)$,
  - equations in knots: $2(n-1)$,
  - additional equation needed (derivatives, slopes),
- less tendency to oscillate between data points,
- similar approach – Bezier curves.

12.2 Approximation

Approximation of a function $f(x)$ is a process of finding coefficients a_0, a_1, a_m of an approximating function $F(x)$ which is a linear combination of known functions $\varphi_i(x)$:

$$F(x) = a_0\varphi_0(x) + \ldots + a_n\varphi_n(x), \tag{12.11}$$

in a way, that it fulfils required conditions, for example, minimizing the difference $\|f(x) - F(x)\|$. Curve fitting is applied to data that contain noise (scatter), usually caused by measurement errors, thus, the function does not necessarily go through the data points. The task is to find a smooth curve that fits all the data points with approximately the same error. The curve should have a possibly simple form, to avoid reproducing the noise.

12.2.1 Least squares method

The form of $f(x)$ is usually determined from the theory associated with the experiment from which the data are obtained. Curve fitting involves two steps: choosing the appropriate form of $f(x)$ and computing its parameters to produce the best fit to the data. What is the **best fit**? Provided that the noise only appears in a y coordinate, the most common measure is the **least-squares fit**, which minimizes the function:

$$S(a_0, a_1, \ldots, a_m) = \sum_{i=1}^{n} \left(y_i - f(x_i)\right)^2, \tag{12.12}$$

with respect to each a_i. The terms $y_i - f(x_i)$ are called **residuals** and represent the discrepancy between the data points y_i and the fitting function at x_i. The spread of data around the fitting curve is described with the standard deviation, defined as:

$$\sigma = \sqrt{\frac{S}{n-m}}. \tag{12.13}$$

If $n = m$, we have interpolation, both numerator and denominator are zero and σ is indeterminate.

A function often used with least-squares method is a polynomial. With a degree of a polynomial m, the fitted function has a form of $f(x) = \sum_{j=0}^{m} a_j \varphi_j(x) = \sum_{j=0}^{m} a_j x^j$, the basis functions are:

$$\varphi_j(x) = x^j \qquad j = 0, 1, \ldots, m. \tag{12.14}$$

Assuming the matrix form of the equations for data points:

$$\mathbf{A} = \begin{bmatrix} \varphi_0(x_1) & \ldots & \ldots & \varphi_m(x_1) \\ \varphi_0(x_2) & \ldots & \ldots & \varphi_m(x_2) \\ \vdots & & & \vdots \\ \varphi_0(x_n) & \ldots & \ldots & \varphi_m(x_n) \end{bmatrix}, \qquad \mathbf{x} = \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_m \end{bmatrix}, \qquad \mathbf{b} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}, \tag{12.15}$$

where $\mathbf{A}$ is the coefficient matrix, $\mathbf{x}$ is the vector of unknowns and $\mathbf{b}$ is the vector of fitted values.

The best fit in least-squares is found by minimizing the Euclidean distance between the data points and the fitted curve: the shortest distance from a point to a plane is along the line segment perpendicular (orthogonal) to the plane:

$$(b - \mathbf{A}\bar{x}) \perp \mathbf{A}x | x \in \mathbb{R}^n. \tag{12.16}$$

If the vectors u and v are perpendicular, $u^T \cdot v = 0$, using the matrix notation:

$$(\mathbf{A}x)^T (b - \mathbf{A}\bar{x}) = 0. \tag{12.17}$$

Knowing that $(A + B)^T = A^T + B^T$ and $(AB)^T = A^T B^T$:

$$x^T \mathbf{A}^T (b - \mathbf{A}\bar{x}) = 0. \tag{12.18}$$

The above condition means that $\mathbf{A}^T(b - \mathbf{A}\bar{x})$ is perpendicular to every x, including itself; it is possible only if:

$$\mathbf{A}^T (b - \mathbf{A}\bar{x}) = 0, \tag{12.19}$$

this gives a system of equations that defines the least squares solution (**system of normal equations**):

$$\mathbf{A}^T \mathbf{A}\mathbf{x} = \mathbf{A}^T \mathbf{b}. \tag{12.20}$$

The coefficient matrix in the **system of linear equations** is positive-definite and symmetrical – it provides that the solution is unique and can be found using methods for solving systems of linear equations:

$$\mathbf{x} = (\mathbf{A}^T \mathbf{A})^{-1} \mathbf{A}^T \mathbf{b}.$$

The normal equations become ill-conditioned with increasing polynomial degree m – in curve-fitting only low-order polynomials are useful, high-order polynomials are not recommended since they reproduce the noise in the data.

12.2.2 Periodic functions

To approximate periodic phenomenons with period 2π, use sine and cosine functions. The approximated model is still linear.

The simplest model:

$$f(x) = c_1 + c_2 \cos(2\pi t) + c_3 \sin(2\pi t),$$

where t is part of the phenomenon's period $T = 2\pi$.

More complex model is possible: 1, $\sin x$, $\cos x$, $\sin 2x$, $\cos 2x$, $\ldots$, $\sin mx$, $\cos mx$:

$$F(x) = a_0 + \sum_{k=1}^{m} a_k \sin kx + b_k \cos kx$$

Example

The table below includes temperature values measured in Washington. Fit the data using a periodic model.

time	t	temperature [C]
00:00	0	-2.2
3:00	1/8	-2.8
6:00	1/4	-6.1
9:00	3/8	-3.9
12:00	1/2	0.0
15:00	5/8	1.1
18:00	3/4	-0.6
21:00	7/8	-1.1

We assume the model has a form: $F(x) = c_1 + c_2\cos(2\pi t) + c_3\sin(2\pi t)$, $T = 24$ h. Using the least-square fit, it is possible to determine the values of c_1, c_2 and c_3.

12.3 Interpolation and approximation in Python

Some of the interpolation and approximation methods are available in Python – in `NumPy` and `SciPy` modules, especially `polynomial`[2] and `interpolate` (example 12.1).

Example 12.1. Importing modules

```
1 from numpy import polynomial as p
2 from scipy import interpolate as i
```

To define an interpolating polynomial, one of the polynomial classes has to be used (example `[1]`) and either coefficients or roots (`fromroots`) need to be provided.

In [1]:
```
1 from numpy import polynomial as p
2
3 p1 = p.Polynomial([1, 2, 3]) # coefficients
4 p2 = p.Polynomial.fromroots([-1, 1]) # roots
5 print(p2.roots())     # check roots of p2
6 #Legendre and Chebyshev methods are also available
7 c1 = p.Chebyshev([1, 2, 3])
8 print(c1.coef, c1.roots())
```

```
[-1.  1.]
[1. 2. 3.] [-0.76759188 0.43425855]
```

Solving a polynomial interpolation problem requires filling a matrix with values of basis functions for available data points. This matrix is called a Vandermonde matrix. A method `polyvander` is available to do it automatically. Then, `linalg.solve` function from `NumPy` can be used to solve for coefficients (example `[2]`).

[2] https://numpy.org/doc/stable/reference/routines.polynomials.polynomial.html#module-numpy.polynomial.polynomial
https://numpy.org/doc/stable/reference/routines.polynomials.chebyshev.html#module-numpy.polynomial.chebyshev

In [2]:
```
import numpy as np

# data points (x, y)
x = [0, 1, 2, 3]
y = [0, 1, 8, 27]

A = p.polynomial.polyvander(x, 3)   # Vandermonde matrix
c = np.linalg.solve(A, y)    # coefficients
print(f"{A=}\n {c=}")
```

```
A=array([[ 1., 0., 0., 0.],
[ 1., 1., 1., 1.],
[ 1., 2., 4., 8.],
[ 1., 3., 9., 27.]])
c=array([ 0., 0., -0., 1.])
```

SciPy's `interpolate` module contains couple of functions that can be also used for interpolation. One of those is `interp1d`[3] that creates an interpolant function based on a set of datapoints and allows to get a value in between the data points (example [3]). Another example is `splrep` used for creating a B-spline representation of a 1-D curve[4].

In [3]:
```
from scipy import interpolate as i
import numpy as np

x = np.linspace(-3, 3, 30)
y = np.sin(x)
x_0 = 1.2

f = i.interp1d(x, y, kind='cubic')
print(f(x_0)) # interpolate at point x_0
```

```
0.9320358111414948
```

Curve-fitting usually requires using a least-squares approach. A function `lstsq` is available in `NumPy's linalg` module (example [4]). Vandermonde matrix can, again, be automated with `np.vander` or row-by-row using `np.vstack`.

In [4]:
```
import numpy as np

# data for curve fitting
x = np.linspace(-5, 5, 50)
y = x**4 + x/2 + 1 + 3*np.random.random() # adding random noise

A = np.vstack([np.ones(len(x)), x.T, x.T**2]) #vandermonde matrix with base
    polynomial a + b x + c x**2
#A.T
sol = np.linalg.lstsq(A.T, y) # returns x, sum of squared residuals, rank of A,
    singular values of A
sol
```

[3] https://docs.scipy.org/doc/scipy/reference/generated/scipy.interpolate.interp1d.html
[4] https://docs.scipy.org/doc/scipy/reference/generated/scipy.interpolate.splrep.html#scipy.interpolate.splrep

Out[4]:
```
(array([-55.59512701, 0.5      ,  22.27345749]),
array([131673.66710665]),
3,
array([82.43110885, 20.8248282 , 4.70279306]))
```

Another solution would be using `polyfit`[5] from `numpy.polynomial` module. This method returns coefficients of a polynomial of a chosen degree that are a least-squares fit to the provided data (example [5]). It is also possible to do 2D fits.

In [5]:
```
import numpy as np
from numpy.polynomial import polynomial as P

# data for curve fitting
x = np.linspace(-5, 5, 50)
y = x**4 + x/2 + 1 + 3*np.random.random() # adding random noise
c, stats = P.polyfit(x, y, 3, full=True) # polynomial of degree 3, return
    coefficients and statistics
```

```
c=array([-5.66767889e+01, 5.00000000e-01, 2.22734575e+01, -2.88308015e-16])
stats=[array([131673.66710665]), 4, array([1.38447092, 1.3211945 , 0.5044255 ,
    0.28851388]), 1.1102230246251565e-14]
```

12.4 Exercises

Exercise 12.1. Use the data in table 12.2 to interpolate function $f(x)$:

- using a power-base polynomial,
- using a Lagrange interpolation polynomial

[6].

Table 12.2. Data for Exercise 12.1

x	2.1	4.1	7.1
$f(x)$	-12.4	7.3	10.1

1. Verify if $P(x) = f(x)$ for every value of x.
2. Using the obtained polynomial, compute $f(x)$ for $x = 3$.
3. Using the obtained polynomial, compute $f(x)$ for $x = 8$.
4. Plot function $f(x)$ on the interval $x \in \langle 0,\ 10\rangle$, mark given data points and interpolated and extrapolated points (from points 2 and 3).

[5] https://numpy.org/doc/stable/reference/generated/numpy.polynomial.polynomial.polyfit.html#numpy.polynomial.polynomial.polyfit

[6] Exercise adapted from Gerald and Wheatley (2004).

Exercise 12.2. Using the data points in Table 12.3, interpolate the function $f(x)$ using Newton's method at $x = 0, 0.5, 1.0, 7.0, 8.0$. Compare computed values with exact values $f(x) = 4.8 \cos \frac{\pi x}{20}$.[7]

Table 12.3. Data for Exercise 12.2

x	0.15	2.30	3.15	4.85	6.25	7.95
$f(x)$	4.79867	4.49013	4.22430	3.47313	2.66674	1.51909

Exercise 12.3. Approximate a linear and a parabolic function to the data in Table 12.4. Which approximation is better? Why?[8]

Table 12.4. Data for Exercise 12.3

x	1.0	2.5	3.5	4.0	1.1	1.8	2.2	3.7
y	6.008	15.722	27.130	33.772	5.257	9.549	11.098	28.828

Exercise 12.4. Create a complicated function and compute values for 100 points of this function. Next, add some random noise to the data. Approximate a polynomial curve with degree n, $n \in \{0, 1, 2, 3, 4, 10\}$. Compare variance values. Which curve is the best fit for observation data?

Exercise 12.5. Given the data in Table 12.5, approximate a periodic curve:[9]

1. $F(t) = c_1 + c_2 \cos(2\pi t) + c_3 \sin(2\pi t)$
2. $F(t) = c_1 + c_2 \cos(2\pi t) + c_3 \sin(2\pi t) + c_4 \cos(4\pi t)$

Table 12.5. Data for Exercise 12.5

t	0	1/6	1/3	1/2	2/3	5/6
y	0	2	0	-1	1	1

References

Burden, R. L. and Faires, J. D. (2011). *Numerical Analysis*. Cengage, 9th edition.

Gerald, C. F. and Wheatley, P. O. (2004). *Applied Numerical Analysis*. Pearson Addison Wesley.

Kiusalaas, J. (2013). *Numerical methods in Python*.

Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P. (2007). *Numerical recipes. The art of scientific computing*. Cambridge University Press, Cambridge, 3rd edition.

Sauer, T. (2012). *Numerical Analysis*. Pearson, USA, 2nd edition.

Stoer, J. and Bulirsch, R. (2013). *Introduction to Numerical Analysis*, volume 12. Springer Science & Business Media, 2nd edition.

[7] Exercise adapted from Kiusalaas (2013).
[8] Exercise adapted from Kiusalaas (2013)
[9] Exercise adapted from Kiusalaas (2013)