Метод Ломанных
Федоров Даниил Михайлович МетОпт 1.1

Как работает программа:

1. **Инициализация**: После того как мы определили функцию F(x) и границы отрезка $[x_{start}, x_{end}]$, программа создаёт массив точек, в который изначально включаются только границы отрезка точки x_{start} и x_{end}.
2. **Построение ломаной линии**: На основе этих точек строится ломаная линия, которая является **нижней огибающей функцией**. Ломаная линия состоит из прямых, соединяющих соседние точки. Для каждой такой прямой, которая соединяет две соседние точки на графике функции, строятся дополнительные прямые с наклонами ±L, где L — это приближённо рассчитанная **константа Липшица** для функции.
3. **Точки пересечения**: Прямые с наклонами ±L образуют точки пересечения между собой под графиком функции. Эти точки представляют собой приближённые минимумы на графике, потому что функция не может опускаться быстрее, чем эти прямые с наклонами ±L.
4. **Выбор минимальной точки**: Из всех точек пересечения выбирается точка, которая даёт минимальное значение функции. Эту точку добавляем в массив точек, то есть добавляем новую точку $(x_{new}, F(x_{new}))$ на график.
5. **Обновление ломаной линии**: После добавления новой точки строится новая ломаная линия, которая включает уже и новую точку. Этот процесс повторяется: на каждом шаге добавляется новая точка, и ломаная линия строится заново с учётом всех найденных точек.
6. **Завершение**: Процесс продолжается до тех пор, пока разница между значением функции в минимальной точке (которая определяется по ломаной линии) и значением функции в новой точке пересечения не станет меньше заданной точности eps, то есть пока $F(x_{min}) - y_{new} < eps$
7. **Результат**: Когда точность достигнута, точка $(x_{min}, F(x_{min}))$ считается минимальной точкой функции на заданном отрезке с

точностью eps. Эта точка возвращается как результат работы алгоритма.

1. Основной класс: ***`Main_class`***

1.1 Конструктор ***`__init__`***

Конструктор инициализирует следующие параметры:

- ***`func`*** – строковое представление функции, которую нужно минимизировать.
- ***`x_start`*** и ***`x_end`*** – границы отрезка, на котором будет искать минимум.
- ***`eps`*** – точность вычислений, до которой алгоритм будет искать минимум.

```python
class Main_class:  2 usages

    def __init__(self, func: str, x_start: float, x_end: float, eps=0.01) -> None:
        self.func_str = func
        self.x_start = x_start
        self.x_end = x_end
        self.eps = eps

        self.func = self._create_func()

        self.L = self._get_L()
        self.points = [(self.x_start, self.func(self.x_start)), (self.x_end, self.func(self.x_end))]
        self.intersect_cache = {}
        self.iteration_count = 0
```

Конструктор выполняет следующие шаги:

1. Преобразует строку с функцией в исполнимую лямбда-функцию, используя *`eval()`*.
2. Рассчитывает **константу Липшица** для функции (необходимую для метода Пиявского).
3. Инициализирует начальные точки на отрезке (границы отрезка).

4. Инициализирует пустой кэш для хранения точек пересечения прямых.

1.2 Метод *step*

```python
def step(self) -> bool:  1 usage
    self.iteration_count += 1

    x_new, lower_bound = self._find_next_point()

    f_new = self.func(x_new)

    self.points.append((x_new, f_new))
    self.points.sort(key=lambda p: p[0])

    current_best_f = min(p[1] for p in self.points)
    gap = current_best_f - lower_bound

    return gap < self.eps
```

Этот метод выполняет **один шаг метода Пиявского**:

1. Увеличивает счетчик итераций (*iteration_count*).
2. Находит следующую точку для исследования, используя метод *_find_next_point*.
3. Вычисляет значение функции в этой точке.
4. Добавляет новую точку в список и сортирует точки по координате *x*.
5. Проверяет, насколько хорошо текущая точка приближает минимум, сравнив минимальное значение функции с **нижней оценкой**. Если разница меньше заданной точности (*eps*), возвращает *True* (условие завершения).

1.3 Метод *plot*

```python
def plot(self, show_solution=True) -> None:  1 usage
    x_smooth = np.linspace(self.x_start, self.x_end,  num: 1000)
    y_smooth = [self.func(x) for x in x_smooth]

    if show_solution:
        x_points = [p[0] for p in self.points]
        y_points = [p[1] for p in self.points]

        x_polyline, y_polyline = self._build_polyline()

    plt.figure(figsize=(12, 7))
    plt.plot( *args: x_smooth, y_smooth, 'b-', linewidth=2, label='Функция f(x)')

    if show_solution:
        plt.plot( *args: x_polyline, y_polyline, 'r-', linewidth=1, label='Ломаная (нижняя оценка)')
        plt.scatter(x_points, y_points, color='red', s=20, label='Вычисленные точки')

        best_point = min(self.points, key=lambda p: p[1])
        plt.scatter( x: [best_point[0]],  y: [best_point[1]], color='green', s=100,
                    label=f'Минимум: f({best_point[0]:.3f}) = {best_point[1]:.3f}')

    plt.xlabel( xlabel: 'x', fontsize=12)
    plt.ylabel( ylabel: 'f(x)', fontsize=12)
    plt.title( label: f'Метод Пиявского - Итерация #{self.iteration_count}', fontsize=14)
    plt.grid( visible: True, alpha=0.2)
    plt.legend()
    plt.tight_layout()
    plt.show()
```

Метод рисует график функции и ломаной линии:

1. Создает плавный график функции, используя *np.linspace* для создания множества точек.
2. Если флаг *show_solution=True*, строит ломаную линию, которая является нижней огибающей функции, и отображает вычисленные точки.
3. Вычисляет точку минимума и отображает её на графике.
4. На графике будут отображены:
    - Функция (синий график).
    - Ломанная линия (красная линия).
    - Вычисленные точки (красные маркеры).
    - Минимальная точка (зелёный маркер).

1.4 Метод *solve*

```python
def solve(self, max_iterations=10000) -> tuple[float, float, float, int]:  1 usage
    start_time = time.time()

    for _ in range(max_iterations):
        if self.step():
            break

    elapsed_time = time.time() - start_time
    min_point = min(self.points, key=lambda p: p[1])

    return min_point[0], min_point[1], elapsed_time, self.iteration_count
```

Этот метод запускает процесс нахождения минимума:

1. Запускает итерации метода Пиявского до тех пор, пока не будет достигнута заданная точность или не будет превышено максимальное количество итераций (*`max_iterations`*).
2. Для каждой итерации вызывает метод *`step`*, который обновляет список точек и проверяет точность.
3. Возвращает:
    - *`minimum`* — минимальную координату `x` найденной точки.
    - *`min_value`* — значение функции в точке минимума.
    - *`elapsed_time`* — время, затраченное на вычисления.
    - *`iteration_count`* — количество итераций.

1.5 Вспомогательные методы

- ***`_find_next_point`***:

```python
def _find_next_point(self) -> tuple[float, float]:  1 usage
    best_x, best_lower_bound = None, float('inf')
    best_interval_x1, best_interval_x2 = None, None

    for i in range(len(self.points) - 1):
        x1, f1 = self.points[i]
        x2, f2 = self.points[i + 1]
        x_intersect, y_intersect = self._get_intersect_coord(x1, f1, x2, f2)

        if (x1, x2) not in self.intersect_cache:
            self.intersect_cache[(x1, x2)] = (x_intersect, y_intersect)

        if y_intersect < best_lower_bound:
            best_lower_bound = y_intersect
            best_x = x_intersect
            best_interval_x1, best_interval_x2 = x1, x2

    del self.intersect_cache[(best_interval_x1, best_interval_x2)]
    return best_x, best_lower_bound
```

Этот метод находит точку пересечения двух прямых, соединяющих соседние вычисленные точки. Пересечение этих прямых приближает минимальную точку. Метод выбирает точку пересечения, которая даёт наименьшее значение функции.

- ***_get_intersect_coord***:

```python
def _get_intersect_coord(self, x1: float, f1: float, x2: float, f2: float) -> tuple[float, float]:  2 usages
    if (x1, x2) in self.intersect_cache:
        return self.intersect_cache[(x1, x2)]

    x_intersect = ((f1 - f2) / (2 * self.L)) + ((x1 + x2) / 2)
    y_intersect = f1 - self.L * (x_intersect - x1)
    return x_intersect, y_intersect
```

Находит точку пересечения двух прямых. Прямые аппроксимируют функцию между двумя соседними точками на ломаной линии. Пересечение этих прямых даёт следующую точку для поиска минимума.

- ***_create_func***:

```python
def _create_func(self):  1 usage
    func_expr = self.func_str.replace( _old: "f(x)=",  _new: "").strip()
    return lambda x: eval(func_expr, {"math": math, "x": x})
```

Преобразует строку с функцией в исполнимую

лямбда-функцию с использованием *eval()*. Это позволяет работать с произвольными математическими выражениями.

- ***_get_L***:

```
def _get_L(self) -> float:  1 usage
    n_points = 1000
    x_samples = np.linspace(self.x_start, self.x_end, n_points)
    max_deriv = 0
    h = x_samples[1] - x_samples[0]

    for x in x_samples:
        if x - h >= self.x_start and x + h <= self.x_end:
            derivative = (self.func(x + h) - self.func(x - h)) / (2 * h)
            max_deriv = max(max_deriv, abs(derivative))

    return max_deriv * 1.2
```

Этот метод вычисляет **константу Липшица**, которая необходима для метода Пиявского. Он вычисляет производную функции в точках на отрезке и находит её максимальное значение.

- ***_build_polyline***:

```
def _build_polyline(self):  1 usage
    x_polyline, y_polyline = [], []

    for i in range(len(self.points) - 1):
        x1, f1 = self.points[i]
        x2, f2 = self.points[i + 1]

        x_polyline.append(x1)
        y_polyline.append(f1)

        x_intersect, y_intersect = self._get_intersect_coord(x1, f1, x2, f2)
        x_polyline.append(x_intersect)
        y_polyline.append(y_intersect)

    x_polyline.append(self.points[-1][0])
    y_polyline.append(self.points[-1][1])

    return x_polyline, y_polyline
```

Строит ломаную, которая аппроксимирует функцию. Ломаная

состоит из прямых, соединяющих соседние точки, и точек пересечения этих прямых.

2. Пример использования

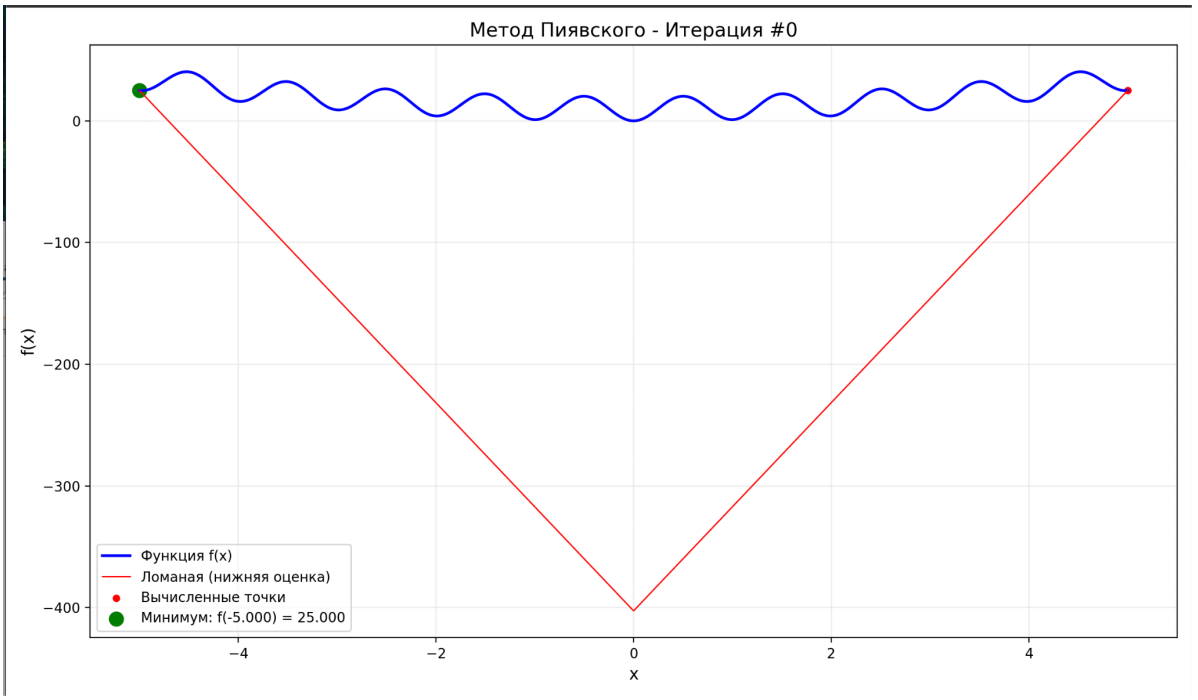
1. **Инициализация класса**:

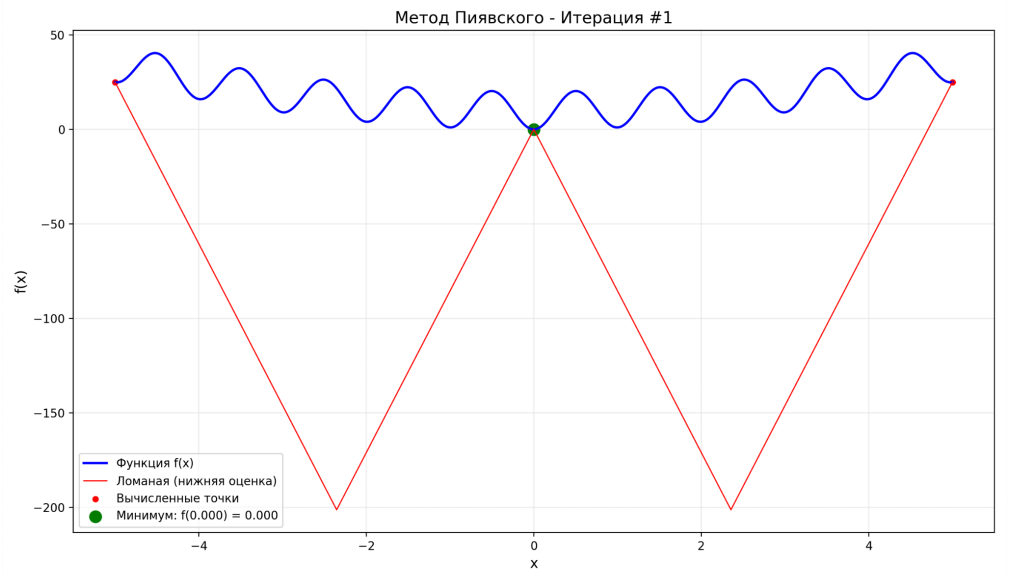
```
piyavsky = Main_class( func: "f(x)=10 + x**2 - 10 * math.cos(2 * math.pi * x)", -5, x_end: 5, eps=0.01)
```

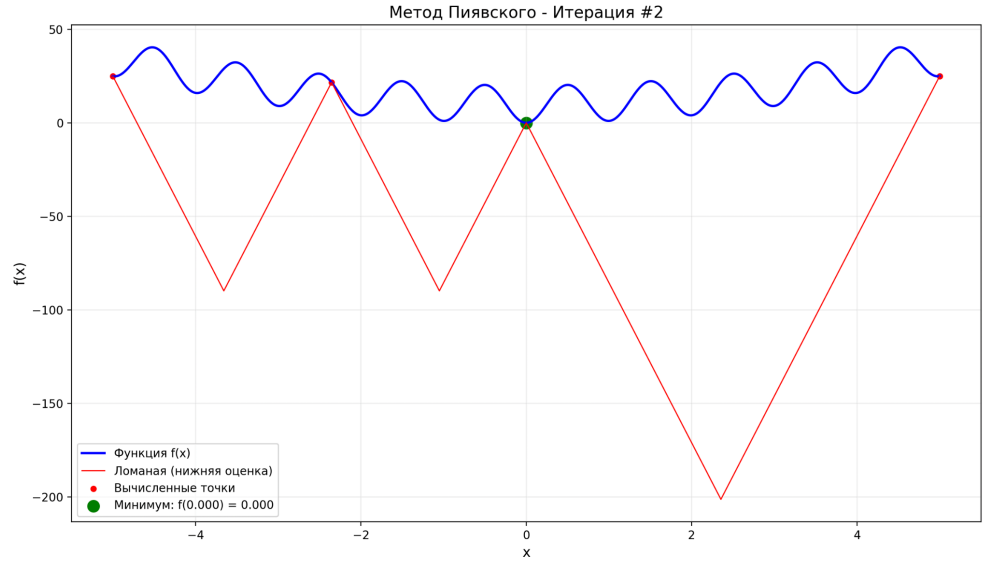
Возьмем функцию Растринга, а именно Для одномерной функции можно использовать её упрощённую версию.

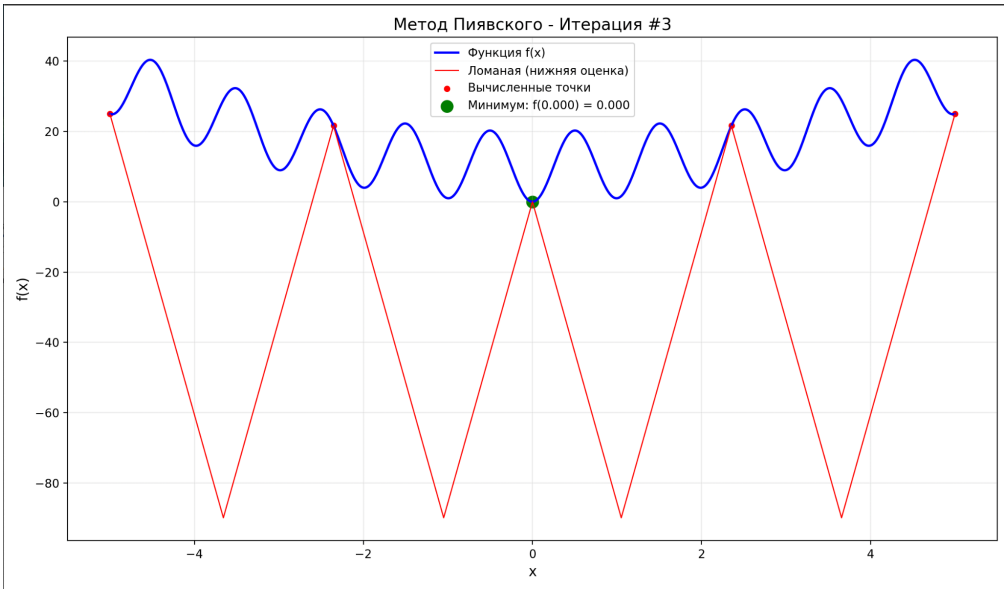
```
#Выполним 3 итераций
for _ in range(4):
    piyavsky.plot()
    piyavsky.step()
```

Результат:




Метод Пиявского - Итерация #1
f(x)
x
Функция f(x)
Ломаная (нижняя оценка)
Вычисленные точки
Минимум: f(0.000) = 0.000

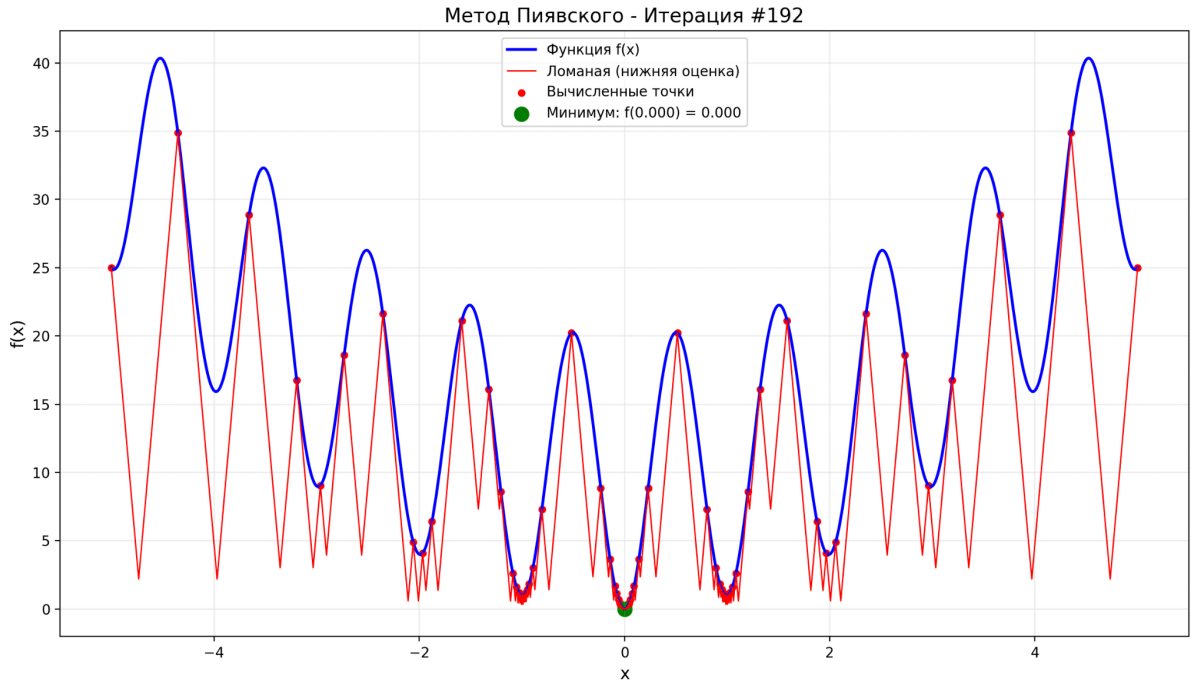

Метод Пиявского - Итерация #2
f(x)
x
Функция f(x)
Ломаная (нижняя оценка)
Вычисленные точки
Минимум: f(0.000) = 0.000


Метод Пиявского - Итерация #3
f(x)
x
Функция f(x)
Ломаная (нижняя оценка)
Вычисленные точки
Минимум: f(0.000) = 0.000

```
minimum, min_value, elapsed_time, iterations = piyavsky.solve()

print(f"Минимум найден: x = {minimum}, f(x) = {min_value}")
print(f"Число итераций: {iterations}")
print(f"Время выполнения: {elapsed_time:.4f} секунд")
piyavsky.plot(show_solution=True)
```

В конечном итоге алгоритм продолжал выполнять шаги до достижения минимальной точки с заданной точностью eps и выполнил 192 итерации




```
Минимум найден: x = 0.0, f(x) = 0.0
Число итераций: 192
Время выполнения: 0.0104 секунд
```

Вывод: данный код реализует метод Пиявского для нахождения глобального минимума функции на отрезке, используя ломаную для аппроксимации функции. Делает он это как визуально через график,

так и позволяет получит координаты минимума и оценить время выполнения.