Exercise 1. Sometimes we need to find a polynomial curve that passes through given points. More specifically, we begin with a table of values:

x	x_0	x_1	•••	x_n
y	y_0	y_1	•••	y_n

and assume that the x-values form a set of $n+1$ distinct points. A polynomial $p(x)$ for which $p(x_i)=y_i$ when $0\le i\le n$ is said to interpolate the table. The points x_i are called nodes. A corresponding theorem states that if x_i's are distinct real numbers, then for arbitrary y_i's there exists a unique polynomial of degree at most n. A straightforward way to find this polynomial is to set a system of equations, $p(x_i)=y_i$, treating its coefficients as unknowns, $p(x)=a_0+a_1x+a_2x^2+\cdots+a_nx^n$:

$$\begin{cases} p(x_0)=a_0+a_1x_0+a_2x_0^2+\cdots+a_nx_0^n=y_0 \\ p(x_1)=a_0+a_1x_1+a_2x_1^2+\cdots+a_nx_1^n=y_1 \\ \quad\cdots \\ p(x_n)=a_0+a_1x_n+a_2x_n^2+\cdots+a_nx_n^n=y_n \end{cases}$$

In vector-matrix notation it is written as

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ 1 & x_2 & x_2^2 & \cdots & x_2^n \\ \vdots & \vdots & \cdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{bmatrix}\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix}=\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

This matrix, say V, is named after Alexandre-Théophile Vandermonde (1735 – 1796) and may be specified also by $V_{i,j}=x_{i-1}^{j-1}$. One can show that this matrix is non-singular because

$$\det V=\prod_{0\le i<j\le n}(x_j-x_i)=(x_n-x_{n-1})\cdot\ldots\cdot(x_1-x_0)\ne 0$$

when the nodes, x_i, are distinct. So we can, in theory, solve this system.

Write a computer code that builds a Vandermonde matrix or read and use a suitable build-in function, in MATLAB – `vander`, in Python – `numpy.vander`, paying attention to the order of the components.

Then do the following.

a) Test your code using nodes, $\mathbf{x}=(1, 2, 3)$ and values $\mathbf{y}=(3,2,1)$. Find and print coefficients a_i by directly solving the above system of equations then evaluate the interpolation polynomial at the original nodes that must give the original values. Note that the suggested points belong to a straight line implying that $a_2=0$.

b) Repeat item a) using nodes, $\mathbf{x}=(1200.5, 1201.5, 1202.5, 1203.0, 1204.0, 1205.0)$ and values $\mathbf{y}=(3.0, 1.5, 1.5, 1.0, 1.0, 0.0)$. Evaluate the interpolation polynomial at the original nodes and print the norm of the error vector, $\|\mathbf{y}-p(\mathbf{x})\|$ using 1-, 2- and infinite-norm.

Exercise 2. Isaac Newton (1664 – 1727) invented an elegant way how to build the interpolation polynomial in a step-by-step manner using divided differences. Because of this, the Newton form of the interpolation polynomial is sometimes called Newton's divided differences interpolation polynomial. It is a linear combination of Newton's basis polynomials, $\pi_i(x)$:

$$p(x)=\sum_{i=0}^{n} c_i \pi_i(x) \qquad \pi_0(x)\equiv 1 \qquad \pi_i(x)\equiv \prod_{j=0}^{i-1}\left(x-x_j\right) \quad 1\le i\le n$$

The coefficients, c_i, divided differences, can be found recursively as

$$c_0=f[x_0]\equiv f(x_0) \qquad c_i=f[x_0,x_1,\cdots,x_i]\equiv\frac{f[x_1,x_2,\cdots,x_i]-f[x_0,x_1,\cdots,x_{i-1}]}{x_i-x_0}$$

The divided difference theorem gives a simple interpretation of the divided difference of order *k* as the *k*-th derivative at some intermediate point ξ

$$f[x_0,x_1,\cdots,x_k]=\frac{f^{(k)}(\xi)}{k!}$$

Similar to Horner's method, a polynomial written in the Newton form can be effectively evaluated using the following nested representation.

$$p(x)=c_0+c_1(x-x_0)+c_2(x-x_0)(x-x_1)+\ldots+c_n(x-x_0)(x-x_1)\ldots(x-x_{n-1})=$$
$$=c_0+(x-x_0)\Big(c_1+(x-x_1)\Big(\ldots\Big(c_{n-2}+(x-x_{n-2})\Big(c_{n-1}+(x-x_{n-1})(c_n+0)\Big)\Big)\Big)\ldots\Big)$$

Write two computer functions such that one of them calculates coefficients, c_i, (divided differences) and the other evaluates the Newton interpolation polynomial in the nested form. For your inspiration, two suggested functions in Python are given in the Appendix.
Then repeat Exercise 1 using the Newton interpolation polynomial.

Exercise 3. Read and learn how to work with polynomials in one variable using built-in functions. For instance, evaluating a polynomial, $p(x) = x^2 - 4x + 4$, at $x = 3.0$ goes in several steps. In MATLAB, you first define a coefficient vector, `p = [1 -4 4]` (no commas here!), then evaluate, `y = polyval(p,x)`. In Python, you do it in three similar steps. You first define a coefficient list, `a = [1, -4, 4]`, (note commas here) then you define a `poly1d` object, `p = numpy.poly1d(a)`, finally you evaluate, `y = p(x)`.

Verify that polynomials

$$p(x) = 5x^3 - 27x^2 + 45x - 21, \qquad q(x) = x^4 - 5x^3 + 8x^2 - 5x + 3$$

interpolate the data

x	1	2	3	4
y	2	1	6	47

and explain why this does not violate the uniqueness part of the theorem on existence of polynomial interpolation. To this end, do the following steps.

a) Define two objects, p and q, that represent the above polynomials. Set a vector of nodes, x = [1, 2, 3, 4] then evaluate and print these polynomials at the interpolation nodes, x, thus making sure the corresponding built-in functions can work on arrays.

b) Write a computer function that returns Wilkinson's polynomial of degree n $w_n(x) \equiv \prod_{i=1}^{n}(x-i) = (x-1)\cdot(x-2)\cdot\ldots\cdot(x-n)$. Such polynomials with known and well separated roots were used by James H. Wilkinson in 1963 to illustrate a difficulty when finding its roots. Here we will use $w_4(x)$ in a completely different context. The interpolation data states that $q(i) - p(i) = 0$ when i = 1, 2, 3, 4; which in turn suggests that the difference $q(x) - p(x)$ is proportional to $(x-1)(x-2)(x-3)(x-4) = w_4(x)$. Print and compare the coefficients of $q(x) - p(x)$ and those of $w_4(x)$. Note that both MATLAB and Python supports arithmetic operations between polynomials such as addition, subtraction and multiplication.

c) Construct Wilkinson's polynomial of 5th order, say w, and compute its roots. In MATLAB, it's done by, `r = roots(w)`, whereas in Python it is simply `w.r`

d) Finally, build Wilkinson's polynomial of 20th degree then compute and print its roots.

Exercise 4. Let $p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$ be a polynomial of degree n with canonical coefficients a_i. Given nodes $x_0, x_1, \cdots, x_n$ and values $y_0, y_1, \cdots, y_n$, Newton's algorithm efficiently builds coefficients $c_0, c_1, \cdots, c_n$ in Newton's basis $\pi_i(x)$.

$$p(x) = \sum_{i=0}^{n} c_i \pi_i(x) \qquad \pi_0(x) \equiv 1 \qquad \pi_i(x) \equiv \prod_{j=0}^{i-1} (x - x_j) \quad 1 \le i \le n$$

Sometimes one needs just the opposite, i.e. to find the canonical coefficients, a_i, when x_i and c_i are known. To this end, write a computer function that returns an array of the coefficients a_i given x_i and c_i. The simplest way to do so is based on the nested form in Exercise 2. You begin with the innermost parentheses and progressively build the polynomial in question treating summations and multiplications as the operations between polynomial objects. Both MATLAB and Python supports such operations. Finally, perform the following steps.

a) Consider a polynomial, $p(x)$, that interpolates the data, $p(x_i) = y_i$ with the nodes $x_i = [1, 2, 3, 4]$ and the values $y_i = [0, 1, 6, 33]$. Then find the Newton coefficients, c_i, calculate coefficients a_i and report $p(x)$ in the both Newton and canonical forms.

b) What is an obvious relationship between a_n and c_n ?

c) If $n + 2$ distinct points are $s_0, s_1, \cdots, s_n, s_{n+1}$, what is $p[s_0, s_1, \ldots, s_{n+1}]$?

d) If $n + 1$ distinct points are $s_0, s_1, \cdots, s_n$, what is $p[s_0, s_1, \ldots, s_n]$?

Exercise 5. From census data, the approximate population of the United States was 150.7 million in 1950, 179.3 million in 1960, 203.3 million in 1970, 226.5 million in 1980, and 249.6 million in 1990. Using Newton's interpolation polynomial for these data, find an approximate (extrapolated) value for the population in 2000. Compare the extrapolated value with the exact one, 282.2 million in 2000 by evaluating the absolute and relative errors. Then use the polynomial to estimate the population in 1920 based on these data. What conclusion should be drawn? In other words, steps to be done are:

a) Evaluate (extrapolate) the US population in 2000 using the interpolation polynomial (in Newton or Lagrange form at your choice) that is built on the above data.

b) Compare the extrapolated and the exact value, 282.2 million by evaluating the absolute and relative errors.

c) Extrapolate (backward) the US population in 1920 using the same interpolation polynomial and give your conclusion.

Exercise 6. Divided differences is an algorithm historically used for computing tables of logarithms and trigonometric functions. Nowadays, divided differences are frequently used to construct and evaluate interpolation polynomials. However, specific properties of divided differences, such as studied in Exercise 3, may be useful to examine tabulated data, $\{(x_i, y_i) \mid i = 0,1,\cdots,n\}$, directly. To this end, a slightly different but equivalent notation may be used. The divided difference of 0-th order is simply defined as

$$[y_i] \equiv y_i \quad i = 0,1,\cdots,n$$

The (forward) divided difference of order *j* is recursively given by

$$[y_i, y_{i+1}, \cdots, y_{i+j-1}, y_{i+j}] \equiv \frac{[y_{i+1}, \cdots, y_{i+j-1}, y_{i+j}] - [y_i, y_{i+1}, \cdots, y_{i+j-1}]}{x_{i+j} - x_i}$$

Here, the order *j* is squeezed between, $1 \le j \le n$, and the starting index *i* for computing a divided difference of order *j* obeys $0 \le i \le n - j$. Clearly, this recursive definition suggests a simple and elegant recursive algorithm to calculate divided differences. Read the appropriate documentation how to define a recursive function in MATLAB or Python and write your own function that evaluates divided differences based on arrays, x_i and y_i. Then inspect the following table.

x	-2	-1	0	1	2	3
y	1	4	11	16	13	-4

It is suspected to come from a cubic polynomial. Verify this hypothesis using the mean value theorem for the divided differences

$$f[x_i, x_{i+1}, \cdots, x_{i+j}] = \frac{f^{(j)}(\xi)}{j!} \qquad \min_{i \le k \le i+j} x_k < \xi < \max_{i \le k \le i+j} x_k$$

Thus we expect for a cubic polynomial, $p(x)$,

$$p[x_i, x_{i+1}, x_{i+2}, x_{i+3}, x_{i+4}] = \frac{p^{(4)}(\xi)}{4!} = 0 \quad \text{and} \quad p[x_i, x_{i+1}, x_{i+2}, x_{i+3}] = \frac{p^{(3)}(\xi)}{3!} = const$$

Carry out several evaluations for either relationship to check the above hypothesis.

Exercise 7. An alternative form of the interpolating polynomial at a set of fixed nodes, $x_0, x_1, \cdots, x_n$ was suggested by Joseph Louis Lagrange (1736 – 1813) through a system of n + 1 special polynomials of degree n known as cardinal polynomials (functions). They are typically denoted as $\ell_0, \ell_1, \cdots \ell_n$ and defined by the property

$$\ell_i\left(x_j\right) = \delta_{ij} = \begin{cases} 0 & \text{if } i \neq j \\ 1 & \text{if } i = j \end{cases}$$

Once these are available, we can interpolate any function *f* by the Lagrange form of the interpolation polynomial as

$$p_n(x) = \sum_{i=0}^{n} f(x_i)\,\ell_i(x)$$

It is easy to find that the cardinal functions are given by

$$\ell_i(x) = \prod_{\substack{j=0\\ j\neq i}}^{n} \frac{x - x_j}{x_i - x_j} = \frac{x - x_0}{x_i - x_0}\cdot\frac{x - x_1}{x_i - x_1}\cdots\frac{x - x_{i-1}}{x_i - x_{i-1}}\cdot\frac{x - x_{i+1}}{x_i - x_{i+1}}\cdots\frac{x - x_{n-1}}{x_i - x_{n-1}}\cdot\frac{x - x_n}{x_i - x_n}$$

The interpolation error is given by a theorem that states

$$f(x) - P(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!}(x - x_0)\cdot(x - x_1)\cdot\ldots\cdot(x - x_n)$$

Write a computer code that implements the cardinal functions and the Lagrange form of the interpolating polynomial based on nodes, x_i, and values, $y_i = f(x_i)$. Then perform the following steps.

a) For a better insight into Lagrange interpolation, plot all the 5 cardinal functions defined on the nodes, [-1, -0.5, 0, 0.5, 1]

b) Interpolate the exponential function, e^x, at the same nodes and plot the original function, e^x, in a solid line, and the Lagrange interpolant on the interval, [-3, 2] in a broken line.

c) Assess the interpolation error at $x = 0.25$ using the above formula. In doing so, you will need to estimate $\max\limits_{\xi}\left|f^{(n+1}(\xi)\right|$

d) Evaluate the actual absolute and relative interpolation error at the same node and verify if it obeys the theoretical estimate.

Exercise 8. When interpolating data points $(x_0, y_0), (x_1, y_1), \cdots, (x_n, y_n)$ with a polynomial of degree at most *n*, one can use the standard (natural) basis of monomials $\{1, x, x^2, \cdots, x^n\}$. Another choice is the Newton polynomials, $\{\pi_0(x), \pi_1(x), \cdots, \pi_n(x)\}$. Finally, we have explored the Lagrange cardinal polynomials, $\{\ell_0(x), \ell_1(x), \cdots, \ell_n(x)\}$. It turns out that there are better choices for the basis functions. One of them is the Chebyshev polynomials which have more desirable features. They can be defined recursively as

$$\begin{cases} T_0(x) = 1, \quad T_1(x) = x \\ T_k(x) = 2xT_{k-1}(x) - T_{k-2}(x) \end{cases}$$

We know pretty much about these polynomials. For instance, the roots of $T_n(x)$ are located within the interval, [-1, 1], and are given by

$$z_k = \cos\left(\frac{k + 1/2}{n}\pi\right) \quad k = 0, 1, \cdots, n-1 \quad T_n(z_k) = 0$$

All the extrema on [-1, 1] are either 1 or -1 and they are located at

$$m_k = \cos\left(\frac{k}{n}\pi\right) \quad k = 0, 1, \cdots, n \quad T_n(m_k) = (-1)^k$$

Another unique property of the Chebyshev polynomials is that the roots of $T_{n+1}(x)$ minimize the term $\prod_{k=0}^{n}(x - x_k)$ that occurs in the interpolation error formula, i.e.

$$\min_{x_k} \max_{-1 \le x \le 1} \left| \prod_{k=0}^{n} (x - x_k) \right| = \max_{-1 \le x \le 1} \left| \prod_{k=0}^{n} (x - z_k) \right| = 2^{-n}$$

Armed with this piece of knowledge, carry out the following steps.

a) Plot the first 5 Chebyshev polynomials. It might be helpful first reading how to multiply polynomials in MATLAB/Python.

b) Plot the product, $\prod_{k=0}^{8}(x - x_k)$, for 9 even spaced nodes x_k on [-1, 1], the built-in function `linspace` might be helpful here.

c) Set the nodes, x_k, to be the roots of T_9 and plot the same product again.

Exercise 9. A process called inverse interpolation is often used to approximate an inverse function. Suppose that values $y_i = f(x_i)$ have been computed at $x_0, x_2, \cdots, x_n$. Regarding the variable *y* as independent and *x* as dependent, we first form a table

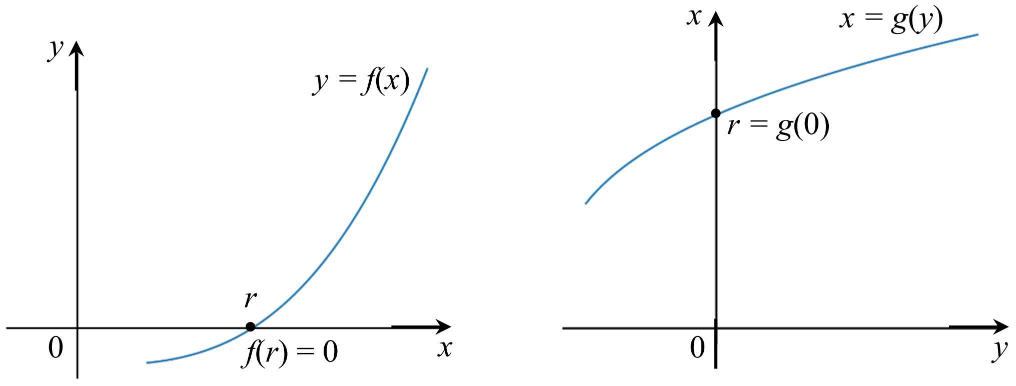
y	y_0	y_1	$\cdots$	y_n
x	x_0	x_1	$\cdots$	x_n

Then we construct the interpolation polynomial

$$p(y) = \sum_{i=0}^{n} c_i \prod_{j=0}^{i-1} (y - y_j)$$

The original relationship, $y = f(x)$, has an inverse, $x = f^{-1}(y)$, under certain condition. This inverse is being approximated by $x = p(y)$. Computer functions from Exercise 2 can be used to carry out the inverse interpolation by reversing the arguments *x* and *y* in the calling sequence.

Inverse interpolation can be used to find where a given function *f* has a root or zero. This means inverting the equation, $f(x) = 0$. It is recommended to do this by creating a table of values, $(f(x_i), y_i)$, and interpolating with a polynomial $p(y)$ such that, $p(y_i) = x_i$. The points x_i, should be chosen near the unknown root, *r*. The root is then approximately given by $r \approx p(0)$. See the next figure for an example of function $y = f(x)$ and its inverse $y = g(x)$.



For a concrete case, let the table of known values be

y	-0.5789200	-0.3626370	-0.1849160	-0.0340642	0.0969858
x	1.0	2.0	3.0	4.0	5.0

Carry out the following steps.

a) Plot a (direct) interpolation polynomial of a suitable degree and determine visually where the underlying function has zero (root).

b) For the above data, find the inverse interpolation polynomial, $x = p(y)$ and report its canonical coefficients.

c) Find an approximation of the root by inverse interpolation, i.e. by evaluating the inverse interpolation polynomial at zero, $r = p(0)$.

Exercise 10. When speaking about polynomials, it is difficult to avoid making reference to the fundamental theorem of algebra, which states that every non-zero, single-variable, degree *n* polynomial with complex coefficients has, counted with multiplicity, exactly *n* complex roots. It allows proving many important statements without doing much algebra. A typical and simplified example is as follows. Let two polynomials be

$$p(x) = ax^2 + bx + c \quad \text{and} \quad q(x) = A(x - z_1)(x - z_2)$$

If it is known that they pass through three points $(x_0, y_0), (x_1, y_1), (x_2, y_2)$, then they must be equivalent to each other. A straightforward way of proving this is to establish equivalence of their canonical coefficients, which involves a good deal of algebraic manipulations. On contrary, the fundamental theorem simply says that $p(x) - q(x)$ has three roots at x_0, x_1, x_2 according to the condition. On the other hand, $p(x) - q(x)$ is clearly a polynomial with a degree at most 2. Hence the only possibility left is when this difference is totally equal to zero (zero function).

Armed with this knowledge prove the following.

a) The Newton and Lagrange polynomials that interpolate the same data, $(x_0, y_0), (x_1, y_1), \cdots, (x_n, y_n)$ are equivalent to each other, i.e.

$$\sum_{i=0}^{n} c_i \pi_i(x) = \sum_{i=0}^{n} y_i \ell_i(x)$$

b) The Lagrange cardinal functions obey the identity $\sum_{i=0}^{n} \ell_i(x) = 1$

Appendix

```
import numpy as np
import matplotlib.pyplot as plt
from numpy import cos, exp, sqrt, abs, log, log2, log10, floor, ceil,
round
from numpy import pi
from numpy.linalg import norm
from math import inf, factorial

def newcoef(xn,yn):
   """ Newton coefficients (divided differences)
   Input
      xn - nodes  xn[i] (array-like)
      yn - values yn[i] (array-like)
   Output
      c - coefficients a[i] (ndarray of float type)
   """
   c = np.array(yn,dtype = float)
   n = len(xn)
   for j in range(1,n):
      for i in range(n-1,j-1,-1):
         c[i] = (c[i] - c[i-1])/(xn[i] - xn[i-j])
   return c

def newpoly(c,xn,x):
   """ Newton polynomial p(x)
   Input
      c  - coefficients c[i] calculated in newcoef
      xn - nodes xn[i] (array-like)
      x  - number or list/array of numbers (typically real)
   Output
      p = p(x) - Newton's polynomial evaluated at x (float)
   """
   n = len(xn) - 1 # Now n is degree of polynomial.
   p = 0.0
   for i in range(n,-1,-1):
      p = c[i] + (x - xn[i])*p
   return p

def new2can(c,xn):
   """ Convert coefficients of Newton's polynomial, c, to canonical
coefficients, a.
   Input
      c  : coefficients of Newton's polynomial
      xn : x-nodes
   Output
      a  : coefficients of canonical polynomial
   """
   n = len(xn)
   p = np.poly1d([0])
   for i in range(n-1,-1,-1):
      b = np.poly1d([1.0, -xn[i]]) # Binomial (x - xn[i])
      p = p*b + c[i]
   return p.c


def newdd(xn,yn):
   """ Newton's Divided Differences by recurrence
      Input
         xn : x-nodes xn[i]
         yn : y-nodes yn[i]
      Output
         [y0,y1,...,y(n-1)] : Divided differences or order (n-1)
   """
   n = len(yn)
   if n == 1:
      return yn[0]

   return (newdd(xn[1:],yn[1:]) - newdd(xn[:n-1],yn[:n-1]))/(xn[-1] -
xn[0])

def binprod(x,xn): # Binomials product = (x-x[0])*(x-x[1])*...*(x-
x[n])
   p = 1.0
   for i in range(len(xn)):
      p *= (x-xn[i])
   return p

def lacard(i,xn,x): # Lagrange cardinal polynomial
   n = len(xn)-1
   if i > n:
      print('i = %d > %d = n-1 (last index)'%(i,n))
      exit()
   p = 1.0
   for j in range(i):
      p *= (x - xn[j])/(xn[i] - xn[j])
   for j in range(i+1,n+1):
      p *= (x - xn[j])/(xn[i] - xn[j])
   return p

def lagran(xn,yn,x): # Lagrange interpolation polynomial
   p = 0.0
   for i in range(len(xn)):
      p += yn[i]*lacard(i,xn,x)
   return p

def wilpoly(n): # Wilkinson's polynomial of degree n
   w = np.poly1d([1])
   for i in range(1,n+1):
      bx = np.poly1d([1,-i]) # Binomial (x-i)
      w *= bx
   return w
```