

ΑΣΚΗΣΗ 5

α) Στην πολυωνυμική προσέγγιση χρησιμοποίησα την μέθοδο του Lagrange. Στο πρόγραμμα μου έκανα 2 ξεχωριστές συναρτήσεις από τις οποίες η μία υπολογίζει με την μέθοδο του Lagrange ένα συγκεκριμένο ημίτονο (και το όνομα που της έδωσα είναι lagrangeInterpolationForAnX), και η άλλη χρησιμοποιεί όλα τα σημεία του x ώστε να δημιουργήσει το γράφημα με την μέθοδο του Lagrange (και το όνομα που της έδωσα είναι makeInterpolationAndThenShowError). Η πρώτη συνάρτηση που απλά υπολογίζει το ημίτονο (lagrangeInterpolationForAnX) δημιουργεί έναν πίνακα x με δέκα τιμές χρησιμοποιώντας το linspace της numpy και έναν πίνακα y με τα ημίτονα των x. Στην συνέχεια αρχικοποιεί την τιμή του πολυωνύμου με μηδέν και με διπλό for loop χρησιμοποιεί τον τύπο του Lagrange κάθε φορά που i != j και ως x χρησιμοποιεί το theta το οποίο είναι η γωνία σε radians και μετά καλεί την συνάρτηση plotFunctionWithComputedX η οποία κάνει plot τις τιμές του ημιτόνου από -π εώς π και κάνει scatter plot την καινούργια υπολογισμένη τιμή του ημιτόνου.

```
def lagrangeInterpolationForAnX(theta, numberOfPoints):
    x = np.linspace(-np.pi, np.pi, numberOfPoints)
```

```
y = np.sin(x)
```

```
P = 0 # initial polynomial value
```

```
n = len(x) # the degree of the polynomial
```

```
for i in range(n):
```

```
    L = 1
```

```
    for j in range(n):
```

```
        if i != j:
```

```
            L *= (theta - x[j]) / (x[i] - x[j])
```

```
P += L * y[i]
```

```
plotFunctionWithComputedX(x, y, theta, P)
```

```
def plotFunctionWithComputedX(x, y, newX, newY):
```

```
xPoints = np.array([-np.pi, np.pi], np.float)
```

```
xPoints = np.linspace(xPoints[0], xPoints[-1], 200)
```

```
yPoints = np.sin(xPoints)
```

```
plt.scatter(newX, newY, c='green', marker='o', s=80)
```

```
plt.plot(x, 'ro', xPoints, yPoints, 'b-')
```

```
plt.xlabel("x")
```

```
plt.ylabel("f(x)")
```

```
plt.grid(linestyle="dotted")
```

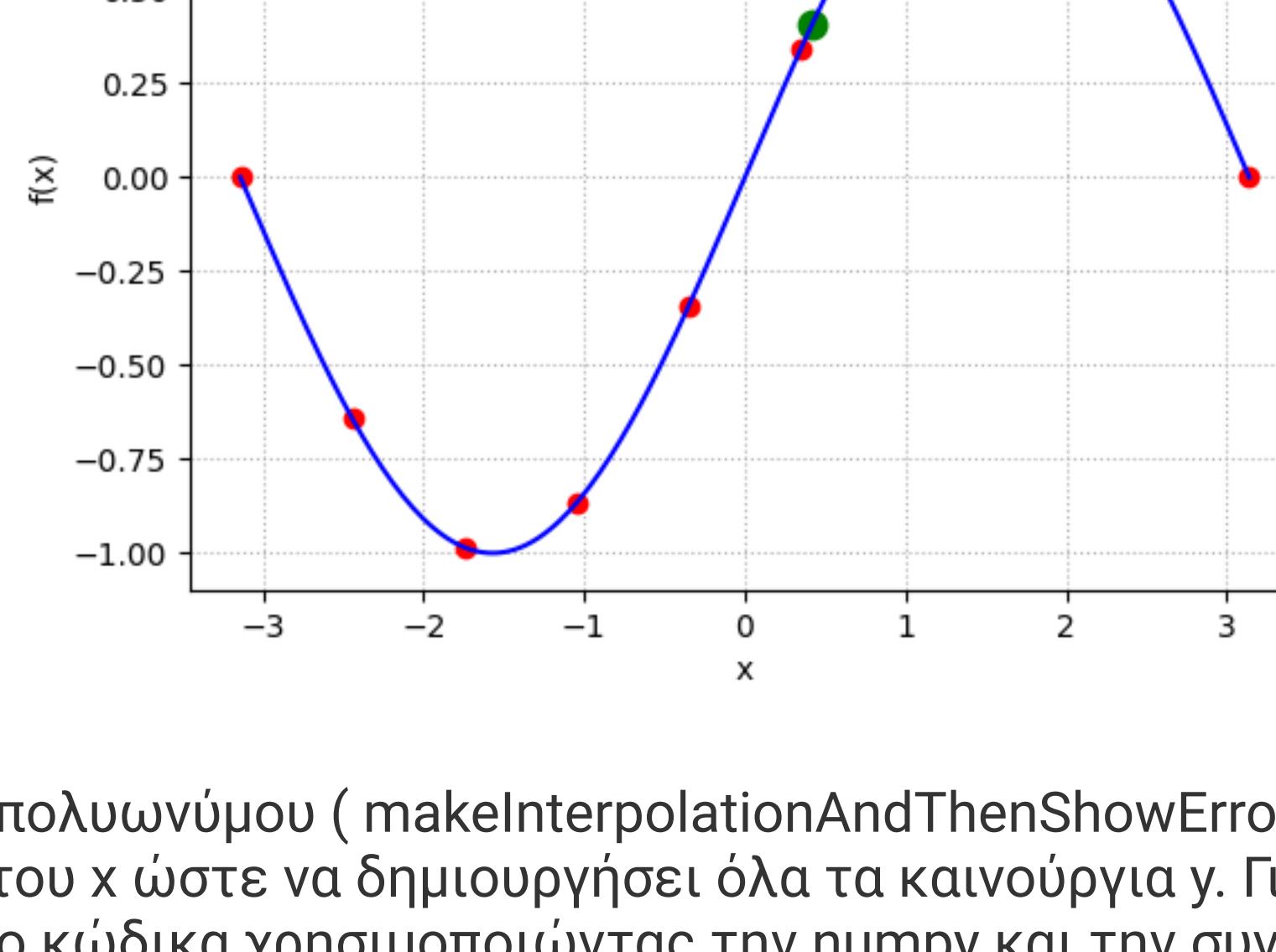
```
plt.legend(loc="upper center")
```

```
plt.show()
```

$$P(x) = \sum_{i=0}^n p_i(x) y_i \quad p_i(x) = \prod_{j=0, j \neq i}^n \frac{(x - x_j)}{(x_i - x_j)}$$

The algorithm of the Lagrange's interpolation

$$P(x) = \sum_{i=0}^n \left(\prod_{j=0, j \neq i}^n \frac{(x - x_j)}{(x_i - x_j)} \right) y_i$$



Η δεύτερη συνάρτηση που δημιουργεί το γράφημα του πολυωνύμου (makeInterpolationAndThenShowError) λειτουργεί ακριβώς το ίδιο μόνο που χρησιμοποιεί όλα τα σημεία του x ώστε να δημιουργήσει όλα τα καινούργια y. Για να μην κάνω τριπλό loop χρησιμοποίησα λίγο πιο προχωρημένο κώδικα χρησιμοποιώντας την numpy και την συνάρτηση zip.

```
def makeInterpolationAndThenShowError(numberOfPoints):
```

```
x = np.linspace(-np.pi, np.pi, numberOfPoints)
```

```
fx = np.sin(x)
```

```
y = []
```

```
n = len(x)
```

```
for xp in x:
```

```
L = np.sum([yi * np.prod([(xp - xi) / (xi - xj) for xi, yi in zip(x, fx)]) for xi, yi in zip(x, fx)])
```

```
y.append(L)
```

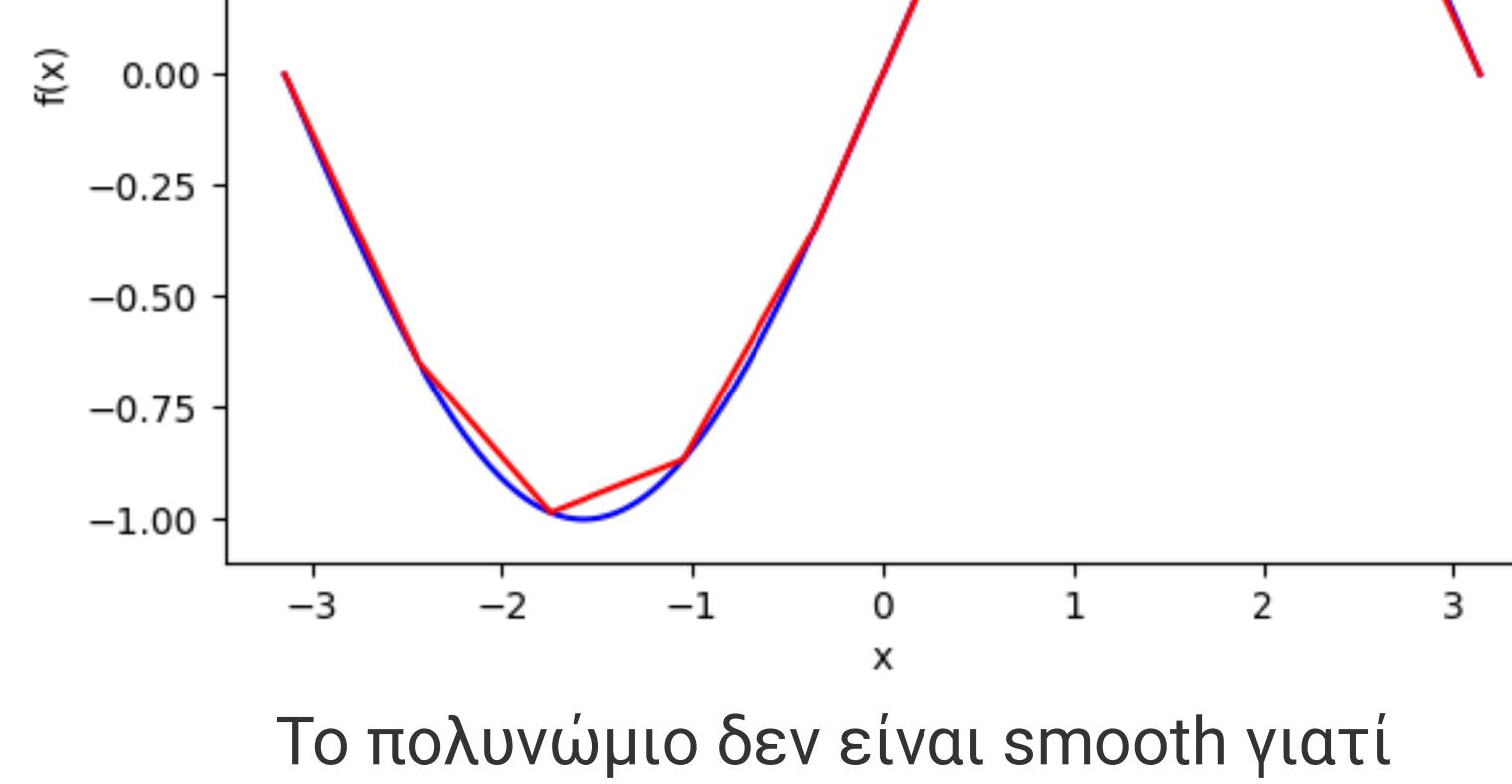
```
plotPolynomialWithError(x, y)
```

Διαλέγει το x αν το x δεν είναι ίδιο με το xi

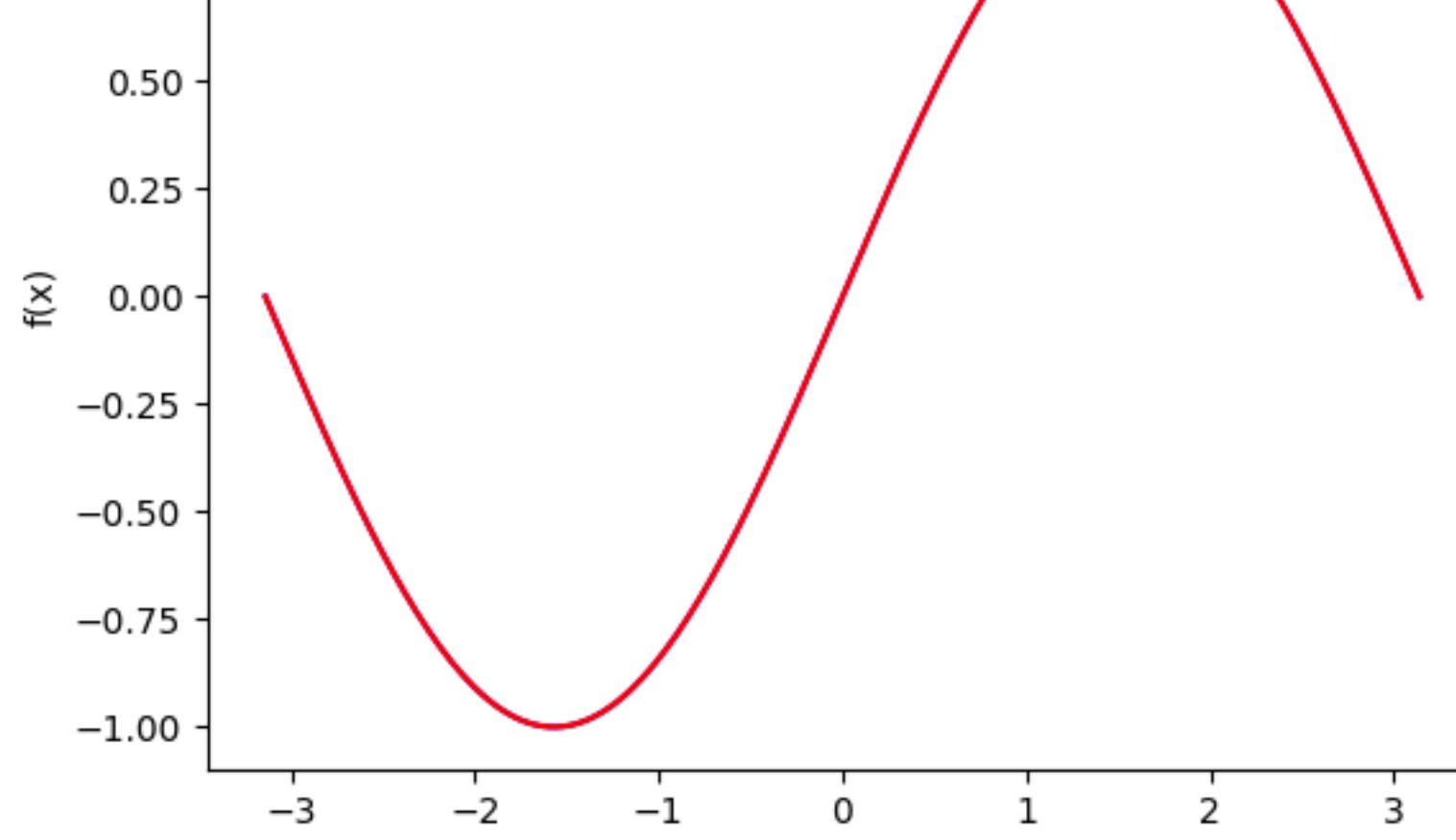
```
def plotPolynomialWithError(x, y):
```

```
xPoints = np.linspace(-np.pi, np.pi, 200)
yPoints = np.sin(xPoints)
```

```
plt.plot(xPoints, yPoints, 'b-', label="sin(x)")
plt.plot(x, 'r', label='polynomial')
plt.xlabel("x")
plt.ylabel("f(x)")
plt.grid()
plt.legend(loc="upper left")
plt.show()
```



Το πολυνώμιο δεν είναι smooth γιατί επέλεξα μόνο 10 σημεία. Αμά όμως το δοκιμάσω με διακόσια σημεία τότε οι συναρτήσεις έχουν σφάλμα σχεδόν ίσο με το μηδέν.



Αυτά είναι τα γραφήματα με 200 σημεία από -π έως π. Από ότι φαίνεται η προσέγγιση με Lagrange έκανε μια αρκετά καλή προσέγγιση. Πρέπει να κάνουμε αρκετές φορές zoom για να δούμε την μπλε γραμμή.

β) Στα cubic splines δημιουργησα 4 συναρτήσεις, την coefficients() η οποία υπολογίζει τα a,b,c και d των S πολυωνύμων, την S() η οποία υπολογίζει το spline, την η οποία χρησιμοποιει αυτές τις 2 συναρτήσεις για να υπολογίσει όλα τα splines και τέλος την συνάρτηση plotError() η οποία κάνει plot τα splines και την κανονική συνάρτηση. Για τα splines χρησιμοποίησα τα natural splines, και οι τύποι οι οποίοι χρησιμοποίησα είναι οι εξής:

$$1) \quad a_k = y_k, \quad b_k = \frac{y_{k+1} - y_k}{h_k} - \frac{2y''_k + y''_{k+1}}{6} h_k, \quad c_k = \frac{y''_k}{2}, \quad d_k = \frac{y''_{k+1} - y''_k}{6h_k}$$

$$2) \quad S_k(x) = a_k + b_k(x - x_k) + c_k(x - x_k)^2 + d_k(x - x_k)^3, \quad k = 0, \dots n - 1$$

$$3) \quad \begin{pmatrix} 2(h_0 + h_1) & h_1 & & & \\ h_1 & 2(h_1 + h_2) & h_2 & & \\ & h_2 & 2(h_2 + h_3) & h_3 & \\ & & \ddots & \ddots & \\ & & h_{n-3} & 2(h_{n-3} + h_{n-2}) & h_{n-2} \\ & & & h_{n-2} & 2(h_{n-2} + h_{n-1}) \end{pmatrix} \cdot \begin{pmatrix} y''_1 \\ y''_2 \\ y''_3 \\ \vdots \\ y''_{n-2} \\ y''_{n-1} \end{pmatrix} = \begin{pmatrix} 6\left(\frac{y_2 - y_1}{h_1} - \frac{y_1 - y_0}{h_0}\right) - h_0 y''_0 \\ 6\left(\frac{y_3 - y_2}{h_2} - \frac{y_2 - y_1}{h_1}\right) \\ \vdots \\ 6\left(\frac{y_n - y_{n-1}}{h_{n-1}} - \frac{y_{n-1} - y_{n-2}}{h_{n-2}}\right) - h_{n-1} y''_n \end{pmatrix}$$

Η συνάρτηση coefficients δέχεται τα yk, τα hk και τα yppk (ρρ σημαίνει double prime). Η συνάρτηση αυτή εκτελεί την μέθοδο 1 που φαίνεται πιο πάνω και επιστρέφει ένα tuple με όλα τα coefficients.

```
def coefficients(yk, hk, yppk):
```

```
    ak = yk
    bk = np.array([(yk[i+1] - yk[i])/hk[i] - (2.*yppk[i] + yppk[i + 1])*hk[i]/6 for i in range(len(hk))], np.float)
    ck = yppk * 0.5
    dk = np.array([(yppk[i + 1] - yppk[i])/(hk[i]*6) for i in range(len(hk))], np.float)
    return (ak, bk, ck, dk)
```

Η συνάρτηση S δέχεται ένα συγκεκριμένο x, τα a,b,c και d coefficients και τα xk και αφού υπολογίσει μέχρι πιο xk είναι το x μεγαλύτερο (δηλαδή θέλουμε την διαφορά με το πρώτο xk[i] που θα είναι μικρότερο του x), αποθηκεύει την διαφορά τους σε μία μεταβλητή dx = x - xk[i], και επιστρέφει την τιμή του spline χρησιμοποιώντας την μέθοδο 2 που φαίνεται πιο πάνω αλλά με τον τρόπο του Horner (για computational efficiency).

```
def S(x, a, b, c, d, xk):
```

```
    i = 0
    while i < len(xk) - 1 and x >= xk[i]: # find the interval that contains x
        i += 1
    i -= 1
    dx = x - xk[i]
    return a[i] + dx*(b[i] + dx*(c[i] + d[i]*dx)) # Horner's method for computational efficiency: a + x*(b + x*(c + x*d))
```

Η συνάρτηση cubicSplines αρχικοποιεί έναν πίνακα xk με τα σημεία που επέλεξα, έναν πίνακα yk με τα ημίτονα των xk, έναν πίνακα hk ο οποίος έχει τις διαφορές των x σημείων και μια μεταβλητή n = len(xk) - 1. Επίσης αρχικοποιεί σε μεταβλητές ypp0 και yppn τα natural splines (S''(x0) = 0, S''(xn) = 0), και τέλος αρχικοποιεί δύο πίνακες A και b τους οποίους θα χρησιμοποιήσει για να λύσει την μέθοδο 3 που φαίνεται πιο πάνω. Αφού περάσει τις σωστές τιμές στους πίνακες αυτούς με ένα for loop χρησιμοποιεί την συνάρτηση np.linalg.solve(A, b) για να πάρει τα ypp (ρρ σημαίνει double prime) και μετά καλεί την συνάρτηση coefficients ώστε να πάρει τα a,b,c,d (τα οποία θα είναι πίνακες με τα coefficients) και θα τα περάσει στην συνάρτηση plotError για να κάνει plot την συνάρτηση με τα splines.

```
def cubicSplines():
```

```
    xk = np.array([-np.pi, -3, -2.97, -1.97, -0.32, 0.13, 1.4, 2.1, 2.9, 3.1], np.float)
```

```
    yk = np.sin(xk)
```

```
    n = len(xk) - 1
```

```
    hk = np.array([xk[i + 1] - xk[i] for i in range(n)])
```

```
    ypp0 = 0 # natural splines y0" = 0
```

```
    yppn = 0 # natural splines yn" = 0
```

```
    A = np.zeros((n - 1, n - 1))
```

```
    b = np.zeros(n - 1)
```

```
    for i in range(n - 1):
```

```
        A[i, i] = 2 * (hk[i] + hk[i + 1])
```

```
        if i > 0:
```

```
            A[i, i - 1] = hk[i]
```

```
        if i < n - 2:
```

```
            A[i, i + 1] = hk[i + 1]
```

```
        b[i] = 6 * ((yk[i + 2] - yk[i + 1]) / hk[i + 1] - (yk[i + 1] - yk[i]) / hk[i])
```

```
    b[0] -= hk[0] * ypp0
```

```
    b[-1] -= hk[n - 1] * yppn
```

```
    x = np.linalg.solve(A, b)
```

```
    ypp = np.insert(x, (0, len(x)), (ypp0, yppn))
```

```
    coeffs = coefficients(yk, hk, ypp)
```

```
    plotError(coeffs, xk)
```

```
def plotError(coeffs, xk):
```

```
    x_splines = np.linspace(-np.pi, np.pi, 200)
```

```
    a, b, c, d = coeffs
```

```
    spline = [S(x,a,b,c,d,xk) for x in x_splines]
```

```
    x_sinx = np.linspace(-np.pi, np.pi, 200)
```

```
    y_sinx = np.sin(x_sinx)
```

```
    plt.title("Cubic spline interpolation")
```

```
    plt.xlabel("x")
```

```
    plt.ylabel("y")
```

```
    plt.grid(linestyle="dotted")
```

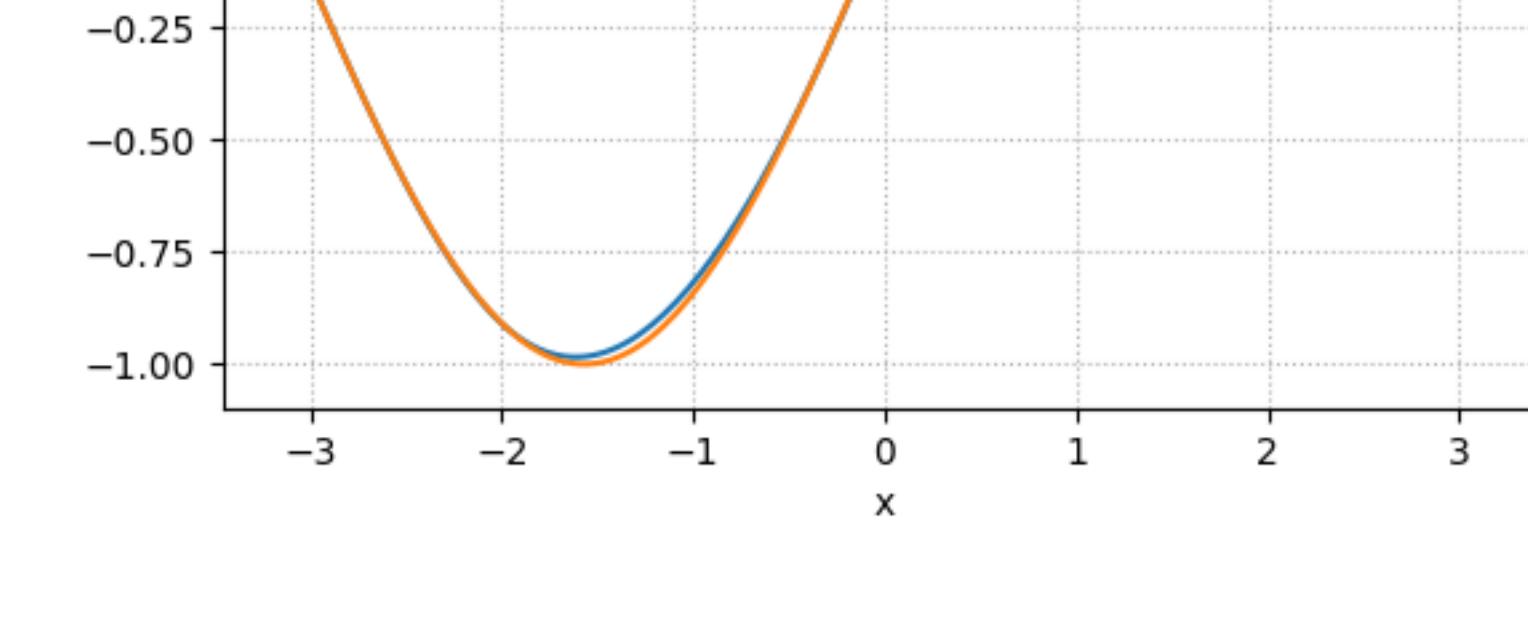
```
    plt.plot(x_splines,spline, label="polynomial")
```

```
    plt.plot(x_sinx, y_sinx, label="sin(x)")
```

```
    plt.legend(loc="upper center")
```

```
    plt.show()
```

Όπως φαίνεται το σφάλμα μεταξύ τους είναι πολύ μικρό όταν το τεστάρουμε για 200 σημεία ανάμεσα στο -π έως το π.



γ) Στο least squares χρησιμοποίησα τους παρακάτω τύπους για να λύσω την εξίσωση $y = mx + b$:

$$m = \frac{n(\sum xy) - (\sum x)(\sum y)}{n(\sum x^2) - (\sum x)^2}$$

← Από το κανάλι Krista King

$$b = \frac{(\sum y) - m(\sum x)}{n}$$

$$y = mx + b$$

Στο πρήβλημα αυτό χρησιμοποίησα 2 συναρτήσεις, την `leastSquares` που δέχεται μία γωνία σε radians και υπολογίζει τις τιμές της εξίσωσης $y = mx + b$ και βρίσκει το καινούργιο y της γωνίας `theta` και καλεί την δεύτερη συνάρτηση `plotSlope` με παραμέτρους τα m, b, x, y ώστε να κάνει scatter plot όλες τις τιμές μαζί με το slope.

`def leastSquares(theta):`

`x = np.array([-np.pi, -3, -2.97, 3.1, 1.3, 0, -1.3, np.pi, 3, 1.5], np.float)`

`x = np.append(x, theta)`

`x.sort()`

`y = np.sin(x)`

`xy = x * y`

`x_square = x**2`

`n = len(x)`

`m = (n * np.sum(xy) - np.sum(x)*np.sum(y)) / (n * np.sum(x_square) - np.sum(x)**2)`

`b = (np.sum(y) - m * np.sum(x)) / n`

`plotSlope(m, b, x, y)`



`def plotSlope(m, b, x, y):`

`for i in range(len(x)):`

`plt.scatter(x[i], y[i], c='orange', marker='o', s=80)`

`slope = m * x + b`

`error = y - slope`

`plt.plot(x, slope, label="least squares")`

`plt.plot(x, error, label="error")`

`plt.title("Least Squares")`

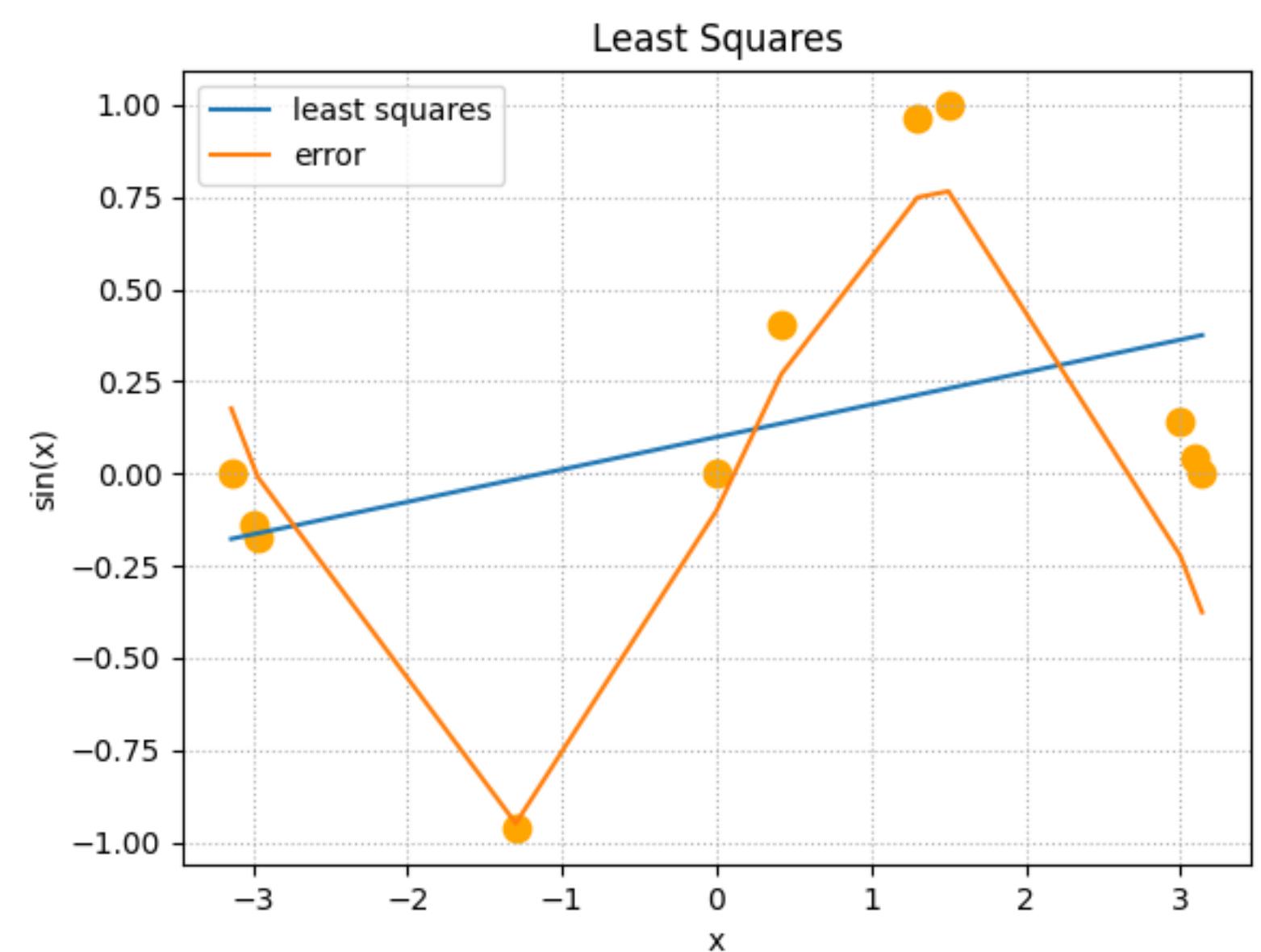
`plt.xlabel("x")`

`plt.ylabel("sin(x)")`

`plt.grid(linestyle="dotted")`

`plt.legend(loc="upper left")`

`plt.show()`



ΑΣΚΗΣΗ 6

Στην μέθοδο του Simpson έκανα μια συνάρτηση με όνομα simpson η οποία δέχεται ένα διάστημα με τιμές των x και υπολογίζει το ολοκλήρωμα της συνάρτησης χρησιμοποιώντας τον τύπο του Simpson και επιστρέφει την τιμή. Αφού καλέσω την συνάρτηση και πάρω το ολοκλήρωμα υπολογίζω το κανονικό ολοκλήρωμα και μετά αφαιρώ αυτές τις 2 τιμές για να βρω το σφάλμα. Έκανα το ίδιο και για την μέθοδο του τραπεζίου. Ο κώδικας των συναρτήσεων του simpson και του τραπεζίου είναι οι εξής:

```
def simpson(x):
    n = len(x)

    a = x[0]
    b = x[n - 1]
    h = (b - a) / n

    integral = (h/3)*(f(a) + 4*np.sum(x[1:(n - 1):2]) + 2*np.sum(x[2:(n - 1):2]) + f(b))

    return integral
```

```
def f(x):
    return np.sin(x)
```

```
def trapezoidal(x):
    n = len(x)

    a = x[0]
    b = x[n - 1]
    h = (b - a) / n

    integral = (h/2)*(f(a) + 2*np.sum(x[1:(n - 1)]) + f(b))

    return integral
```

Μέθοδος simpson:

```
x = np.linspace(0.5, np.pi/2, 11)

integral_f = -np.cos(x[len(x) - 1]) + np.cos(x[0])
print("Integral of f(x) is: " + str(integral_f))

integral_p = simpson(x)
print("Integral using simpson rule is: " + str(integral_p))

error = np.fabs(integral_p - integral_f)

print("error = " + str(error))
```

Το σφάλμα θεωρητικά είναι:

$$-\frac{h^4}{180}(b - a)f^{(4)}(\xi)$$

Ενώ αριθμητικά:

```
Windows@DESKTOP-2PA4JPJ MINGW64 ~/Desktop/Analysi 2
s/Windows/Desktop/Analysi 2/askisi2/simpson.py"n/Pyth
Integral of f(x) is: 0.8775825618903726
Integral using simpson is: 0.9887205707974754
error = 0.11113800890710279
```

Μέθοδος τραπεζίου:

```
x = np.linspace(0.5, np.pi/2, 11)

integral_f = -np.cos(x[len(x) - 1]) + np.cos(x[0])
print("Integral of f(x) is: " + str(integral_f))

integral_p = trapezoidal(x)
print("Integral using trapezoidal rule is: " + str(integral_p))

error = np.fabs(integral_p - integral_f)

print("error = " + str(error))
```

Το σφάλμα θεωρητικά είναι:

$$\text{error} = -\frac{(b - a)^3}{12N^2}f''(\xi)$$

Ενώ αριθμητικά:

```
Windows@DESKTOP-2PA4JPJ MINGW64 ~/Desktop/Analysi 2
s/Windows/Desktop/Analysi 2/askisi2/trapezoidal.py"t
Integral of f(x) is: 0.8775825618903726
Integral using trapezoidal rule is: 0.979126060679772
error = 0.10154349878940006
```

ΑΣΚΗΣΗ 7

Οι μετοχές που επέλεξα για αυτή την άσκηση είναι οι Cener και TITC. Οι αρχικές μέρες με τις οποίες ξεκίνησα (πριν τα γενέθλια μου 15 Αυγούστου) είναι οι 3,4,5,6,7,10,11,12,13 και 14 Αυγούστου 2020, τις οποίες αποθήκευσα σε πίνακα x_1 για την μετοχή Cener και x_2 τις ίδιες ακριβώς μέρες για την μετοχή TITC. Για τα γενέθλια την ακριβεία των πολυωνύμων αποθήκευσα σε πίνακες x_{1_next} και y_{1_next} όπως και x_{2_next} και y_{2_next} τις επόμενες 6 ημέρες οι οποίες είναι οι 17,18,19,20,21 και 24 Αυγούστου μαζί με τις τιμές εκείνων των ημερών. Αυτοί είναι οι πίνακες:

```
# First stock
x_1 = [3, 4, 5, 6, 7, 10, 11, 12, 13, 14] # Cener 3/8/2020 έως 14/8/2020
y_1 = [0.93, 0.944, 0.95, 0.947, 0.924, 0.876, 0.915, 0.934, 0.931, 0.94]
```

```
x_1_next = [17, 18, 19, 20, 21, 24]
y_1_next = [0.938, 0.92, 0.93, 0.932, 0.931, 0.913]
```

```
# Second stock
x_2 = [3, 4, 5, 6, 7, 10, 11, 12, 13, 14] # TITC 3/8/2020 έως 14/8/2020
y_2 = [11.3, 11.26, 11.42, 11.4, 11.24, 11.2, 11.2, 11.26, 11.34, 11.3]
```

```
x_2_next = [17, 18, 19, 20, 21, 24]
y_2_next = [11.3, 11.4, 11.34, 11.22, 11.18, 11.2]
```

Στο πρόγραμμα μου έκανα 6 συναρτήσεις, την polynomialRegression η οποία δέχεται ως παραμέτρους έναν πίνακα x , έναν πίνακα y και τον βαθμό του πολυωνύμου και υπολογίζει και επιστρέφει τα coefficients a, b, c και d , την getXMatrix η οποία δέχεται έναν πίνακα x και τον βαθμό του πολυωνύμου και κατασκευάζει τον πίνακα με τα coefficients, την computePolynomialForX η οποία δέχεται coefficients και μια συγκεκριμένη τιμή x και υπολογίζει την τιμή του πολυωνύμου και τέλος την plotPolynomial η οποία κάνει plot τα σημεία των πινάκων μαζί με το πολυώνυμο στις συγκεκριμένες ημέρες και κάθε φορά που οι μέρες αυξάνονται αυτό αλλάζει. Επίσης έκανα δύο συναρτήσεις οι οποίες χρησιμοποιούν τις παραπάνω συναρτήσεις για να πάρουν το πολυώνυμο της συγκεκριμένης μετοχής που χρησιμοποιούν, και τα ονόματα που τις έδωσα είναι CenerStocks και TitcStocks.

```
def polynomialRegression(x, y, degree):
    X = getXMatrix(x, degree) ←
    # (X_transpose*X)^-1 * X_transpose * y
    coeffs = np.matmul(np.matmul(np.linalg.inv(np.matmul(np.transpose(X), X)), np.transpose(Y)), y)

    return coeffs
```

```
def getXMatrix(x, degree):
    X = []
    n = len(x)

    for i in range(n):
        row = []
        for j in range(degree + 1):
            row.append(x[i]**j)
        X.append(row)

    return np.array(X)
```

```
def plotPolynomial(x, y, coeffs, title):
    points = len(x)

    # Stock Values Scatter Plot
    for i in range(points - 1):
        plt.scatter(x[i], y[i], c='blue', marker='o', s=50)

    # Polynomial Graph
    y_regression = [computePolynomialForX(coeffs, xi) for xi in x]
    plt.plot(x, y_regression, 'g-')

    # Actual Closing Price in day n (last element of x)
    plt.scatter(x[points - 1], y[points - 1], c='orange', marker='o', s=75)

    # New estimated closing price using the polynomial
    plt.scatter(x[points - 1], y_regression[points - 1], c='red', marker='o', s=75)

    plt.xlabel("x")
    plt.ylabel("y")
    plt.grid(linestyle="dotted")
    plt.title(title)
    plt.show()
```

```
def computePolynomialForX(coeffs, x):
    return np.sum([coeffs[i]*(x**i) for i in range(len(coeffs))])
```

