Министерство науки и высшего образования Российской Федерации

федеральное государственное автономное образовательное учреждение высшего образования

“НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО”

Факультет	**Программной Инженерии и Компьютерной Техники**
Направление подготовки (специальность)	**Нейротехнологии и программирование**
Дисциплина	**Программирование на Python**

Лабораторная работа 10
ОТЧЕТ

Выполнил студент:	**Иголкин Владислав Андреевич (504623)**
Группа:	**P3124**
Преподаватель:	**Жуков Николай Николаевич (261087)**

г. Санкт-Петербург

2025 г.

Лабораторная работа №10

Оптимизация вычислений в Python

Цель работы

Изучить, как можно ускорить выполнение вычислительного кода в Python с помощью потоков и процессов на примере численного интегрирования.

Итерация 1 — Обычная реализация на Python

В первой итерации была реализована функция численного интегрирования методом прямоугольников на чистом Python.

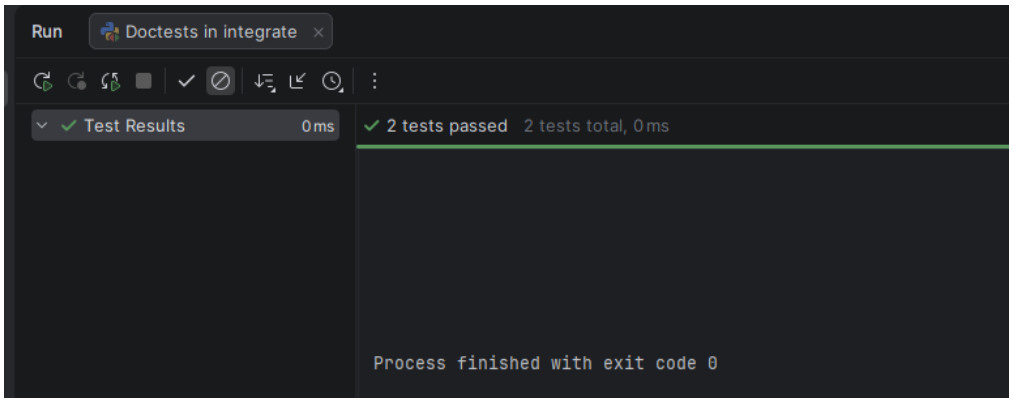
Функция вычисляет значение интеграла, последовательно проходя по всем шагам разбиения интервала. Для повышения точности используется большое количество итераций.

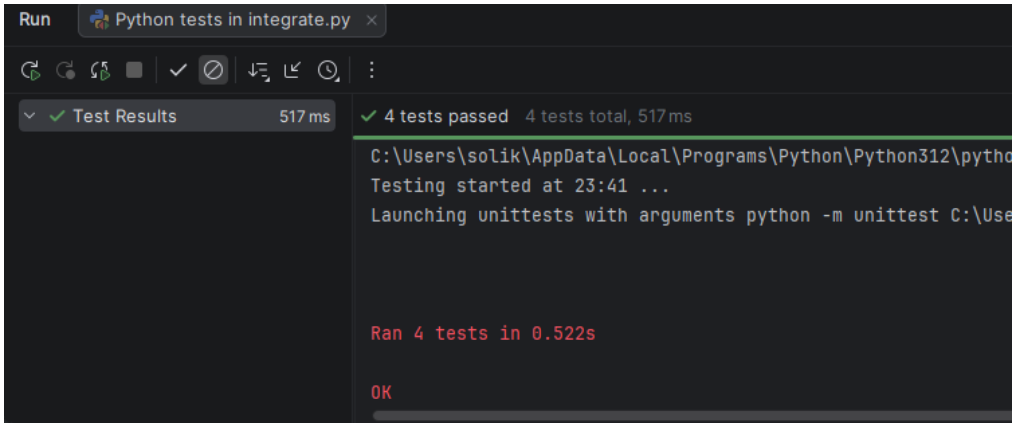
Корректность работы функции была проверена с помощью тестов и примеров в docstring. Также было измерено время выполнения при различном количестве итераций.

Вывод:
Функция работает правильно, но при большом числе итераций выполняется медленно, так как все вычисления происходят последовательно в одном потоке.

```
import math
import unittest
from typing import Callable


def integrate(f: Callable[[float], float], a: float, b: float, *, n_iter: int =
100_000) -> float:
    """
    Вычисляет определённый интеграл функции `f` на интервале [a, b] методом
прямоугольников.

    Метод основан на разбиении интервала [a, b] на `n_iter` равных частей и
    суммировании площадей прямоугольников, построенных на значениях функции
    в левом конце каждого подинтервала.

    Args:
        f (Callable[[float], float]): Функция одного аргумента, которую нужно
интегрировать.
        a (float): Левая граница интегрирования.
        b (float): Правая граница интегрирования.
        n_iter (int, optional): Количество шагов разбиения интервала.
                                Чем больше значение, тем выше точность.
                                По умолчанию 100 000.

    Returns:
        float: Приближённое значение определённого интеграла функции `f` на [a,
b].

    Raises:
        ValueError: Если n_iter <= 0 или a >= b.

    Examples:
        >>> round(integrate(math.cos, 0, math.pi, n_iter=1000),2)
        0.0
        >>> round(integrate(lambda x: x**2, 0, 1, n_iter=1000),2)
        0.33
    """
    if n_iter <= 0:
```

```
        raise ValueError("n_iter должно быть положительным")
    if a >= b:
        raise ValueError("Левая граница a должна быть меньше правой b")

    acc = 0.0
    step = (b - a) / n_iter
    for i in range(n_iter):
        acc += f(a + i * step) * step
    return acc


class TestIntegrate(unittest.TestCase):
    def test_trig_function(self):
        # интеграл cos(x) от 0 до pi равен 0
        result = integrate(math.cos, 0, math.pi, n_iter=1_000_000)
        self.assertAlmostEqual(result, 0.0, places=5)

    def test_polynomial_function(self):
        # интеграл x^2 от 0 до 1 равен 1/3
        result = integrate(lambda x: x**2, 0, 1, n_iter=1_000_000)
        self.assertAlmostEqual(result, 1/3, places=5)

    def test_n_iter_effect(self):
        # Проверка устойчивости к различному количеству итераций
        res_low = integrate(lambda x: x**2, 0, 1, n_iter=100)
        res_high = integrate(lambda x: x**2, 0, 1, n_iter=1_000_000)
        self.assertTrue(abs(res_high - res_low) > 1e-5)

    def test_invalid_parameters(self):
        # Проверка обработки ошибок
        with self.assertRaises(ValueError):
            integrate(lambda x: x**2, 1, 0)
        with self.assertRaises(ValueError):
            integrate(lambda x: x**2, 0, 1, n_iter=0)

if __name__ == "__main__":
    unittest.main()
```





Итерация 2 — Использование потоков

Во второй итерации вычисление интеграла было распараллелено с использованием потоков (ThreadPoolExecutor).

Интервал интегрирования разбивался на несколько частей, и каждая часть вычислялась в отдельном потоке. После завершения всех потоков результаты суммировались.

Были проведены замеры времени для разного количества потоков.

Наблюдение:
Использование потоков не привело к ускорению работы программы. В некоторых случаях время выполнения даже увеличивалось.

Причина:
В Python существует механизм GIL (Global Interpreter Lock), который не позволяет нескольким потокам одновременно выполнять вычисления. Из-за этого потоки не подходят для задач, активно использующих процессор.

Вывод:
Потоки неэффективны для вычислительно сложных задач в Python.

```
import math
import timeit
from typing import Callable
from concurrent.futures import ProcessPoolExecutor, as_completed

from integrate import *


def integrate_processes(
    f: Callable[[float], float],
    a: float,
    b: float,
    *,
    n_iter: int = 100_000,
    n_jobs: int = 2
) -> float:
    """
    Численное интегрирование с использованием процессов (ProcessPoolExecutor).

    Интервал [a, b] разбивается на n_jobs частей, каждая часть
    обрабатывается в отдельном процессе.
    """
    if n_jobs <= 0:
        raise ValueError("n_jobs должно быть положительным")
    if n_iter <= 0:
        raise ValueError("n_iter должно быть положительным")

    step = (b - a) / n_jobs
    iter_per_job = n_iter // n_jobs

    with ProcessPoolExecutor(max_workers=n_jobs) as executor:
        futures = [
            executor.submit(
                integrate,
                f,
                a + i * step,
                a + (i + 1) * step,
                n_iter=iter_per_job
            )
            for i in range(n_jobs)
        ]
```

```
        return sum(f.result() for f in as_completed(futures))


if __name__ == "__main__":
    print("Проверка корректности:")
    print(integrate(math.cos, 0, math.pi, n_iter=1_000_000))
    print(integrate_processes(math.cos, 0, math.pi, n_iter=1_000_000, n_jobs=4))

    print("\nЗамеры времени (процессы):")
    for jobs in [1, 2, 4, 6, 8]:
        t = timeit.timeit(
            lambda: integrate_processes(
                math.cos,
                0,
                math.pi,
                n_iter=1_000_000,
                n_jobs=jobs
            ),
            number=3
        )
        print(f"n_jobs={jobs}: {t:.5f} секунд")
```

```
Проверка корректности:
3.14159265351633327e-06
3.141592641087154e-06

Замеры времени (процессы):
n_jobs=1: 1.71257 секунд
n_jobs=2: 1.61062 секунд
n_jobs=4: 1.82174 секунд
n_jobs=6: 1.84547 секунд
n_jobs=8: 2.20649 секунд
```

Итерация 3 — Использование процессов

В третьей итерации для вычисления интеграла были использованы процессы (ProcessPoolExecutor).

Каждый процесс выполнялся независимо в своём интерпретаторе Python и мог использовать отдельное ядро процессора. Интервал интегрирования также делился на части, которые обрабатывались параллельно.

Были выполнены замеры времени при различном количестве процессов.

Наблюдение:
Использование процессов значительно сократило время выполнения программы по сравнению с обычной реализацией и вариантом с потоками.

Вывод:
Процессы являются эффективным способом ускорения вычислительных задач в Python, так как они не ограничены общим GIL.

```
import math
import timeit
from typing import Callable
from concurrent.futures import ThreadPoolExecutor, as_completed
```

```python
from integrate import *


def integrate_threaded(
    f: Callable[[float], float],
    a: float,
    b: float,
    *,
    n_iter: int = 100_000,
    n_jobs: int = 2
) -> float:
    """
    Численное интегрирование с использованием потоков (ThreadPoolExecutor).
    """
    step = (b - a) / n_jobs
    iter_per_job = n_iter // n_jobs

    with ThreadPoolExecutor(max_workers=n_jobs) as executor:
        futures = [
            executor.submit(
                integrate,
                f,
                a + i * step,
                a + (i + 1) * step,
                n_iter=iter_per_job
            )
            for i in range(n_jobs)
        ]

        return sum(f.result() for f in as_completed(futures))


if __name__ == "__main__":
    print("Проверка корректности:")
    print(integrate(math.cos, 0, math.pi, n_iter=1_000_000))
    print(integrate_threaded(math.cos, 0, math.pi, n_iter=1_000_000, n_jobs=4))

    print("\nЗамеры времени (потоки):")
    for jobs in [1, 2, 4, 6, 8]:
        t = timeit.timeit(
            lambda: integrate_threaded(
                math.cos,
                0,
                math.pi,
                n_iter=1_000_000,
                n_jobs=jobs
            ),
            number=3
        )
        print(f"n_jobs={jobs}: {t:.5f} секунд")
```

```
Проверка корректности:
3.1415926535163327e-06
3.141592641087154e-06

Замеры времени (потоки):
n_jobs=1: 0.33778 секунд
n_jobs=2: 0.35103 секунд
n_jobs=4: 0.33535 секунд
n_jobs=6: 0.33738 секунд
n_jobs=8: 0.33425 секунд
```

Общий вывод

- Обычная реализация проста, но медленная.
- Потоки не дают ускорения для вычислительных задач.
- Процессы позволяют эффективно ускорить выполнение программы.