МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
Факультет прикладной математики и инворматики
Кафедра вычислительной математики

Отчет по лабораторной работе 3
"Интерполирование функций"
Вариант 5

Выполнил:
Карпович Артём Дмитриевич
студент 3 курса 7 группы

Преподаватель:
Репников Василий Иванович

Минск, 2024

Задача 1

На отрезке $[a, b] = [0, 2]$ задана таблица значений функции $f(x)$ с шагом $h = \frac{1}{10}$. Погрешность каждого заданного значения не превышает $e = 10^{-4}$. Используя интерполирования Ньютона для начала и конца таблицы, с помощью многочленов минимальной степени построить таблицу значений функции

$$f(x) = x \ln(x + 2)$$

с шагом $0.5h = \frac{1}{20}$. Погрешность каждого нового значения также не должна превышать заданной величины $e = 10^{-4}$.

[1]:
```
import math
import numpy as np
import scipy.integrate as integrate
import scipy.special as special
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd

pd.options.display.float_format ='{:,.7f}'.format
```

Определим нашу функцию.

[2]:
```
def f(x):
    return x * np.log(x + 2)
```

Перейдем к построению таблицы значений функции $f(x)$ на отрезке $[0, 2]$ с шагом $h = \frac{1}{10}$, то есть у нас будет всего 20 точек, начинать будем с точки $x_0 = 0$.

[3]:
```
def table(a, b, h):
    x = a

    func = []

    while x < b + h:
        func.append([x, f(x)])
        x += h

    new_df = pd.DataFrame(func, columns=['Точка', 'Значение функции'])
    df.loc[:, 'Точка'] = new_df['Точка']
    df.loc[:, 'Значение функции'] = new_df['Значение функции']
```

[4]:
```
df = pd.DataFrame(columns = ['Точка', 'Значение функции'])

a, b, h, epsilon = 0, 2, 1/10, 10**(-4)

table(a, b, h)

df
```

[4]:

	Точка	Значение функции
0	0.0000000	0.0000000
1	0.1000000	0.0741937
2	0.2000000	0.1576915
3	0.3000000	0.2498727
4	0.4000000	0.3501875
5	0.5000000	0.4581454
6	0.6000000	0.5733069
7	0.7000000	0.6952762
8	0.8000000	0.8236955
9	0.9000000	0.9582397
10	1.0000000	1.0986123
11	1.1000000	1.2445423
12	1.2000000	1.3957810
13	1.3000000	1.5520992
14	1.4000000	1.7132856
15	1.5000000	1.8791445
16	1.6000000	2.0494942
17	1.7000000	2.2241658
18	1.8000000	2.4030019
19	1.9000000	2.5858555
20	2.0000000	2.7725887

Таким образом, получили значения функции $f(x)$ в 20 точках. Перейдем к интерполированию в начале таблицы, для остатка интерполирования имеем следующую оценку:

$$|r_k(x)| = \left| h^{k+1} \frac{t(t-1)\dots(t-k)}{(k+1)!} f^{(k+1)}(\xi) \right| \le \left| h^{k+1} \frac{t(t-1)\dots(t-k)}{(k+1)!} \right| \cdot \max_{x\in[a,b]} |f^{(k+1)}(x)|, \quad \xi \in [x_0, x_0+kh],$$

где

$$t = \frac{x - x_0}{h},$$

которое в нашем случае известно, поскольку точки интерполирования мы берем как середины каждого отрезка $[x_i, x_{i+1}]$, $i = \overline{0, n-1}$, а именно

$$t = \frac{1}{2}$$

Из неравенства выше ограничение для погрешности остатка интерполирования задается следующей формулой

$$\left| h^{k+1} \frac{t(t-1)\dots(t-k)}{(k+1)!} \right| \cdot \max_{x\in[a,b]} |f^{(k+1)}(x)| < \varepsilon.$$

Подставляя известные нам значения, имеем

$$\left| \left(\frac{1}{10}\right)^{k+1} \cdot \frac{\frac{1}{2}(\frac{1}{2}-1)\dots(\frac{1}{2}-k)}{(k+1)!} \right| \cdot \max_{x\in[0,1]} |(x\cos x)^{(k+1)}| < 10^{-5}.$$

Из этой формулы мы и можем оценивать значение k — степени интерполирующего многочлена, которую нам и надо взять. Причем получить значение k можно лишь подбором в данном случае.

Возьмем $k = 2$. Тогда необходимо вычислить 3-ую производную от интерполируемой функции:

$$f'(x) = \ln(x+2) + \frac{x}{x+2},$$

$$f^{(2)}(x) = \frac{x+4}{x^2+4\,x+4},$$

$$f^{(3)}(x) = -\frac{x+6}{x^3+6\,x^2+12\,x+8}.$$

Оценим наибольшее значение 3-ой производной на всем отрезке $[0, 2]$. Найдем ее производную

$$f^{(4)}(x) = \frac{2\,x+16}{x^4+8\,x^3+24\,x^2+32\,x+16} = 0,$$

найдем корни уравнения $x = -8$. Подставим в $f^{(3)}(x)$:

$$|f^{(3)}(-8)| = \frac{1}{108},$$

$$|f^{(3)}(0)| = \frac{3}{4},$$

$$|f^{(3)}(2)| = \frac{1}{8}.$$

Таким образом

$$\max_{x\in[0,2]} |f^{(3)}(x)| = \frac{3}{4}.$$

Вычислим значение второго множителя:

$$\left|\left(\frac{1}{10}\right)^3 \cdot \frac{\frac{1}{2}(\frac{1}{2}-1)(\frac{1}{2}-2)}{3!}\right| = 6.25 \cdot 10^{-5}.$$

Таким образом,

$$|r_2(x)| \le 6.25 \cdot 10^{-5} \cdot \frac{3}{4} = 0.46875 \cdot 10^{-4} < 10^{-4},$$

что означает достижение необходимой погрешности.

Теперь же составим компьютерный алгоритм вычисления оценки остатка по заданной формуле и равним получившиеся результаты. Для этого определим методы, соответствующие 3-ей производной, а также метод, который будет с помощью цикла вычислять значение оценки остатка интерполирования.

[5]:
```
def third_derivative(x):
    return -(x + 6) / (x**3 + 6 * x**2 + 12 * x + 8)

def fourth_derivative(x):
    return (2 * x + 16) / (x**4 + 8 * x**3 + 24 * x**2 + 32 * x + 16)
```

[6]:
```
def remainder_estimation(h, k, t, derivative, a, b, start=True):
    x = np.linspace(a, b, num=100000)
    result = np.max(abs(derivative(x)))
```

```
    if start:
        for i in range(k + 1):
            result *= abs(h * ((t - i) / (i + 1)))
    else:
        for i in range(k + 1):
            result *= abs(h * ((t + i) / (i + 1)))

    return result
```

[7]:
```
t = 1/2
k = 2

remainder_estimation(h, k, t, third_derivative, a, b)
```

[7]:
```
4.6875000000000015e-05
```

[8]:
```
remainder_estimation(h, k, t, third_derivative, a, b) < epsilon
```

[8]:
```
True
```

То есть компьютерно вычисленный результат действительно совпал с результатом, который мы получили аналитически. И действительно, степень многочлена можно выбрать $k = 2$.

Рассмотрим аппарат конечных разностей.

• ***Конечная разность нулевого порядка*** *совпадает со значением функции*

$$f(x_i) = f_i.$$

•***Конечная разность первого порядка*** *определяется равенствами*

$$\Delta f_i = f_{i+1} - f_i.$$

•***Конечная разность второго порядка*** *определяется равенствами*

$$\Delta^2 f_i = \Delta f_{i+1} - \Delta f_i.$$

•***Конечная разность k-ого порядка*** *определяется равенствами*

$$\Delta^k f_i = \Delta(\Delta^{k-1} f_i) = \Delta^{k-1} f_{i+1} - \Delta^{k-1} f_i.$$

Процесс вычисления разделенных разностей реализуем через таблицу вида

$$\begin{array}{cccccccc}
x_0 & f(x_0) & \Delta y_0 & \Delta^2 y_0 & \dots & \Delta^{n-1} y_0 & \Delta^n y_0 \\
x_1 & f(x_1) & \Delta y_1 & \Delta^2 y_1 & \dots & \Delta^{n-1} y_1 & - \\
\vdots & \vdots & \vdots & \vdots & \ddots & - & - \\
x_{n-1} & f(x_{n-1}) & \Delta y_{n-1} & - & \dots & - & - \\
x_n & f(x_n) & - & - & \dots & - & -
\end{array}$$

[9]:
```
def finite_diff_table(df, f, n, start = 0):
    n = n + 1

    nodes = df[start:start+n]

    finite_diff = np.zeros((n, n + 1))

    finite_diff[:, 0] = [f(x) for x in nodes]

    for j in range(1, n + 1):
        for i in range(n - j):
            finite_diff[i, j] = finite_diff[i + 1, j - 1] - finite_diff[i, j - 1]

    finite_diff_df = pd.DataFrame()

    for i in range(n):
        finite_diff_df.loc[:, f'Конечная разность порядка {i}'] = finite_diff[:,␣
↪i]

    finite_diff_df = finite_diff_df.set_index([pd.Index(nodes)])

    return finite_diff_df
```

Рассмотрим таблицу конечных разностей для заданного $k = 2$.

[10]:
```
finite_diff_table(df['Точка'], f, k)
```

[10]:

Точка	Конечная разность порядка 0	Конечная разность порядка 1	Конечная разность порядка 2
0.0000000	0.0000000	0.0741937	0.0093040
0.1000000	0.0741937	0.0834977	0.0000000
0.2000000	0.1576915	0.0000000	0.0000000

Для проверки полученной таблицы реализуем так же рассчет конечных разностей по формуле

$$\Delta^k f_i = \sum_{j=0}^{k} (-1)^j C_k^j f_{i+k-j}.$$

[11]:
```
def finite_diff(nodes, values, n, start=0):
    n = n + 1

    nodes = nodes[start:start+n]
```

```
    values = values[start:start+n]

    table = [[0] * n for _ in range(n)]
    table[0] = values

    for k in range(1, n):
        for i in range(n - k):
            sum_terms = 0
            for j in range(k + 1):
                term = (-1) ** j * math.comb(k, j) * table[0][i + k - j]
                sum_terms += term
            table[k][i] = sum_terms

    return table[n - 1][0]
```

[12]:
```
finite_diff(df['Точка'], df['Значение функции'], k)
```

[12]:
```
0.009304003126978572
```

Как можем видеть, конечная разность второго порядка, вычисленная по формуле, совпадает с последним значением из таблицы.

Рассмотрим оценку остатка интерполирования в конце таблицы

$$|r_k(x)| \le \left| h^{k+1} \frac{t(t+1)\dots(t+k)}{(k+1)!} \right| \cdot \max_{x\in[a,b]} |f^{(k+1)}(x)|.$$

Возьмем уже заданное заранее $k = 2$.

[13]:
```
abs(remainder_estimation(h, k, t, fourth_derivative, a, b, start=False)) <␣
↪epsilon
```

[13]:
```
False
```

Неравенство не выполняется, соответственно, степень $k = 2$ не подходит для интерполирования в конце таблицы. Займемся перебором, возьмем $k = 3$. Для этого необходимо вычислить производную 4−го порядка и найдем ее максимум.

$$f^{(4)}(x) = \frac{2\,x + 16}{x^4 + 8\,x^3 + 24\,x^2 + 32\,x + 16},$$

$$f^{(5)}(x) = -\frac{6\,x + 60}{x^5 + 10\,x^4 + 40\,x^3 + 80\,x^2 + 80\,x + 32} = 0,$$

Найдем корни этого уравнения $x = -10$. Вычислим значения 4−й производной на концах отрезка и в полученной точке.

$$|f^{(4)}(-10)| = 2 \cdot 10^{-10},$$

$$|f^{(4)}(0)| = 1,$$

$$|f^{(4)}(2)| = \frac{5}{64}.$$

Таким образом,

$$\max_{x\in[0,2]} |f^{(4)}(x)| = 1.$$

Вычислим значение третьего множителя:

$$\left| \left(\frac{1}{10}\right)^4 \cdot \frac{\frac{1}{2}(\frac{1}{2}-1)(\frac{1}{2}-2)(\frac{1}{2}-3)}{4!} \right| = 2.73438 \cdot 10^{-5}.$$

Таким образом,

$$|r_3(x)| \leq 2.73438 \cdot 10^{-5} \cdot 1 = 0.273438 \cdot 10^{-4} < 10^{-4},$$

что означает достижение необходимой погрешности.

Зададим четвертую производную программно.

[14]:
```
def fourth_derivative(x):
    return (2 * x + 16) / (x**4 + 8 * x**3 + 24 * x**2 + 32 * x + 16)
```

[15]:
```
k = 3

abs(remainder_estimation(h, k, t, fourth_derivative, a, b))
```

[15]:
```
3.906250000000001e-06
```

Таким образом, при интерполировании на конце таблицы будем использовать многочлен $k = 3$ степени.

Выберем наибольшее из полученных k, то есть в нашем случае возьмем $k = 3$.

Перейдем к построению интерполяционного многочлена Ньютона степени $k = 3$ для начала и конца таблицы, используя следующие формулы

$$P_k(x) = P_k(x + th) = y_0 + \frac{t}{1!}\Delta y_0 + \frac{t(t-1)}{2!}\Delta^2 y_0 + \ldots + \frac{t(t-1)\ldots(t-k+1)}{k!}\Delta^k y_0,$$

$$P_k(x) = P_k(x_n - th) = y_n + \frac{t}{1!}\Delta y_{n-1} + \frac{t(t+1)}{2!}\Delta^2 y_{n-2} + \ldots + \frac{t(t+1)\ldots(t+k-1)}{k!}\Delta^k y_{n-k}.$$

Подставим наше $k = 3$

$$P_3 = y_0 + \frac{1}{2}\Delta y_0 - \frac{1}{8}\Delta^2 y_0 + \frac{1}{16}\Delta^3 y_0,$$

$$P_3 = y_n + \frac{1}{2}\Delta y_{n-1} + \frac{3}{8}\Delta^2 y_{n-2} + \frac{5}{16}\Delta^3 y_{n-3}.$$

По этим формулам можно получить значения в узлах $x \in [x_0, x_1]$ и $x \in [x_n, x_{n-1}]$ соответственно. Для получения значения из других узлов необходимо “сдвигать” таблицы вычисляемых конечных разностей вперед (назад).

Посмотрим, как выглядит таблица конечных разностей при $k = 3$.

[46]:
```
k = 3

finite_diff_table(df['Точка'], f, k)
```

[46]:

Точка	Конечная разность порядка 0	Конечная разность порядка 1	\
0.0000000	0.0000000	0.0741937	
0.1000000	0.0741937	0.0834977	
0.2000000	0.1576915	0.0921813	
0.3000000	0.2498727	0.0000000	

Точка	Конечная разность порядка 2	Конечная разность порядка 3
0.0000000	0.0093040	-0.0006205
0.1000000	0.0086835	0.0000000
0.2000000	0.0000000	0.0000000
0.3000000	0.0000000	0.0000000

Таким образом, по этой формуле можно вычислить значение в узле $x = 0.05$. Это значение добавляем в таблицу "узел-значение", но мы не сможем это сделать для каждого нового узла, так как последняя таблица, которую мы построим будет иметь вид

[48]:
```
finite_diff_table(df['Точка'], f, k, start=17)
```

[48]:

Точка	Конечная разность порядка 0	Конечная разность порядка 1	\
1.7000000	2.2241658	0.1788361	
1.8000000	2.4030019	0.1828535	
1.9000000	2.5858555	0.1867333	
2.0000000	2.7725887	0.0000000	

Точка	Конечная разность порядка 2	Конечная разность порядка 3
1.7000000	0.0040174	-0.0001377
1.8000000	0.0038797	0.0000000
1.9000000	0.0000000	0.0000000
2.0000000	0.0000000	0.0000000

Что позволит нам вычислить значение в узле $x = 1.95$. То есть в итоге мы вычислим значения в $n - k$ узлах. Для вычисления остальных значений придется использовать интерполирование в конце таблицы аналогичным образом.

Реализуем компьютерный алгоритм, который будет строить таблицу "узел-значение" для интерполирования в начале таблицы.

[18]:
```
def newton_inteprolation_from_start(t, k, h, df):
    nodes = df.values
    values = [f(x) for x in nodes]

    n = nodes.shape[0] - 1
    new_nodes = np.around(np.array([nodes[i] + t * h for i in range(n)]), 5)
    P_from_start = np.zeros(n)
```

```
    for step in range(n - k):
        P_k = 0
        for i in range(k + 1):
            mult = 1
            for j in range(i):
                mult *= (t - j) / (j + 1)
            P_k += mult * finite_diff(nodes, values, i, step)
        P_from_start[step] = P_k

    return pd.DataFrame.from_dict({'x': new_nodes, 'P_from_start':␣
↪P_from_start}, orient='index')
```

[52]:
```
start_table = newton_inteprolation_from_start(t, k, h, df['Точка'])

start_table
```

[52]:

	0	1	2	3	4	5	\
x	0.0500000	0.1500000	0.2500000	0.3500000	0.4500000	0.5500000	
P_from_start	0.0358951	0.1148228	0.2027348	0.2990473	0.4032412	0.5148528	

	6	7	8	9	10	11	\
x	0.6500000	0.7500000	0.8500000	0.9500000	1.0500000	1.1500000	
P_from_start	0.6334650	0.7587018	0.8902221	1.0277158	1.1708994	1.3195135	

	12	13	14	15	16	17	\
x	1.2500000	1.3500000	1.4500000	1.5500000	1.6500000	1.7500000	
P_from_start	1.4733193	1.6320970	1.7956431	1.9637692	2.1363002	0.0000000	

	18	19
x	1.8500000	1.9500000
P_from_start	0.0000000	0.0000000

Реализуем построение таблицы “узел-значение” при интерполировании в конце таблицы.

[53]:
```
def newton_inteprolation_from_end(t, k, h, df):
    nodes = df.values
    values = [f(x) for x in nodes]

    n = nodes.shape[0] - 1
    new_nodes = np.around(np.array([nodes[i] + t * h for i in range(n)]), 5)
    P_from_end = np.zeros(n)

    for step in range(n - 1, k - 1, -1):
        P_k = 0
        for i in range(k + 1):
            mult = 1
            for j in range(i):
                mult *= (t + j) / (j + 1)
```

```
            P_k += mult * finite_diff(nodes, values, i, step - i)
        P_from_end[step] = P_k

    return pd.DataFrame.from_dict({'x': new_nodes, 'P_from_end': P_from_end},␣
↪orient='index')
```

[54]:
```
end_table = newton_inteprolation_from_end(1/2, k, h, df['Точка'])

end_table
```

[54]:
```
                  0         1         2         3         4         5   \
x          0.0500000 0.1500000 0.2500000 0.3500000 0.4500000 0.5500000
P_from_end 0.0000000 0.0000000 0.0000000 0.2990258 0.4032230 0.5148372

                  6         7         8         9         10        11  \
x          0.6500000 0.7500000 0.8500000 0.9500000 1.0500000 1.1500000
P_from_end 0.6334516 0.7586902 0.8902120 1.0277070 1.1708917 1.3195067

                  12        13        14        15        16        17  \
x          1.2500000 1.3500000 1.4500000 1.5500000 1.6500000 1.7500000
P_from_end 1.4733133 1.6320916 1.7956383 1.9637649 2.1362963 2.3130696

                  18        19
x          1.8500000 1.9500000
P_from_end 2.4939325 2.6787428
```

Для наглядности объединим эти две таблицы в одну.

[55]:
```
newton_interpolation_table = pd.concat([start_table, end_table]).
↪drop_duplicates()

newton_interpolation_table
```

[55]:
```
                    0         1         2         3         4         5   \
x             0.0500000 0.1500000 0.2500000 0.3500000 0.4500000 0.5500000
P_from_start  0.0358951 0.1148228 0.2027348 0.2990473 0.4032412 0.5148528
P_from_end    0.0000000 0.0000000 0.0000000 0.2990258 0.4032230 0.5148372

                    6         7         8         9         10        11  \
x             0.6500000 0.7500000 0.8500000 0.9500000 1.0500000 1.1500000
P_from_start  0.6334650 0.7587018 0.8902221 1.0277158 1.1708994 1.3195135
P_from_end    0.6334516 0.7586902 0.8902120 1.0277070 1.1708917 1.3195067

                    12        13        14        15        16        17  \
x             1.2500000 1.3500000 1.4500000 1.5500000 1.6500000 1.7500000
P_from_start  1.4733193 1.6320970 1.7956431 1.9637692 2.1363002 0.0000000
P_from_end    1.4733133 1.6320916 1.7956383 1.9637649 2.1362963 2.3130696
```

```
                    18        19
x            1.8500000 1.9500000
P_from_start 0.0000000 0.0000000
P_from_end   2.4939325 2.6787428
```

Как можно заметить, значения вычисленные для начала и для конца таблицы приблизительно совпадают. Для того, чтобы совместить полученную таблицу с исходной, сперва нужно “склеить” две нижние строки получившейся таблицы. Объединим две последние строки в одну, выбирая максимальное из двух значение функции. Тогда получим таблицу

[56]:
```
interpolation_values = np.around(np.array([max(newton_interpolation_table.
 ↪iloc[1, i], newton_interpolation_table.iloc[2, i]) for i in range(20)]), 5)
interpolation_table = pd.DataFrame.from_dict({'Точка' :␣
 ↪newton_interpolation_table.iloc[0, :].tolist(), 'Значение функции' :␣
 ↪interpolation_values}, orient='index')
```

[57]:
```
interpolation_table
```

[57]:
```
                          0         1         2         3         4         5   \
Точка              0.0500000 0.1500000 0.2500000 0.3500000 0.4500000 0.5500000
Значение функции 0.0359000 0.1148200 0.2027300 0.2990500 0.4032400 0.5148500

                          6         7         8         9         10        11  \
Точка              0.6500000 0.7500000 0.8500000 0.9500000 1.0500000 1.1500000
Значение функции 0.6334600 0.7587000 0.8902200 1.0277200 1.1709000 1.3195100

                          12        13        14        15        16        17  \
Точка              1.2500000 1.3500000 1.4500000 1.5500000 1.6500000 1.7500000
Значение функции 1.4733200 1.6321000 1.7956400 1.9637700 2.1363000 2.3130700

                          18        19
Точка              1.8500000 1.9500000
Значение функции 2.4939300 2.6787400
```

Последним шагом будет объединение исходной таблицы с получившейся таблицей. Для того, чтобы таблица поместилась, протранспонируем ее.

[58]:
```
nodes_value_table = pd.concat([df, interpolation_table.T], ignore_index=True).
 ↪sort_values('Точка', ignore_index=True)
nodes_value_table
```

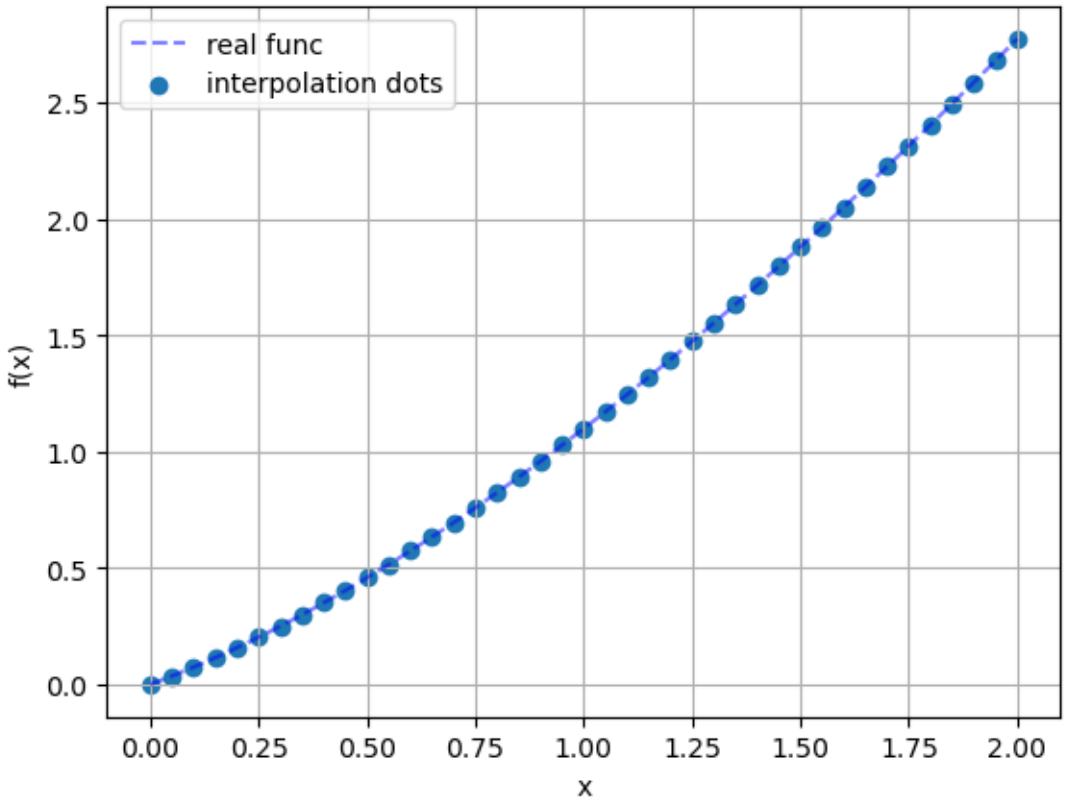
[58]:
```
       Точка Значение функции
0  0.0000000         0.0000000
1  0.0500000         0.0359000
2  0.1000000         0.0741937
3  0.1500000         0.1148200
4  0.2000000         0.1576915
5  0.2500000         0.2027300
```

```
6  0.3000000         0.2498727
7  0.3500000         0.2990500
8  0.4000000         0.3501875
9  0.4500000         0.4032400
10 0.5000000         0.4581454
11 0.5500000         0.5148500
12 0.6000000         0.5733069
13 0.6500000         0.6334600
14 0.7000000         0.6952762
15 0.7500000         0.7587000
16 0.8000000         0.8236955
17 0.8500000         0.8902200
18 0.9000000         0.9582397
19 0.9500000         1.0277200
20 1.0000000         1.0986123
21 1.0500000         1.1709000
22 1.1000000         1.2445423
23 1.1500000         1.3195100
24 1.2000000         1.3957810
25 1.2500000         1.4733200
26 1.3000000         1.5520992
27 1.3500000         1.6321000
28 1.4000000         1.7132856
29 1.4500000         1.7956400
30 1.5000000         1.8791445
31 1.5500000         1.9637700
32 1.6000000         2.0494942
33 1.6500000         2.1363000
34 1.7000000         2.2241658
35 1.7500000         2.3130700
36 1.8000000         2.4030019
37 1.8500000         2.4939300
38 1.9000000         2.5858555
39 1.9500000         2.6787400
40 2.0000000         2.7725887
```

Визуализируем полученный результат. На графике изобразим исходную функцию и точки, взятые из построенной таблицы.

[59]:
```
x = np.linspace(0, 2, 10000)

fig, ax = plt.subplots()
plt.plot(x, f(x), 'b--', alpha=0.5, label='real func')
ax.scatter(nodes_value_table['Точка'], nodes_value_table['Значение функции'],␣
 ↪label='interpolation dots', marker='o')
ax.set_xlabel('x')
ax.set_ylabel('f(x)')
```

```
plt.legend()
plt.grid()
plt.show()
```



Вывод

Мы смогли решить задачу интерполирования путём построения двух интерполяционных многочленом Ньютона, для начала и для конца таблицы. И, как можно заметить на построенном графике, приближение получилось достаточно точным для нашего отрезка $[0, 2]$.

Задача 2

На отрезке $[a, b] = [-1, 5]$ заданы функции

$$f_1(x) = \sin(5x - 3) \text{ и } f_2(x) = |2x - 3|.$$

Построить многочлены степени $n = 3, 5, 7, 10, 15, 20$, интерполирующие каждую из них по узлам

а) равномерно расположенным на указанном отрезке;

б) расположенным на указанном отрезке оптималдьным (минимизирующим погрешность) образом.

Первым делом зададим наши функции.

[29]:
```
def f_1(x):
    return np.sin(5 * x - 3)

def f_2(x):
    return abs(2 * x - 3)
```

Интерполирование многочленами по узлам равномерно расположенным на отрезке

Разобьем наш отрезок $[-1, 5]$ на $n = 3, 5, 7, 10, 15, 20$ частей.

[30]:
```
a, b = -1, 5

interpolation_nodes_df = pd.DataFrame()

for n in [20, 15, 10, 7, 5, 3]:
    interpolation_nodes_df.loc[:, f'Порядок {n}'] = pd.Series(np.linspace(a, b,␣
 ↪n))

interpolation_nodes_df = interpolation_nodes_df.fillna(' ')

interpolation_nodes_df
```

[30]:

	Порядок 20	Порядок 15	Порядок 10	Порядок 7	Порядок 5	Порядок 3
0	-1.0000000	-1.0000000	-1.0000000	-1.0000000	-1.0000000	-1.0000000
1	-0.6842105	-0.5714286	-0.3333333	0.0000000	0.5000000	2.0000000
2	-0.3684211	-0.1428571	0.3333333	1.0000000	2.0000000	5.0000000
3	-0.0526316	0.2857143	1.0000000	2.0000000	3.5000000	
4	0.2631579	0.7142857	1.6666667	3.0000000	5.0000000	
5	0.5789474	1.1428571	2.3333333	4.0000000		
6	0.8947368	1.5714286	3.0000000	5.0000000		
7	1.2105263	2.0000000	3.6666667			
8	1.5263158	2.4285714	4.3333333			
9	1.8421053	2.8571429	5.0000000			
10	2.1578947	3.2857143				
11	2.4736842	3.7142857				
12	2.7894737	4.1428571				
13	3.1052632	4.5714286				
14	3.4210526	5.0000000				
15	3.7368421					
16	4.0526316					
17	4.3684211					
18	4.6842105					
19	5.0000000					

Интерполяционный многочлен Лагранжа

Займемся построением интерполяционного многочлена Лагранжа, для этого введем в рассмотрение следующие формулы:

$$c_i = \frac{1}{(x_i - x_0)\dots(x_i - x_{i-1})(x_i - x_{i+1})\dots(x_i - x_n)}.$$

[31]:
```
def c(x_i, nodes):
    tmp = 1

    for node in nodes:
        if node == x_i:
            continue

        tmp *= (x_i - node)

    return 1 / tmp
```

Рассмотрим полином $(n+1)$ степени:

$$\omega_{n+1}(x) = (x - x_0)\dots(x - x_n).$$

[32]:
```
def w(x, nodes):
    res = 1

    for node in nodes:
        res *= (x - node)

    return res
```

Тогда

$$c_i = \frac{1}{w'_{n+1}(x_i)}.$$

Отсюда мы можем записать формулу

$$P_n(x) = \sum_{i=0}^{n} l_i(x) f(x_i) = \sum_{i=0}^{n} \frac{w_{n+1}(x)}{(x - x_i) w'_{n+1}(x_i)} f(x_i).$$

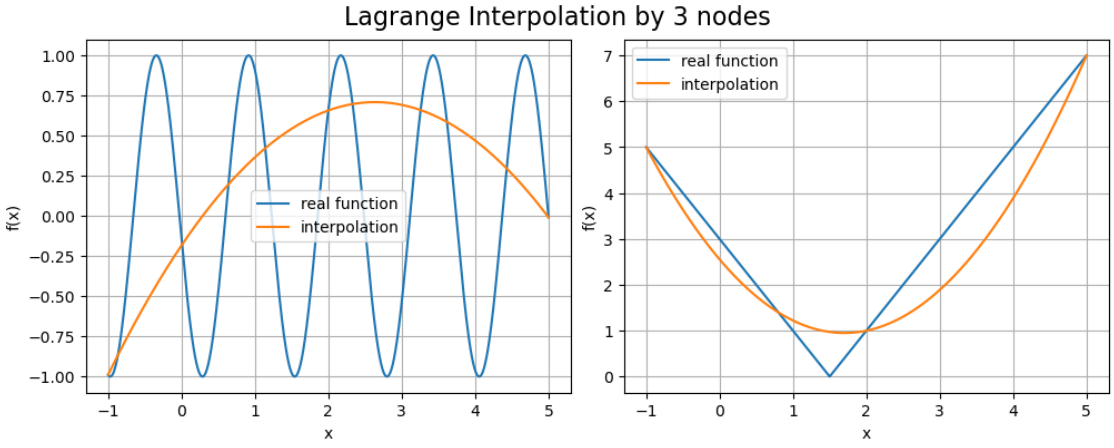
Эта формула называется **формулой Лагранжа для интерполяционного многочлена** P_n.

Зададим компьютерную функцию, которая будут вычислять значение этого многочлена при заданном x.

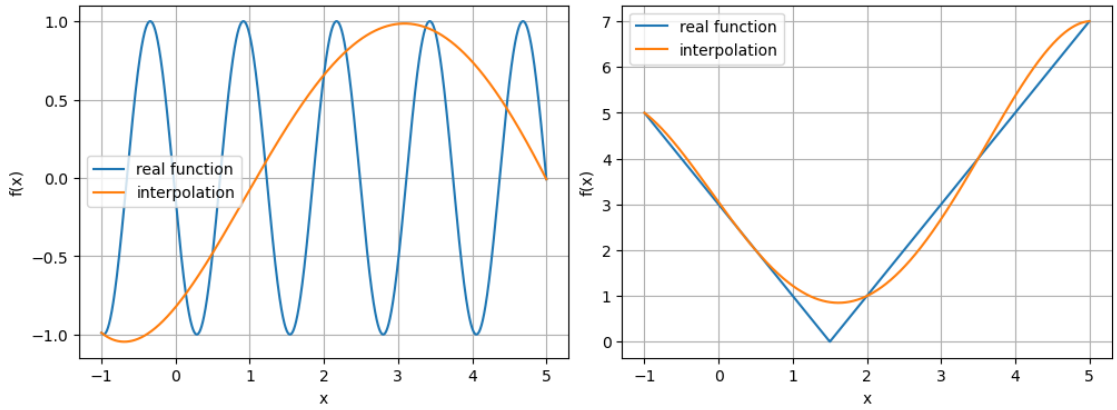
[33]:
```
def polynome_lagrange(x, nodes, f):
    return np.sum(w(x, nodes) * f(node) * c(node, nodes) / (x - node) for node␣
 ↪in nodes)
```

Построим графики для каждой функции $f_1(x), f_2(x)$ для каждого из n узлов.

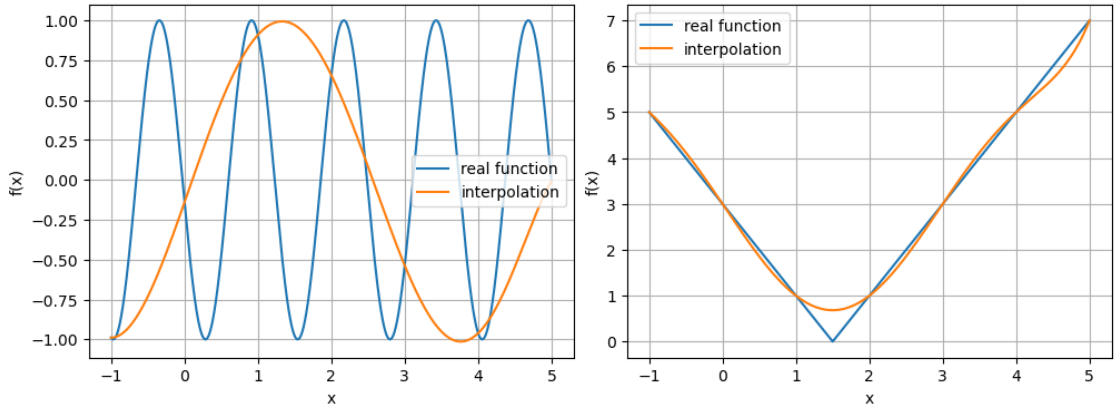
[34]:
```
x = np.linspace(a, b, 10000)

for n in [3, 5, 7, 10, 15, 20]:
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 4), layout='constrained',␣
 ↪sharey=False)

    interpolation_nodes = np.linspace(a, b, n)
    plot_dots = np.setdiff1d(x, interpolation_nodes)

    # построение графика функции f_1
    ax1.plot(x, f_1(x), label='real function')
    ax1.plot(plot_dots, polynome_lagrange(x=plot_dots,␣
 ↪nodes=interpolation_nodes, f=f_1), label='interpolation')
    ax1.set_xlabel('x')
    ax1.set_ylabel('f(x)')
    ax1.legend()
    ax1.grid()

    # построение графика функции f_2
    ax2.plot(x, f_2(x), label='real function')
    ax2.plot(plot_dots, polynome_lagrange(x=plot_dots,␣
 ↪nodes=interpolation_nodes, f=f_2), label='interpolation')
    ax2.set_xlabel('x')
    ax2.set_ylabel('f(x)')
    ax2.legend()
    ax2.grid()

    fig.suptitle(f'Lagrange Interpolation by {n} nodes', fontsize=16)
```



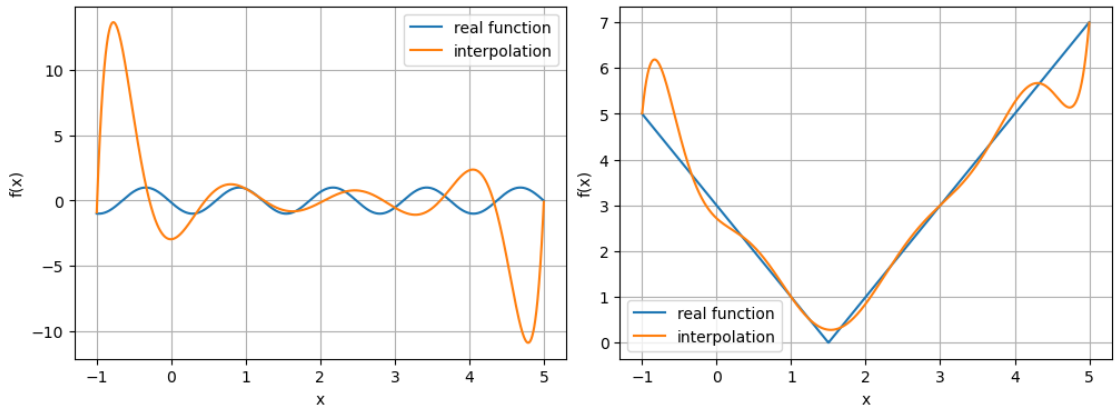
Lagrange Interpolation by 5 nodes

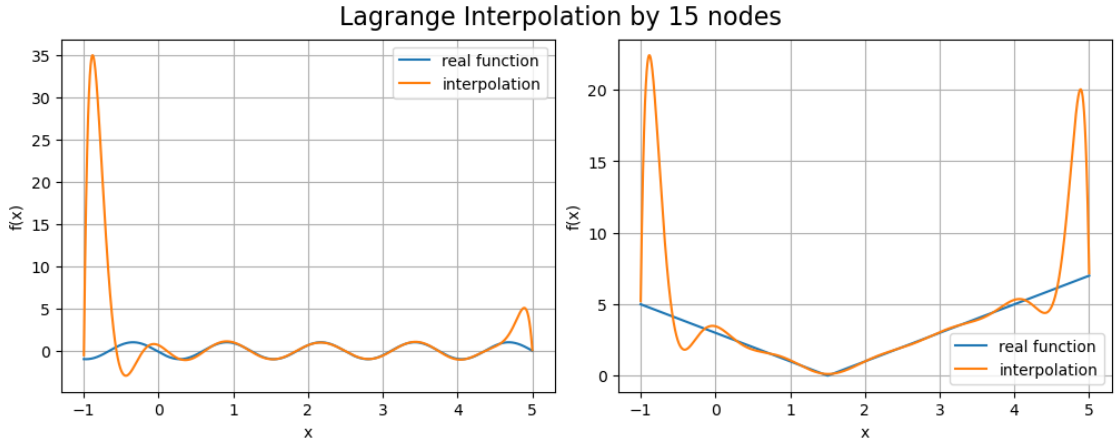


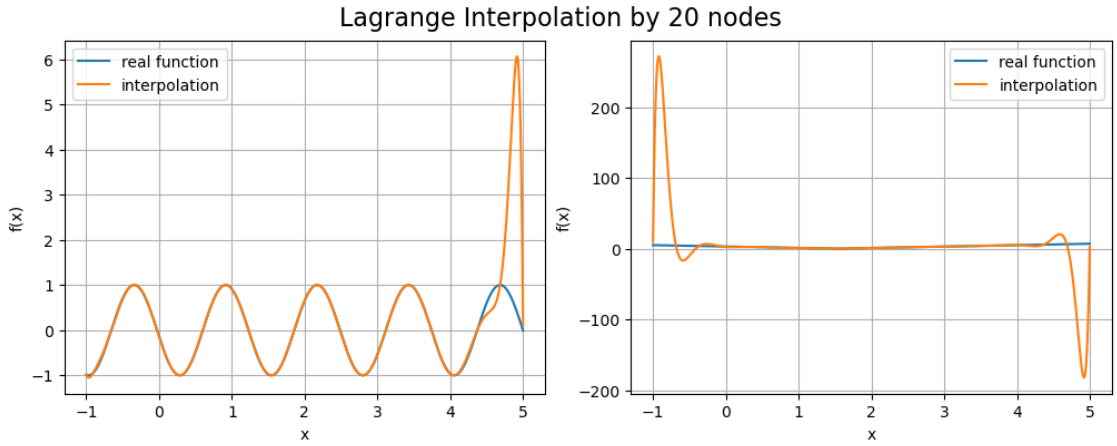
Lagrange Interpolation by 7 nodes



Lagrange Interpolation by 10 nodes







Видим, что интерполяционный многочлен Лагранжа начинает принимать очертания наших функций, начиная с $n = 15$ узлов, однако у нас наблюдается резкие отклонения.

Интерполяционный многочлен Ньютона

Представим интерполяционный многочлен в виде

$$P_n(x) = f(x_0)+(x-x_0)\cdot f(x_0,x_1)+(x-x_0)(x-x_1)\cdot f(x_0,x_1,x_2)+\ldots\ldots+(x-x_0)\ldots(x-x_{n-1})\cdot f(x_0,\ldots,x_n).$$

Такой вид называется интерполяционным многочленом Ньютона. Для этого нам понадобится аппарат разделенных разностей.

• ***Разделенная разность нулевого порядка для функции*** $f(x)$ *совпадают со значениями функции* $f(x_i)$ *в узлах интерполирования.*
•***Разделенная разность первого порядка*** *есть*

$$f(x_i,x_j) = \frac{f(x_j)-f(x_i)}{x_j-x_i}.$$

•***Разделенная разность второго порядка***

$$f(x_i, x_j, x_k) = \frac{f(x_j, x_k) - f(x_i, x_j)}{x_k - x_i}.$$

•***Разделенная разность*** $(k+1)$***-ого порядка***

$$f(x_0, \ldots, x_{k+1}) = \frac{f(x_1, \ldots, x_{k+1}) - f(x_0, \ldots, x_k)}{x_{k+1} - x_0}.$$

Можно показать, что справедлива формула, связывающая разделенную разность k-ого порядка со значениями функции в указанных узлах

$$f(x_0, x_1, \ldots, x_k) = \sum_{j=0}^{k} \frac{f(x_j)}{\omega'_{k+1}(x_j)},$$

где ω_{k+1} определяется из формулы.

[35]:
```
def divided_diff_sum(nodes, f):
    return np.sum(f(node) * c(node, nodes) for node in nodes)
```

В практических вычислениях подсчет разделенных разностей реализуется через таблицу

x_0	$f(x_0)$	$f(x_0, x_1)$	$f(x_0, x_1, x_2)$	⋮	$f(x_0, \ldots, x_n)$
x_1	$f(x_1)$	$f(x_1, x_2)$	⋮		
x_2	$f(x_2)$	⋮	$f(x_{n-2}, x_{n-1}, x_n)$		
⋮	⋮	$f(x_{n-1}, x_n)$			
x_{n-1}	$f(x_{n-1})$				
x_n	$f(x_n)$				

[36]:
```
def divided_diff(nodes, f):
    n = len(nodes)
    divided_diff = np.zeros((n, n + 1))

    divided_diff[:, 0] = [f(x) for x in nodes]

    for j in range(1, n + 1):
        for i in range(n - j):
            divided_diff[i, j] = (divided_diff[i + 1, j - 1] - divided_diff[i, j␣
↪- 1]) / (nodes[i + j] - nodes[i])

    divided_diff_df = pd.DataFrame()

    for i in range(n):
        divided_diff_df.loc[:, f'Разделенная разность порядка {i + 1}'] =␣
↪divided_diff[:, i]
```

```
    divided_diff_df = divided_diff_df.set_index([pd.Index(nodes)])

    return divided_diff_df
```

Сравним значения, полученные в таблице с суммирующей функцией для, например, 3 узлов:

[37]:
```
nodes = [node for node in interpolation_nodes_df['Порядок 3'].tolist() if␣
 ↪isinstance(node, float)]

divided_diff(nodes, f_1)
```

[37]:
```
            Разделенная разность порядка 1  Разделенная разность порядка 2  \
-1.0000000                       -0.9893582                        0.5487816
2.0000000                        0.6569866                       -0.2219460
5.0000000                       -0.0088513                        0.0000000

            Разделенная разность порядка 3
-1.0000000                       -0.1284546
2.0000000                        0.0000000
5.0000000                        0.0000000
```

[38]:
```
divided_diff_sum(nodes, f_1)
```

[38]:
```
-0.1284545974084091
```

Как можем видеть, разделенная разность третьего порядка, вычисленная по формуле, совпадает с последним значением из таблицы.

Реализуем функцию для вычисления интерполяционного многочлена Ньютона.

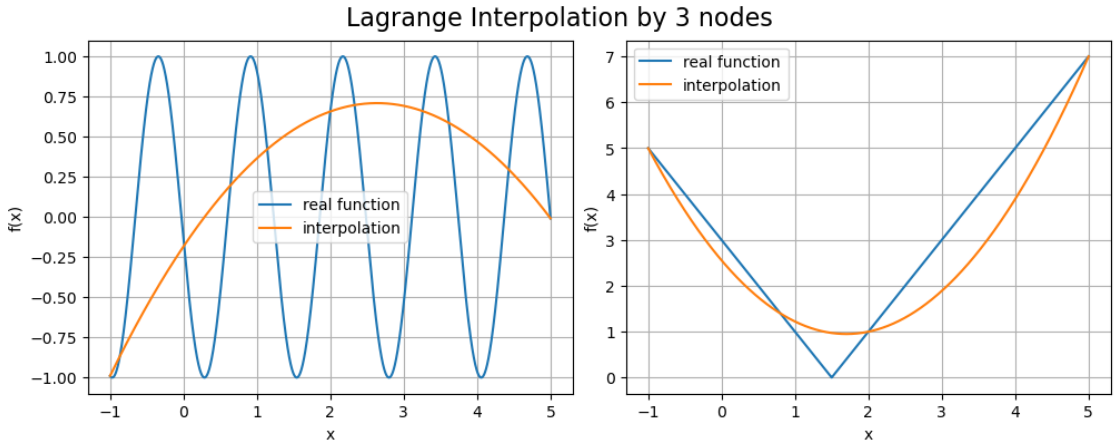
$P_n(x) = f(x_0)+(x-x_0)\cdot f(x_0,x_1)+(x-x_0)(x-x_1)\cdot f(x_0,x_1,x_2)+\ldots\ldots+(x-x_0)\ldots(x-x_{n-1})\cdot f(x_0,\ldots,x_n).$

[39]:
```
def polynome_newton(x, nodes, div_diff, f):
    p = f(nodes[0])
    tmp = x - nodes[0]

    for i in range(1, n):
        p += (tmp * div_diff[i])
        tmp *= (x - nodes[i])

    return p
```

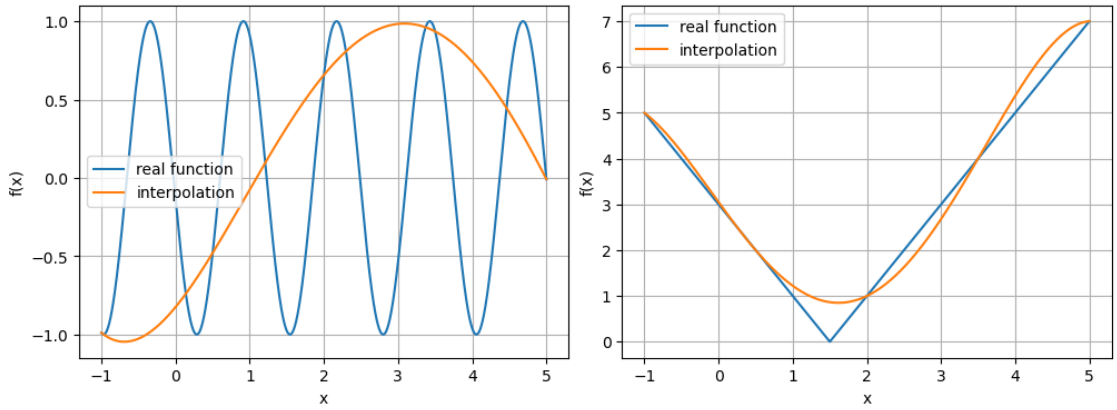
Построим графики для каждой функции $f_1(x), f_2(x)$ для каждого из n узлов.

[40]:
```
x = np.linspace(a, b, 10000)

for n in [3, 5, 7, 10, 15, 20]:
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 4), layout='constrained',␣
 ↪sharey=False)
```

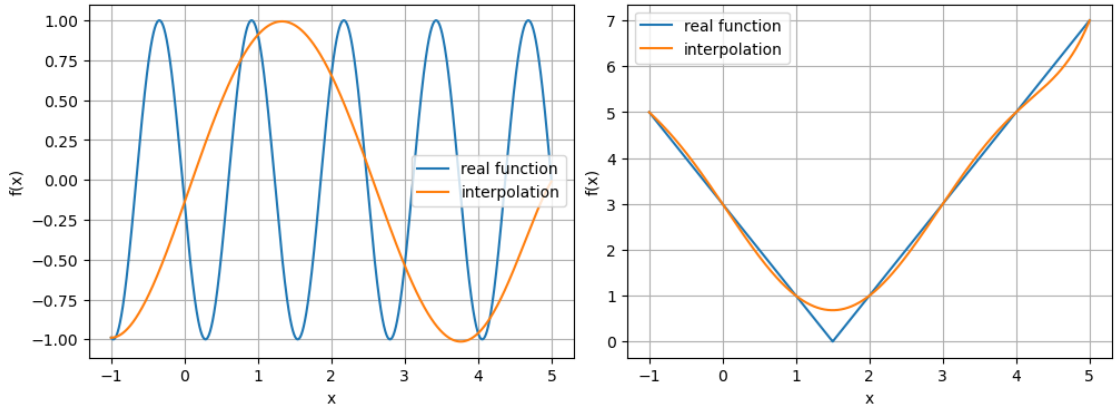
```
    interpolation_nodes = np.linspace(a, b, n)
    plot_dots = np.setdiff1d(x, interpolation_nodes)

    # построение графика функции f_1
    ax1.plot(x, f_1(x), label='real function')
    ax1.plot(plot_dots, polynome_newton(x=plot_dots, nodes=interpolation_nodes,␣
↪div_diff=divided_diff(interpolation_nodes, f_1).to_numpy()[0, :], f=f_1),␣
↪label='interpolation')
    ax1.set_xlabel('x')
    ax1.set_ylabel('f(x)')
    ax1.legend()
    ax1.grid()

    # построение графика функции f_2
    ax2.plot(x, f_2(x), label='real function')
    ax2.plot(plot_dots, polynome_newton(x=plot_dots,␣
↪div_diff=divided_diff(interpolation_nodes, f_2).to_numpy()[0, :],␣
↪nodes=interpolation_nodes, f=f_2), label='interpolation')
    ax2.set_xlabel('x')
    ax2.set_ylabel('f(x)')
    ax2.legend()
    ax2.grid()

    fig.suptitle(f'Lagrange Interpolation by {n} nodes', fontsize=16)
```



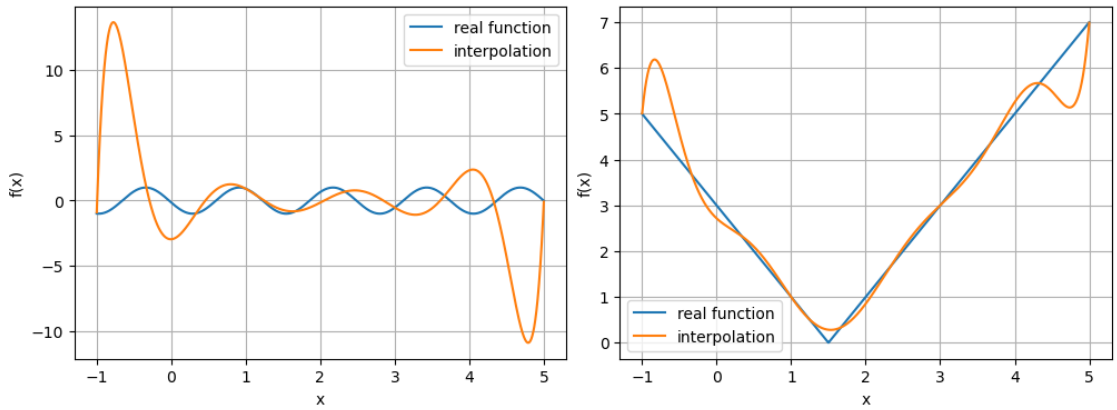
Lagrange Interpolation by 5 nodes

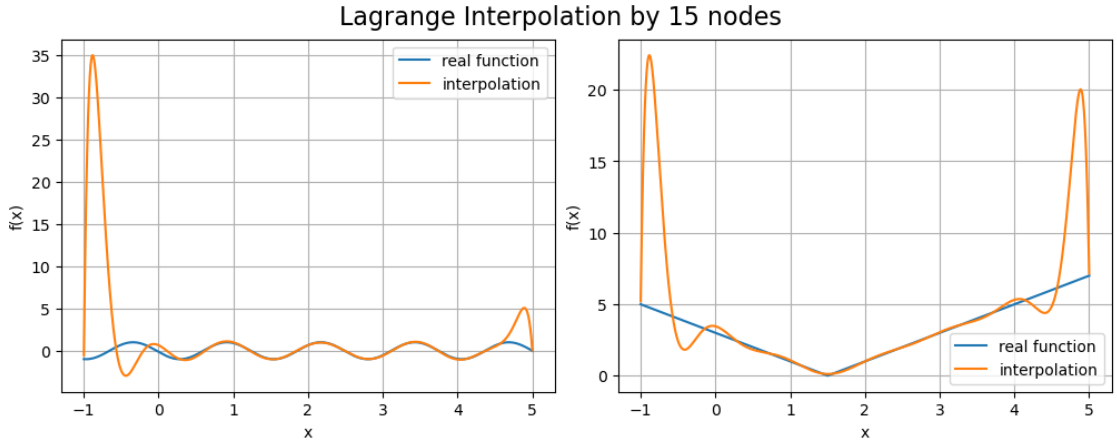


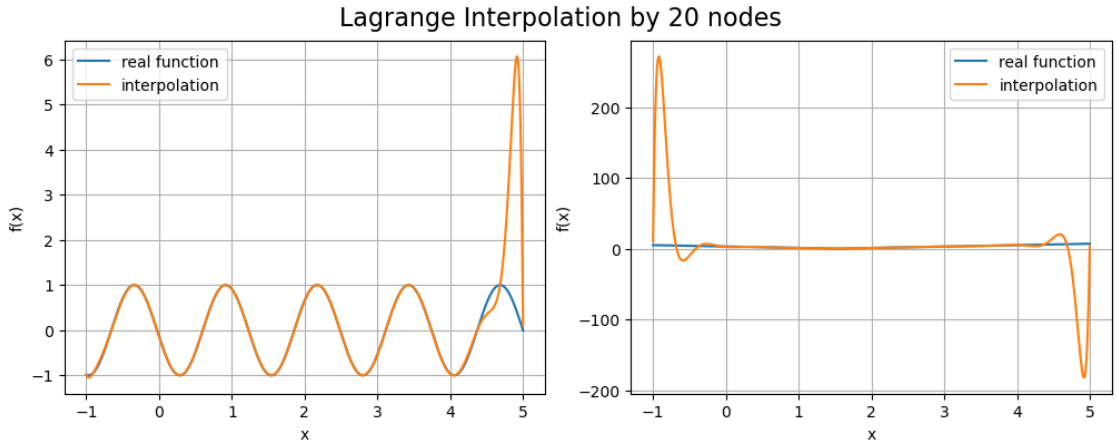
Lagrange Interpolation by 7 nodes



Lagrange Interpolation by 10 nodes







В соответствии с корректностью поставленной задачи, а именно единственностью решения, мы можем увидеть, что оба интерполяционных многочлена дали одинаковый результат, причем вне зависимости от степени полинома. Оба они достигли наилучшего результата при $n = 20$ узлах. Однако в процессе построения функции всегда присутствовали отрезке резкого отклонения.

Интерполирование многочленами по узлам расположенным на указанном отрезке оптимальным образом

Для минимизации остатка интерполирования выберем узлы по формуле

$$x_k = \frac{a+b}{2} + \frac{b-a}{2} \cos \frac{(2k+1)\pi}{2(n+1)}, \ k = \overline{0,n}.$$

Если выбрать узлами интерполирования $x_0, \ldots, x_n$ таким образом, то величина отклонения $\omega_{n+1}(x)$ от нуля окажется минимальной.

Для упорядочивания узлов необходима перенумерация $\widetilde{x}_k = x_{n-k}, \ k = \overline{0,n}$.

Составим таблицу нового разбиения на узлы.

[41]:
```
a, b = -1, 5

interpolation_new_nodes_df = pd.DataFrame()

for n in [20, 15, 10, 7, 5, 3]:
    interpolation_new_nodes_df.loc[: n - 1, f'Порядок {n}'] = np.fromiter(((a +␣
 ↪b) / 2 + np.sum((b - a) / 2 * np.cos((2*k + 1) * np.pi / (2 * (n + 1)))) for k␣
 ↪in range(n)), dtype=float)[::-1]

interpolation_new_nodes_df = interpolation_new_nodes_df.fillna(' ')

interpolation_new_nodes_df
```

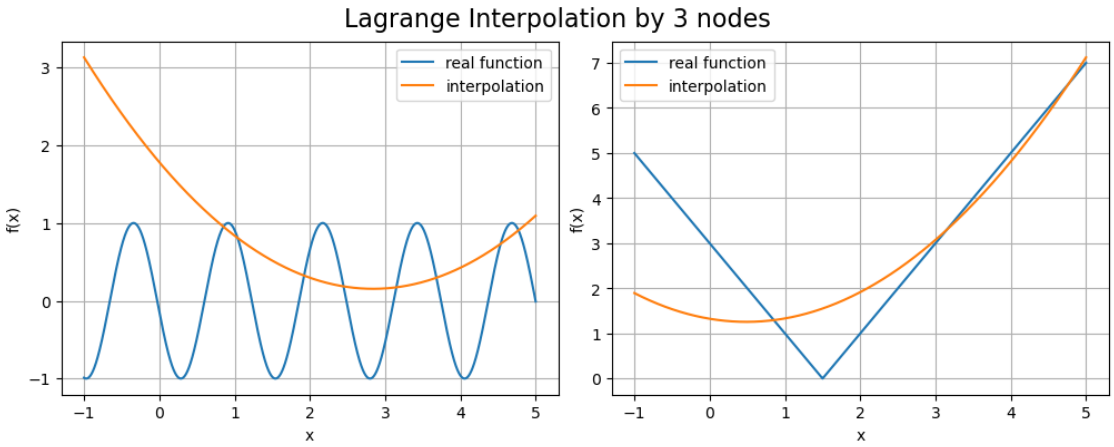
[41]:

	Порядок 20	Порядок 15	Порядок 10	Порядок 7	Порядок 5	Порядок 3
0	-0.9247837	-0.8708210	-0.7288960	-0.4944088	-0.1213203	0.8519497
1	-0.7926212	-0.6457638	-0.2672487	0.3332893	1.2235429	3.1480503
2	-0.5980762	-0.3190314	0.3780775	1.4147290	2.7764571	4.7716386
3	-0.3454944	0.0968201	1.1548023	2.5852710	4.1213203	
4	-0.0405182	0.5858098	2.0000000	3.6667107	4.8977775	
5	0.3100398	1.1291460	2.8451977	4.4944088		
6	0.6983488	1.7059486	3.6219225	4.9423558		
7	1.1157345	2.2940514	4.2672487			
8	1.5528732	2.8708540	4.7288960			
9	2.0000000	3.4141902	4.9694643			
10	2.4471268	3.9031799				
11	2.8842655	4.3190314				
12	3.3016512	4.6457638				
13	3.6899602	4.8708210				
14	4.0405182	4.9855542				
15	4.3454944					
16	4.5980762					
17	4.7926212					
18	4.9247837					
19	4.9916114					

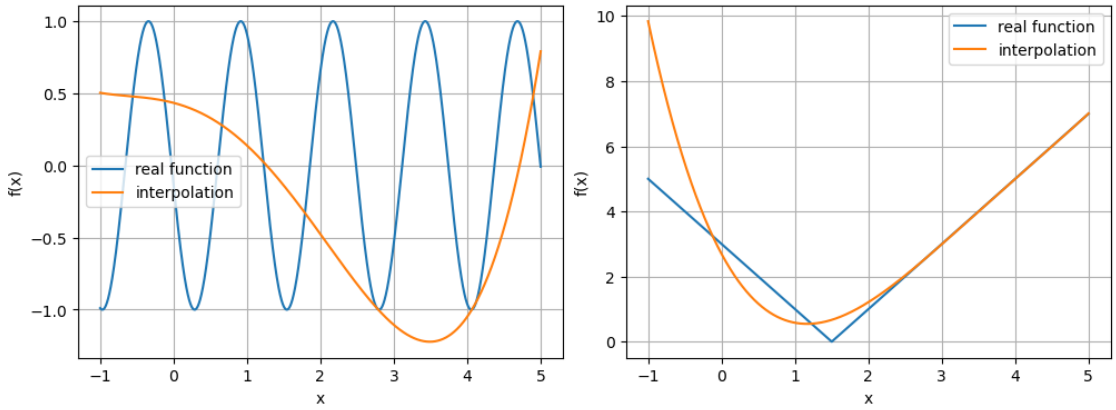
Будем действовать в том же порядке, сначала рассмотрим интерполирование многочленом Лагранжа.

[42]:
```
x = np.linspace(a, b, 10000)

for n in [3, 5, 7, 10, 15, 20]:
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 4), layout='constrained',␣
 ↪sharey=False)

    interpolation_nodes = [node for node in interpolation_new_nodes_df[f'Порядок␣
 ↪{n}'].tolist() if isinstance(node, float)]
```

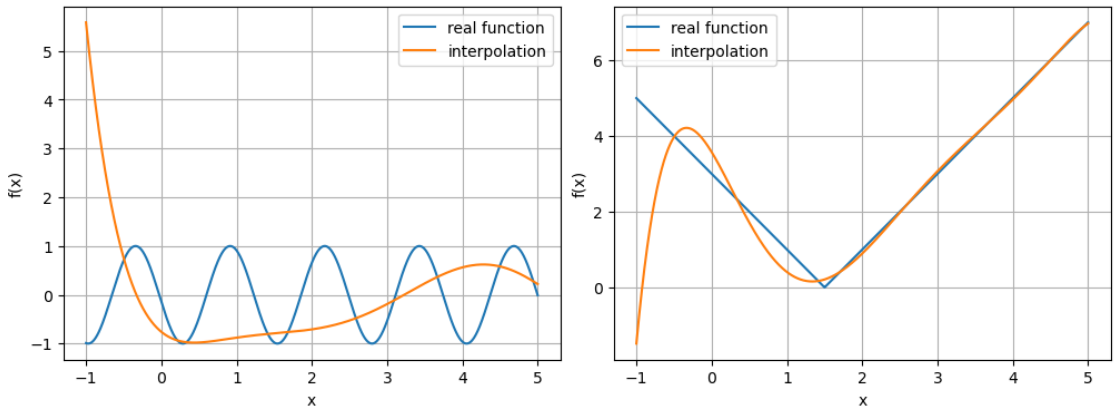
```
    plot_dots = np.setdiff1d(x, interpolation_nodes)

    # построение графика функции f_1
    ax1.plot(x, f_1(x), label='real function')
    ax1.plot(plot_dots, polynome_lagrange(x=plot_dots,␣
↪nodes=interpolation_nodes, f=f_1), label='interpolation')
    ax1.set_xlabel('x')
    ax1.set_ylabel('f(x)')
    ax1.legend()
    ax1.grid()

    # построение графика функции f_2
    ax2.plot(x, f_2(x), label='real function')
    ax2.plot(plot_dots, polynome_lagrange(x=plot_dots,␣
↪nodes=interpolation_nodes, f=f_2), label='interpolation')
    ax2.set_xlabel('x')
    ax2.set_ylabel('f(x)')
    ax2.legend()
    ax2.grid()

    fig.suptitle(f'Lagrange Interpolation by {n} nodes', fontsize=16)
```



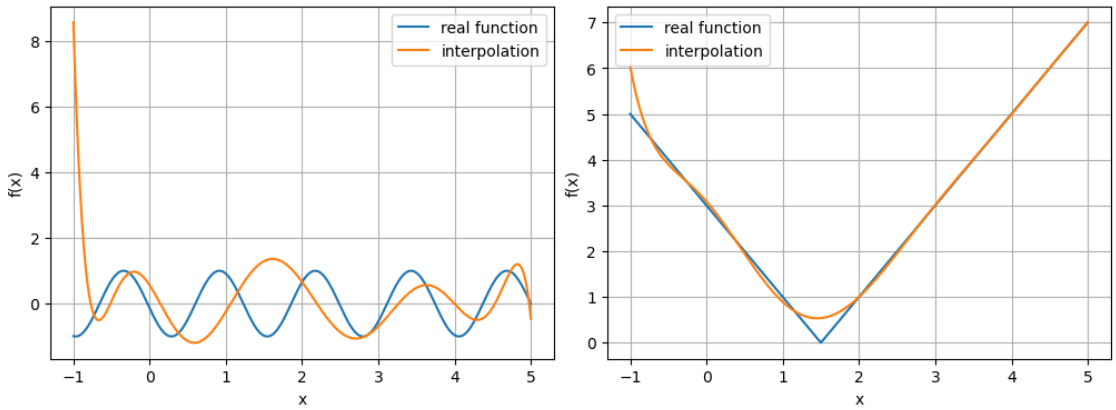
Lagrange Interpolation by 5 nodes

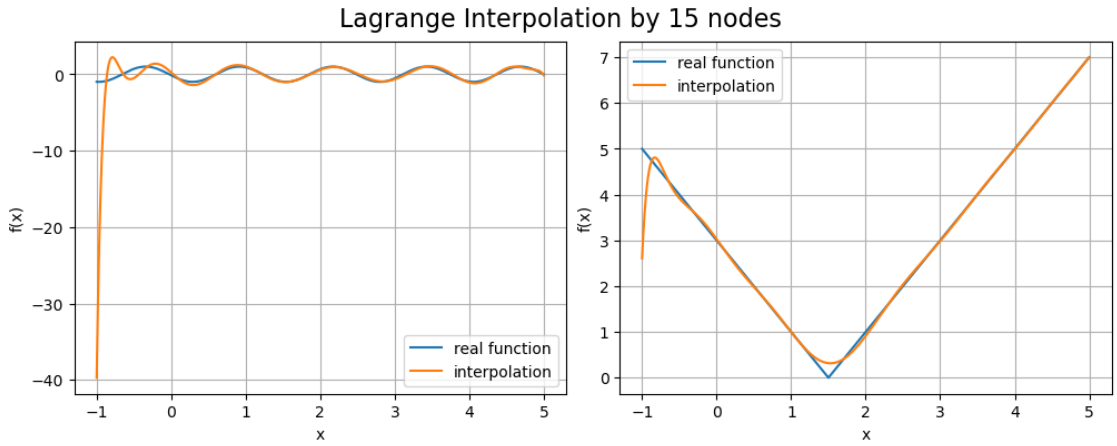


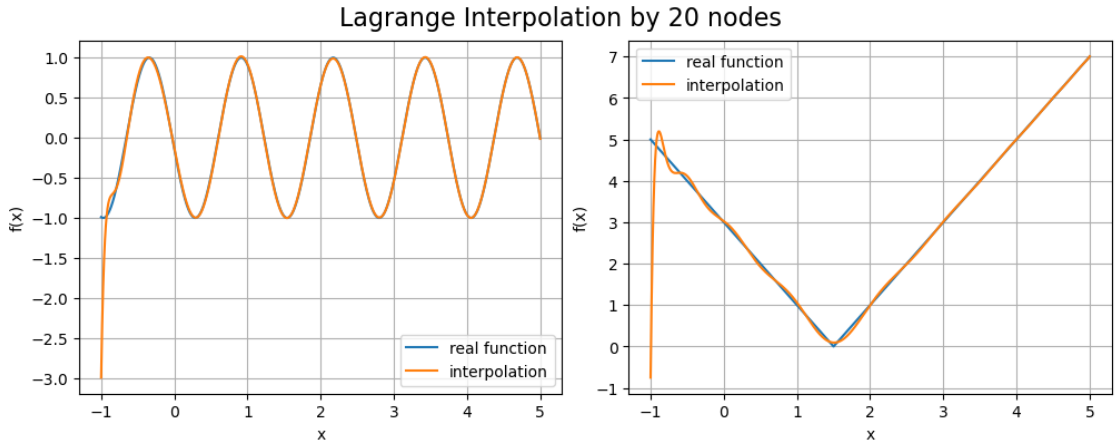
Lagrange Interpolation by 7 nodes



Lagrange Interpolation by 10 nodes



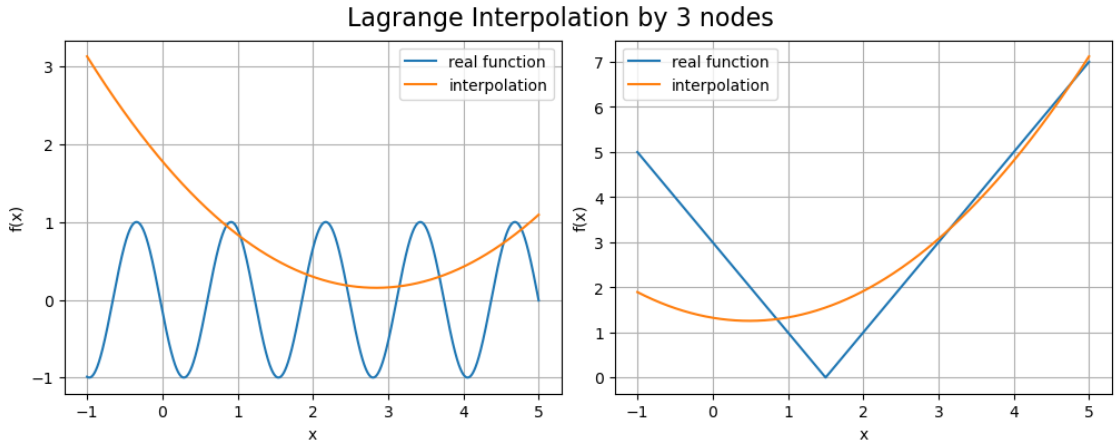




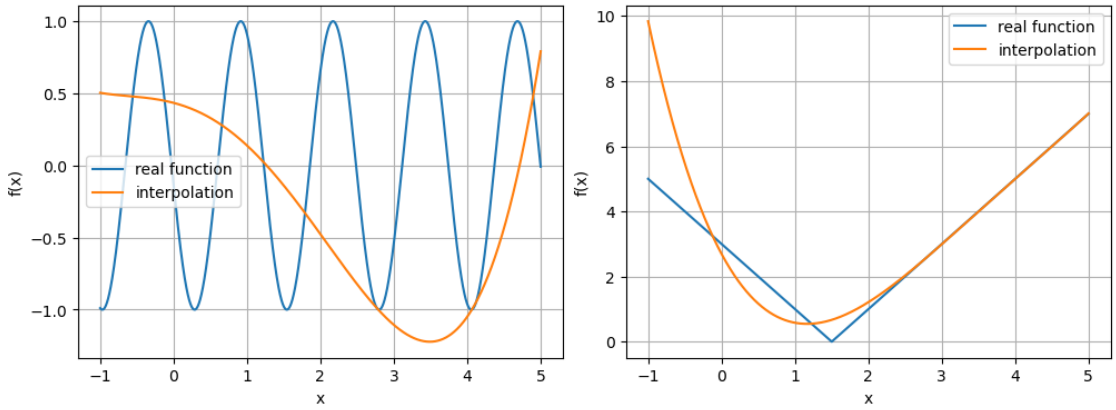
Теперь рассмотрим интерполирование многочленом Ньютона.

[43]:
```
x = np.linspace(a, b, 10000)

for n in [3, 5, 7, 10, 15, 20]:
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 4), layout='constrained',␣
 ↪sharey=False)

    interpolation_nodes = [node for node in interpolation_new_nodes_df[f'Порядок␣
 ↪{n}'].tolist() if isinstance(node, float)]
    plot_dots = np.setdiff1d(x, interpolation_nodes)

    # построение графика функции f_1
    ax1.plot(x, f_1(x), label='real function')
```

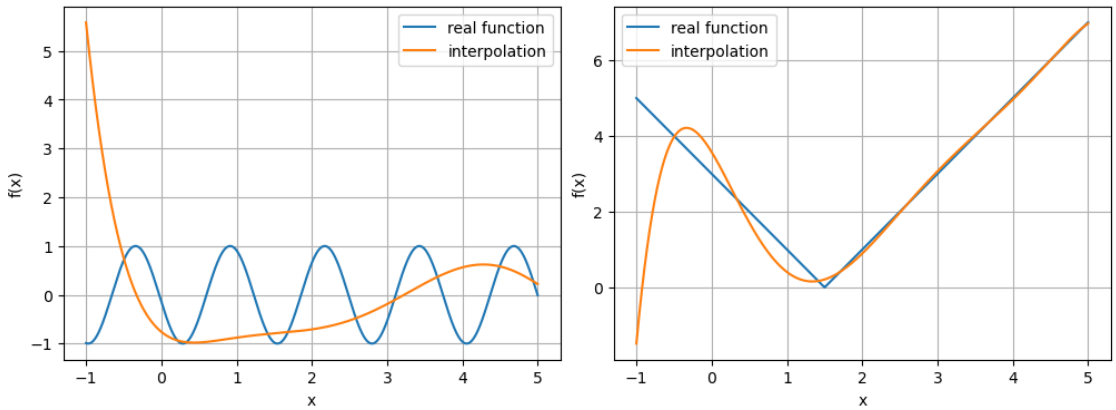
```
    ax1.plot(plot_dots, polynome_newton(x=plot_dots, nodes=interpolation_nodes,␣
↪div_diff=divided_diff(interpolation_nodes, f_1).to_numpy()[0, :], f=f_1),␣
↪label='interpolation')
    ax1.set_xlabel('x')
    ax1.set_ylabel('f(x)')
    ax1.legend()
    ax1.grid()

    # построение графика функции f_2
    ax2.plot(x, f_2(x), label='real function')
    ax2.plot(plot_dots, polynome_newton(x=plot_dots,␣
↪div_diff=divided_diff(interpolation_nodes, f_2).to_numpy()[0, :],␣
↪nodes=interpolation_nodes, f=f_2), label='interpolation')
    ax2.set_xlabel('x')
    ax2.set_ylabel('f(x)')
    ax2.legend()
    ax2.grid()

    fig.suptitle(f'Lagrange Interpolation by {n} nodes', fontsize=16)
```



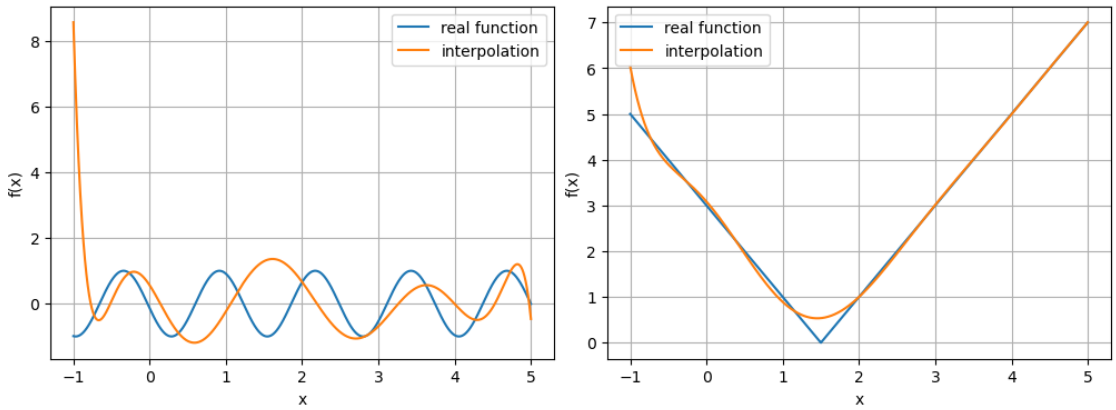
Lagrange Interpolation by 5 nodes

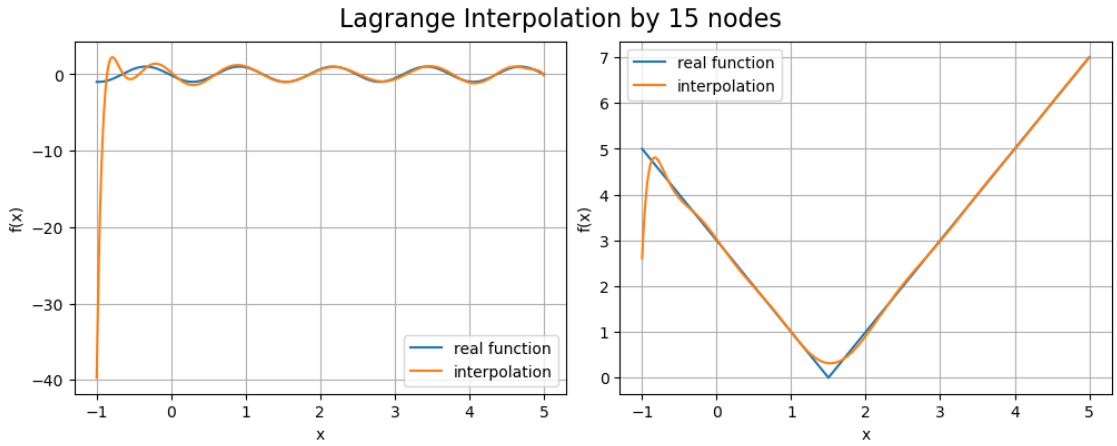


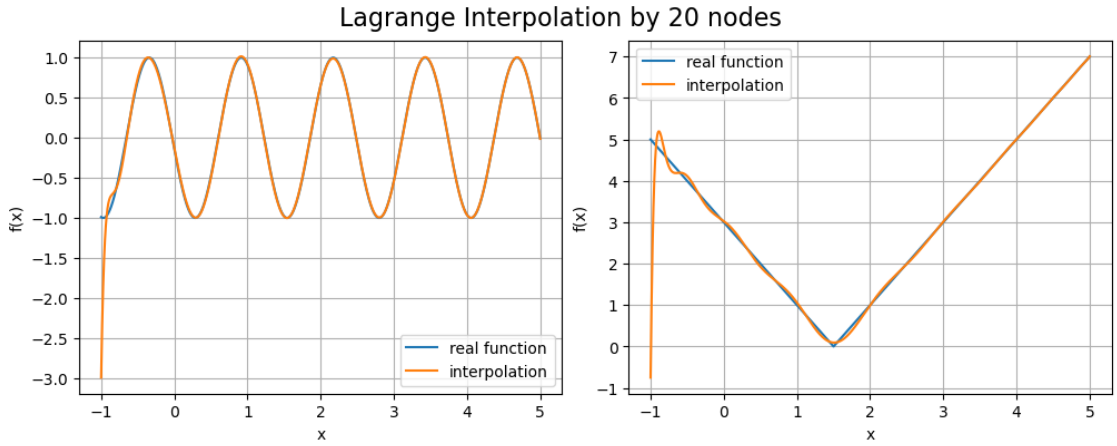
Lagrange Interpolation by 7 nodes



Lagrange Interpolation by 10 nodes







В целом, аналогично раномерному распределению узлов по отрезку, получаем то, что результат интерполирования не отличается от выбора вида полинома. Однако в этом случае результат стал немного лучше как для функции $f_1(x)$ так и для $f_2(x)$. Стоит обратить внимание на то, что для обеих функций пристуствуют отрезке резкого отклонения как для полинома Лагранжа, так и для полинома Ньютона.

Вывод

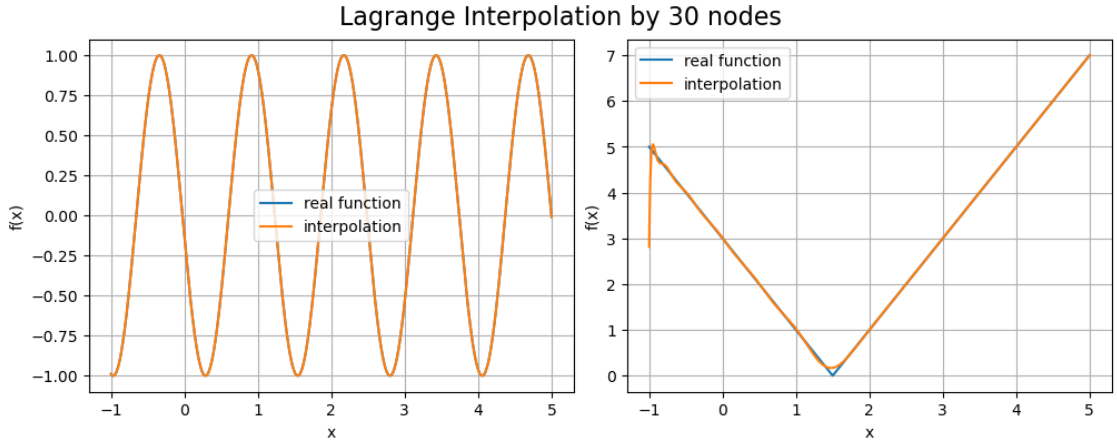
По итогам выполнения задания 2 можно сделать вывод, что для заданных функций $f_1(x) = \sin(5x - 3)$ и $f_2(x) = |2x - 3|$ лучшим образом работает интерполирования по узлам, расположенным оптимальным образом, причем для $f_1(x)$ результат точнее. Наличие отрезков, где наблюдаются сильные различия можно обусловить простым недостатком узлов, возможно, при использовании 30 узлов, результат будет лучше. Проверим это

[44]:
```
a, b = -1, 5
```

```
interpolation_node = []

n = 30

interpolation_node =  np.fromiter(((a + b) / 2 + np.sum((b - a) / 2 * np.
↪cos((2*k + 1) * np.pi / (2 * (n + 1)))) for k in range(n)), dtype=float)[::-1]

interpolation_node
```

[44]:
```
array([-0.96540497, -0.90423136, -0.8132564 , -0.69341362, -0.54593277,
       -0.37232721, -0.17437836,  0.04588255,  0.28619535,  0.54409411,
        0.81693243,  1.10191063,  1.39610444,  1.69649503,  2.        ,
        2.30350497,  2.60389556,  2.89808937,  3.18306757,  3.45590589,
        3.71380465,  3.95411745,  4.17437836,  4.37232721,  4.54593277,
        4.69341362,  4.8132564 ,  4.90423136,  4.96540497,  4.99614952])
```

[45]:
```
x = np.linspace(a, b, 10000)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 4), layout='constrained',␣
↪sharey=False)

plot_dots = np.setdiff1d(x, interpolation_node)

# построение графика функции f_1
ax1.plot(x, f_1(x), label='real function')
ax1.plot(plot_dots, polynome_lagrange(x=plot_dots, nodes=interpolation_node,␣
↪f=f_1), label='interpolation')
ax1.set_xlabel('x')
ax1.set_ylabel('f(x)')
ax1.legend()
ax1.grid()

# построение графика функции f_2
ax2.plot(x, f_2(x), label='real function')
ax2.plot(plot_dots, polynome_lagrange(x=plot_dots, nodes=interpolation_node,␣
↪f=f_2), label='interpolation')
ax2.set_xlabel('x')
ax2.set_ylabel('f(x)')
ax2.legend()
ax2.grid()

fig.suptitle(f'Lagrange Interpolation by {n} nodes', fontsize=16)
```



При использовании 30 узлов, интерполирование Лагранжа работает идеально для функции $f_1(x) = \sin(5x - 3)$, а для функции $f_2(x) = |2x - 3|$ всё еще есть недочеты.