ОДНОМЕРНЫЕ ЧИСЛЕННЫЕ МЕТОДЫ

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
import math
```

1. ГРАДИЕНТНЫЕ МЕТОДЫ

Одномерные градиентные методы представляют собой оптимизационные алгоритмы, которые позволяют нам находить стационарные точки функции *f(x)* при условии, что функция дифференцируема до необходимого порядка.

In [ ]:
```
def func(x):
  f = x**2 - np.sin(2*x)
  return f

def dfunc(x):
  f = 2*x - 2*np.cos(2*x)
  return f

def ddfunc(x):
  f = 2 + 4*np.sin(2*x)
  return f
```

1.1 МЕТОД НЬЮТОНА-РАФСОНА

Итеративная формула Ньютон-Рафсона для одномерной задачи может быть выражена математически следующим образом:

$$x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)}, \tag{1}$$

где x_{n+1} - значение следующей итерации, x_n - текущее значение, $f'(x_n)$ - первая производная функции (градиент), а также $f''(x_n)$ - вторая производная функции в точке x_n. Этот процесс повторяется итеративно до тех пор, пока не будет выполнено определенное условие остановки. Обычно остановка происходит, когда значения итераций становятся очень близкими, то есть:

$$|x_n - x_{n-1}| < \epsilon, \tag{2}$$

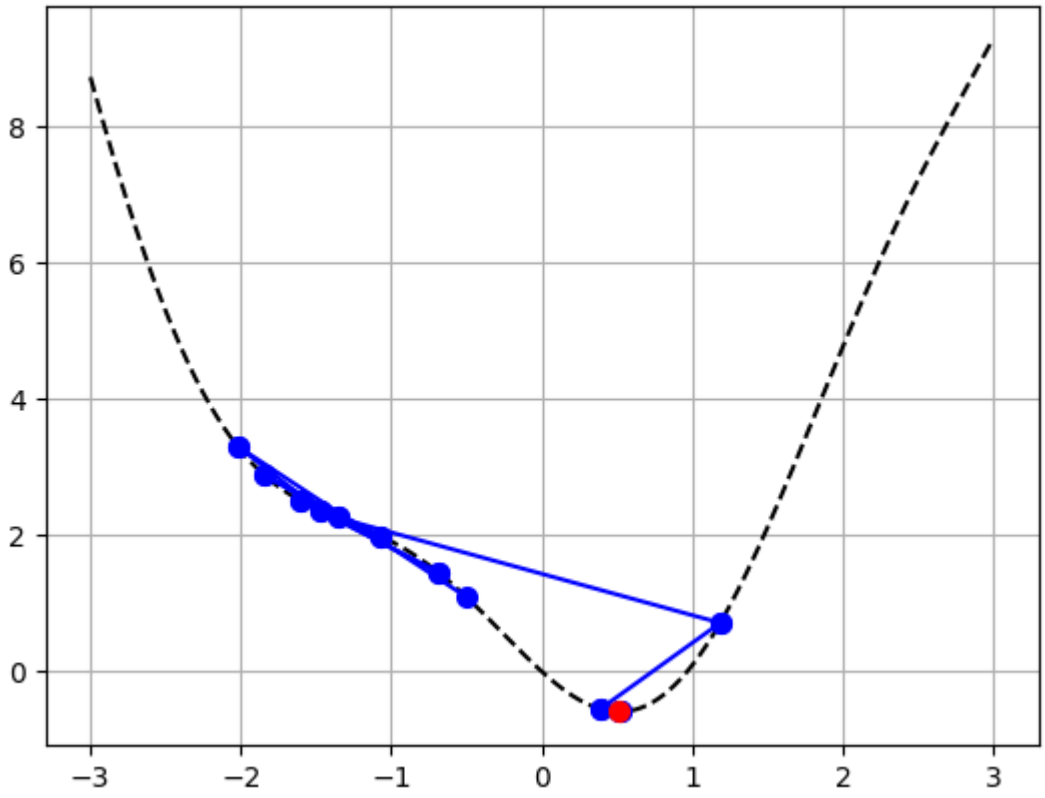
где ϵ представляет собой допуск функции. Как дополнительный критерий остановки, можно использовать так называемый *принудительный конец*. Такой способ остановки предполагает заранее определенное максимальное количество итераций, потому что, если начальное предположение выбрано плохо, алгоритм может расходиться, и первоначальный критерий никогда не будет выполнен.

In [ ]:
```
def njutn_rapsonov_metod(x0, epsilon):
    x_new = x0
    x_previous = math.inf
```

```
    x_niz = [x_new]

    iter = 0
    max_iter = 30

    while (abs(x_previous - x_new) > epsilon):
        iter += 1
        x_previous = x_new
        x_new = x_previous - dfunc(x_previous)/ddfunc(x_previous)

        x_niz.append(x_new)

        if max_iter <= iter:
          break

    xopt = x_new
    fopt = func(xopt)
    return xopt, fopt, iter, x_niz
```

In [ ]:

```
[x_opt, f_opt, iter, x_array] = njutn_rapsonov_metod(x0=-0.5, epsilon=0.1

x = np.linspace(-3, 3, 1000)
f = np.linspace(0, 0, len(x))
for i in range(0, len(x), 1):
    f[i] = func(x[i])

plt.plot(x, f, 'k--', label='f(x)')

for i in range(0, len(x_array)-1):
    plt.plot([x_array[i], x_array[i+1]], [func(x_array[i] ), func(x_array
plt.plot(x_opt, f_opt, '-or', label='max[f(x)]', markersize=5, markeredge
plt.grid(True)
plt.show()
print(f'Стационарная точка функции f(x): {x_opt} а оптимальное значение ф
```



```
Стационарная точка функции f(x): 0.5149601404161019 а оптимальное значение
функции: -0.5920739993172587. Алгоритм завершается за 11 итераций.
```

1.2 МЕТОД СЕЧЕНИЯ

Основное различие между методом секущих и методом Ньютона-Рафсона заключается в том, что в методе секущих вводится аппроксимация для второй производной:

$$f''(x_n) = \frac{f'(x_n) - f'(x_{n-1})}{x_n - x_{n-1}}, \tag{3}$$

поэтому итерационная формула метода секущих принимает следующий вид:

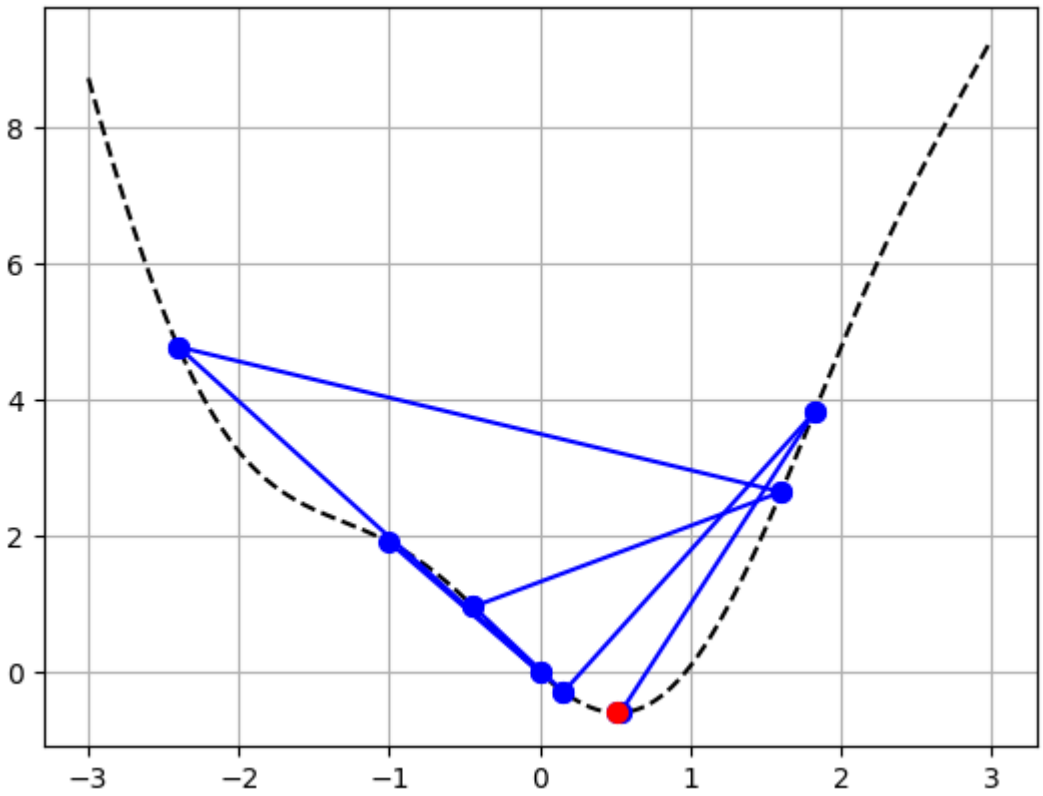
$$x_{n+1} = x_n - f'(x_n)\frac{x_n - x_{n-1}}{f'(x_n) - f'(x_{n-1})}. \tag{4}$$

Как видно из формулы, теперь не требуется, чтобы функция была дифференцируема до второго порядка, но теперь нам нужны две начальные точки. Выбором хорошего начального интервала (начальных точек) мы позволяем алгоритму сходиться к оптимуму. Этот подход может уменьшить скорость сходимости по сравнению с методом Ньютона-Рафсона.

In [ ]:
```
def metod_secice(x0, x1, epsilon):
    x_pre = x0
    x_novo = x1

    x_niz = [x_pre, x_novo]

    iter = 0

    while(abs(x_novo-x_pre) > epsilon):
        iter += 1
        x_ppre = x_pre
        x_pre = x_novo
        x_novo = x_pre-dfunc(x_pre)*(x_pre-x_ppre)/(dfunc(x_pre)-dfunc(x_

        x_niz.append(x_novo)

    x_opt = x_novo
    f_opt = func(x_opt)
    return x_opt, f_opt, iter, x_niz
```

In [ ]:
```
[x_opt, f_opt, iter, x_niz] = metod_secice(x0=-1, x1=0, epsilon=0.1)

x = np.linspace(-3, 3, 1000)
f = np.linspace(0, 0, len(x))
for i in range(0, len(x), 1):
    f[i] = func(x[i])

plt.plot(x, f, 'k--', label='f(x)')

for i in range(0, len(x_niz)-1):
    plt.plot([x_niz[i], x_niz[i+1]], [func(x_niz[i] ), func(x_niz[i+1] )]
plt.plot(x_opt, f_opt, '-or', label='max[f(x)]', markersize=5, markeredge
plt.grid(True)
```

```
plt.show()
print(f'Оптимум функции f(x) находится в точке {x_opt} и равен {f_opt}. A
```




```
Оптимум функции f(x) находится в точке 0.5077214520673443 и равен -0.59193
30806018976. Алгоритм выполнен за 7 итераций.
```

2. МЕТОДЫ ПРЯМОГО ПОИСКА

Методы прямого поиска требуют, чтобы критерий был унимодальной функцией. Таким образом, гарантируется, что уменьшая интервал поиска, мы не потеряем оптимум.

2.1 МЕТОД ФИБОНАЧЧИ

Метод Фибоначчи для поиска минимума одномерной функции - это численный метод, который использует последовательность чисел Фибоначчи для приближенного определения положения минимума (или максимума) функции. Сначала генерируется последовательность чисел Фибоначчи, пока длина начального интервала не станет меньше заданной допускной ошибки ϵ , т.е. критерий остановки становится:

$$F_n > \frac{b_0 - a_0}{\epsilon}. \tag{5}$$

Порядковый номер числа Фибоначчи в последовательности, n, удовлетворяющий критерию остановки, также представляет собой количество итераций. В течение каждой итерации необходимо вычислить новые точки внутри интервала следующим образом:

$$x_1 = a + \frac{F_{n-2}}{F_n}(b-a) \tag{6}$$

$$x_2 = a + b - x_1. \tag{7}$$

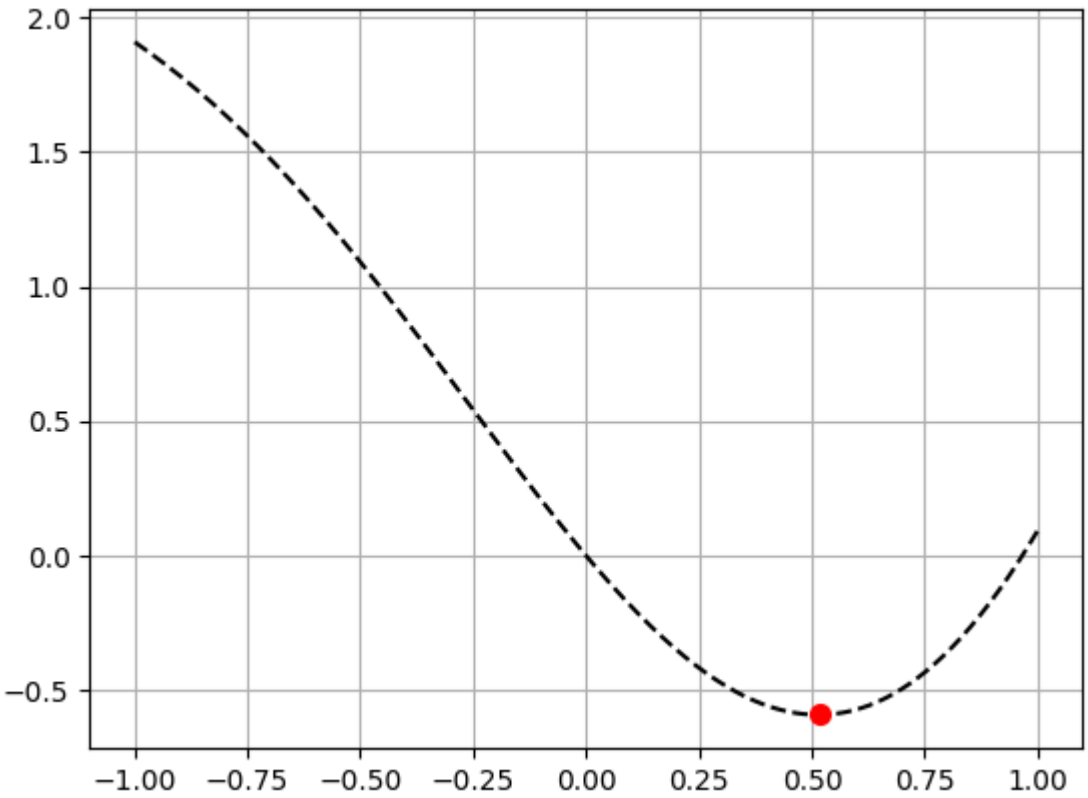
Какой диапазон будет отброшен, зависит от решаемой задачи - минимизации или максимизации функции.

In [ ]:
```
def fibonacijev_broj(n):
    if n == 1 or n == 2:
        f = 1
    else:
        fp = 1
        fpp = 1
        for i in range(3, n+1):
            f = fp + fpp
            fpp = fp
            fp = f
    return f
```

In [ ]:
```
def fibonacijev_metod(a, b, epsilon):

    n = 1
    while ((b-a)/epsilon) > fibonacijev_broj(n):
        n += 1

    x1 = a + fibonacijev_broj(n-2)/fibonacijev_broj(n)*(b-a)
    x2 = a + b - x1

    for i in range(2, n+1):
        if func(x1) <= func(x2):
            b = x2
            x1 = a + fibonacijev_broj(n-2)/fibonacijev_broj(n)*(b-a)
            x2 = a + b - x1
        else:
            a = x1
            x1 = a + fibonacijev_broj(n-2)/fibonacijev_broj(n)*(b-a)
            x2 = a + b - x1

    if func(x1) < func(x2):
        x_opt = x1
        f_opt = func(x_opt)
    else:
        x_opt = x2
        f_opt = func(x_opt)

    return x_opt, f_opt, n
```

In [ ]:
```
[x_opt, f_opt, iter] = fibonacijev_metod(a=-1, b=2, epsilon=0.1)

x = np.linspace(-1, 1, 1000)
f = np.linspace(0, 0, len(x))
for i in range(0, len(x), 1):
    f[i] = func(x[i])

plt.plot(x, f, 'k--', label='f(x)')
plt.plot(x_opt, f_opt, '-or', label='max[f(x)]', markersize=5, markeredge
plt.grid(True)
```

```
plt.show()
print(f'Minimum функции f(x) находится в точке {x_opt} и равен {f_opt}. A
```




```
Minimum функции f(x) находится в точке 0.5192708278113745 и равен -0.59202
28743564877. Алгоритм выполнен за 9 итераций. Число Фибоначчи - 34
```

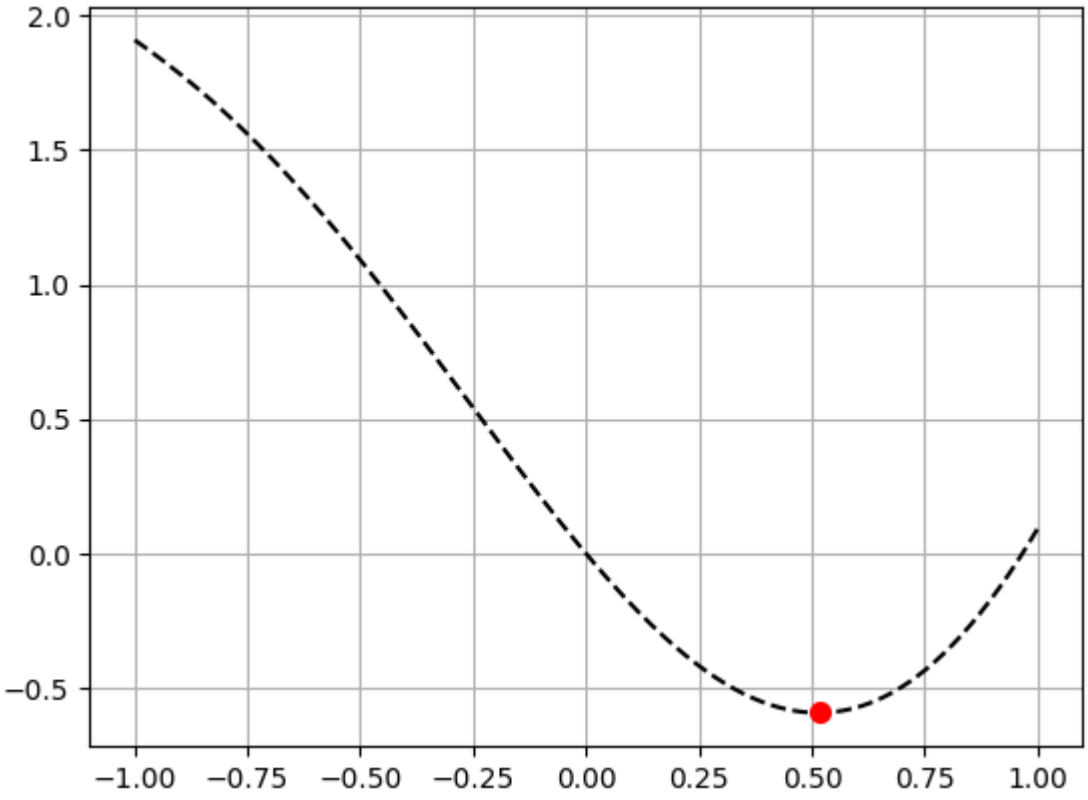
2.2 МЕТОД ЗОЛОТОГО СЕЧЕНИЯ

Этот метод похож на метод Фибоначчи, но не использует последовательность чисел Фибоначчи, а постоянное отношение. Идея этого метода заключается в том, чтобы разделить интервал поиска пополам в точке, которая делит интервал "золотым сечением". Следует отметить, что соотношение $\frac{F_{n-2}}{F_n}$ для больших чисел Фибоначчи постоянно и составляет около 0.38196. Это число фактически представляет собой "золотое сечение" и равно $c = \frac{3-\sqrt{5}}{2}$.

In [ ]:
```
def metod_zlatnog_preseka(a, b, epsilon):

    c = (3 - math.sqrt(5)) / 2

    x1 = a + c * (b - a)
    x2 = a + b - x1
    iter = 1

    x_niz = []

    while (b - a) > epsilon:
        iter += 1
        if func(x1) <= func(x2):
            b = x2
            x1 = a + c * (b - a)
            x2 = a + b - x1
        else:
            a = x1
```

```
            x1 = a + c * (b - a)
            x2 = a + b - x1

    if func(x1) < func(x2):
      x_opt = x1
      f_opt = func(x_opt)
    else:
      x_opt = x2
      f_opt = func(x_opt)
    return x_opt, f_opt, iter
```

In [ ]:
```
[x_opt, f_opt, iter] = metod_zlatnog_preseka(a=-1, b=2, epsilon=0.1)

x = np.linspace(-1, 1, 1000)
f = np.linspace(0, 0, len(x))
for i in range(0, len(x), 1):
    f[i] = func(x[i])

plt.plot(x, f, 'k--', label='f(x)')
plt.plot(x_opt, f_opt, '-or', label='max[f(x)]', markersize=5, markeredge
plt.grid(True)
plt.show()
print(f'Minimum функции f(x) находится в точке {x_opt} и равен {f_opt}. A
```




```
Minimum функции f(x) находится в точке 0.5197334262446374 и равен -0.59201
13802625233. Алгоритм выполнен за 9 итераций.
```