**МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ**

**УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ**

**ГОМЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ**

**УНИВЕРСИТЕТ ИМЕНИ П. О. СУХОГО**

Факультет автоматизированных и информационных систем

Кафедра «Информатика»

Специальность 1-40 04 01 «Информатика и технологии программирования»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовому проекту

по дисциплине «Операционные системы и среды»

на тему: **«ИССЛЕДОВАНИЕ МЕТОДОВ ОРГАНИЗАЦИИ РАСПРЕДЕЛЕННЫХ ВЫЧИСЛЕНИЙ НА ПРИМЕРЕ РЕШЕНИЯ ЗАДАЧ ОБРАБОТКИ И ОТОБРАЖЕНИЯ ПОВЕРХНОСТЕЙ»**

Исполнитель: студентка гр. ИП-31

Процкая М. А.

Руководитель: старший преподаватель

Косинов Г. П.

Дата проверки: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Дата допуска к защите: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Дата защиты: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Оценка работы: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Подписи членов комиссии

по защите курсовой работы: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Гомель 2020

**Содержание**

Введение……………………………………………………..……………………...3

1 Обзор существующих методов решения задачи………………..……..………..4

1.1 Определение поверхности.……………....……….…….…………….…..4

1.2 Вычисление площади поверхности....………….…..………………...…..5

1.3 Построение поверхности…………....……….……….……………….…..8

[1.4](#_Toc468949622) Используемые средства разработки...…………..…..………….……….10

1.5 Получение статистических данных……………………………………..13

1.6 Синхронизация потоков…………………………………………………14

1.7 Распределение вычислений……………………………………………..15

[2](#_Toc468949627) Алгоритмический анализ.……...……………………………………………….25

2.1 Постановка задачи……………………....……….…….…………….…..25

2.2 Построение функциональной модели………….…..………………...…25

3 Разработка приложения…………….…………………………..……………….27

3.1 Разработка программного кода………....……….…….…………….…..27

3.2 Разработка графического интерфейса………….…..………………...…31

3.3 Определение погрешности численного метода..…….…………….…..33

[4 Анализ результатов работы и выводы ………………...………….……………](#_Toc468949632)35

[5 Тестирование и верификация………..………………...…………….……….…](#_Toc468949632)37

[5.1 Тестирование пользовательского интерфейса…...…………….………](#_Toc468949632)37

[5.2 Тестирование функционала………..………...……...…………..………](#_Toc468949632)39

[Заключение](#_Toc468949633)……………………………………………………………..………….40

[Список использованной литературы](#_Toc468949634)……………………………………..………41

[Приложение А](#_Toc468949635)……………………………………………………………..………42

[Приложение](#_Toc468949635) Б……………………………………………………….……..………51

**ВВЕДЕНИЕ**

В связи с динамичным развитием вычислительной техники и постоянно прогрессирующими технологиями, количество задач, для получения точного решения которых приходится прибегать к огромному количеству трудоемких вычислений, нелинейно растет. Порой, найти решение некоторых из них вручную и вовсе невозможно. Это происходит главным образом потому, что искомое решение обычно не выражается в привычных элементарных или других известных функциях. В таких случаях имеет место прибегнуть к соответствующим приближенным математическим вычислениям. В связи с ранее сказанным за последние годы математического прогресса все большее применение получили приближенные и численные методы математического анализа и приобрели исключительно важный характер.

Численные методы решения задач математического анализа ­­–­ это изучение алгоритмов и условий сходимости итерационных методов, определение границ применимости методов, исследования оценок погрешностей методов и вычислений. Основной целью использования численных методов является их реализация на ЭВМ, то есть составление программы для требуемого алгоритма и решения с ее помощью конкретной задачи.

Вычисление интегралов сложных поверхностей решаются, используя все возможности компьютерного процессора. Суть данного метода заключается в разделении задачи вычисление интеграла сложной поверхности на несколько частей и одновременном их решении при помощи ограниченного числа потоков.

Целью данного курсового проекта является разработка компонентного приложения подготовки данных для обработки и отрисовки поверхности с использованием многопоточности.

Задачи данного курсового проекта:

* изучить теоретический материал о методах решения задачи;
* разработать функциональную модель;
* разработать необходимые программные компоненты;
* выполнить алгоритмический анализ поставленной задачи;
* исследовать эффективность организации многопоточности.

Поверхность в геометрии и топологии — двумерное топологическое многообразие. Наиболее известными примерами поверхностей являются границы геометрических тел в обычном трёхмерном евклидовом пространстве. **1 ОБЗОР СУЩЕСТВУЮЩИХ МЕТОДОВ РЕШЕНИЯ ЗАДАЧ**

**1.1 Определение поверхности**

* + 1. Наиболее известными примерами поверхностей являются границы [геометрических тел](https://ru.wikipedia.org/wiki/%D0%A2%D0%B5%D0%BB%D0%BE_(%D0%B3%D0%B5%D0%BE%D0%BC%D0%B5%D1%82%D1%80%D0%B8%D1%8F)) в обычном трёхмерном евклидовом пространстве. С другой стороны, существуют поверхности (например, [бутылка Клейна](https://ru.wikipedia.org/wiki/%D0%91%D1%83%D1%82%D1%8B%D0%BB%D0%BA%D0%B0_%D0%9A%D0%BB%D0%B5%D0%B9%D0%BD%D0%B0)), которые нельзя вложить в трёхмерное евклидово пространство без привлечения [сингулярности](https://ru.wikipedia.org/wiki/%D0%9E%D1%81%D0%BE%D0%B1%D0%B5%D0%BD%D0%BD%D0%BE%D1%81%D1%82%D1%8C) или самопересечения.

«Двухмерность» поверхности подразумевает возможность реализовать на ней [метод координат](https://ru.wikipedia.org/wiki/%D0%9C%D0%B5%D1%82%D0%BE%D0%B4_%D0%BA%D0%BE%D0%BE%D1%80%D0%B4%D0%B8%D0%BD%D0%B0%D1%82), хотя и необязательно для всех точек. Так, поверхность [Земли](https://ru.wikipedia.org/wiki/%D0%97%D0%B5%D0%BC%D0%BB%D1%8F) (в идеале) представляет собой двумерную [сферу](https://ru.wikipedia.org/wiki/%D0%A1%D1%84%D0%B5%D1%80%D0%B0), широта и долгота каждой точки которой являются её координатами (за исключением [полюсов](https://ru.wikipedia.org/wiki/%D0%93%D0%B5%D0%BE%D0%B3%D1%80%D0%B0%D1%84%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%BF%D0%BE%D0%BB%D1%8E%D1%81) и [180-го меридиана](https://ru.wikipedia.org/wiki/180-%D0%B9_%D0%BC%D0%B5%D1%80%D0%B8%D0%B4%D0%B8%D0%B0%D0%BD)).

Концепция поверхности применяется в [физике](https://ru.wikipedia.org/wiki/%D0%A4%D0%B8%D0%B7%D0%B8%D0%BA), [инженерном деле](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D0%B6%D0%B5%D0%BD%D0%B5%D1%80%D0%BD%D0%BE%D0%B5_%D0%B4%D0%B5%D0%BB%D0%BE), [компьютерной графике](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%BD%D0%B0%D1%8F_%D0%B3%D1%80%D0%B0%D1%84%D0%B8%D0%BA%D0%B0) и прочих областях при изучении физических объектов. Например, анализ [аэродинамических качеств](https://ru.wikipedia.org/wiki/%D0%90%D1%8D%D1%80%D0%BE%D0%B4%D0%B8%D0%BD%D0%B0%D0%BC%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%BE%D0%B5_%D0%BA%D0%B0%D1%87%D0%B5%D1%81%D1%82%D0%B2%D0%BE) самолёта базируется на обтекании потоком воздуха его поверхности.

* + 1. Поверхность определяется как множество [точек](https://ru.wikipedia.org/wiki/%D0%A2%D0%BE%D1%87%D0%BA%D0%B0_(%D0%B3%D0%B5%D0%BE%D0%BC%D0%B5%D1%82%D1%80%D0%B8%D1%8F)), координаты которых удовлетворяют определённому виду уравнений:

(1)

Если функция [непрерывна](https://ru.wikipedia.org/wiki/%D0%9D%D0%B5%D0%BF%D1%80%D0%B5%D1%80%D1%8B%D0%B2%D0%BD%D0%B0%D1%8F_%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F) в некоторой точке и имеет в ней непрерывные частные производные, по крайней мере одна из которых не обращается в нуль, то в окрестности этой точки поверхность, заданная уравнением (1), будет правильной поверхностью.

Помимо указанного выше неявного способа задания, поверхность может быть определена явно, если одну из переменных, например, , можно выразить через остальные:

 (2)

* + 1. Интуитивно простую поверхность можно представить как кусок [плоскости](https://ru.wikipedia.org/wiki/%D0%9F%D0%BB%D0%BE%D1%81%D0%BA%D0%BE%D1%81%D1%82%D1%8C_(%D0%B3%D0%B5%D0%BE%D0%BC%D0%B5%D1%82%D1%80%D0%B8%D1%8F)), подвергнутый [непрерывным](https://ru.wikipedia.org/wiki/%D0%9D%D0%B5%D0%BF%D1%80%D0%B5%D1%80%D1%8B%D0%B2%D0%BD%D0%BE%D1%81%D1%82%D1%8C_(%D0%BC%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)) [деформациям](https://ru.wikipedia.org/wiki/%D0%94%D0%B5%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%86%D0%B8%D1%8F) ([растяжениям, сжатиям](https://ru.wikipedia.org/wiki/%D0%90%D1%84%D0%B8%D0%BD%D0%BD%D1%8B%D0%B5_%D0%BF%D1%80%D0%B5%D0%BE%D0%B1%D1%80%D0%B0%D0%B7%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F) и [изгибаниям](https://ru.wikipedia.org/wiki/%D0%98%D0%B7%D0%B3%D0%B8%D0%B1)).

Простой поверхностью называется образ [гомеоморфного](https://ru.wikipedia.org/wiki/%D0%93%D0%BE%D0%BC%D0%B5%D0%BE%D0%BC%D0%BE%D1%80%D1%84%D0%B8%D0%B7%D0%BC) отображения (то есть взаимно однозначного и взаимно непрерывного отображения) внутренности единичного квадрата. Этому определению можно дать [аналитическое](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%B0%D0%BD%D0%B0%D0%BB%D0%B8%D0%B7) выражение.

Пусть на плоскости с прямоугольной системой координат и задан [квадрат](https://ru.wikipedia.org/wiki/%D0%9A%D0%B2%D0%B0%D0%B4%D1%80%D0%B0%D1%82), координаты внутренних точек которого удовлетворяют неравенствам , . Гомеоморфный образ квадрата в пространстве с [прямоугольной системой координат](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D1%8F%D0%BC%D0%BE%D1%83%D0%B3%D0%BE%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0_%D0%BA%D0%BE%D0%BE%D1%80%D0%B4%D0%B8%D0%BD%D0%B0%D1%82) , , задаётся при помощи формул , , ([параметрическое задание поверхности](https://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D1%80%D0%B0%D0%BC%D0%B5%D1%82%D1%80%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B5_%D0%BF%D0%BE%D0%B2%D0%B5%D1%80%D1%85%D0%BD%D0%BE%D1%81%D1%82%D0%B8)). При этом от функций , и требуется, чтобы они были [непрерывными](https://ru.wikipedia.org/wiki/%D0%9D%D0%B5%D0%BF%D1%80%D0%B5%D1%80%D1%8B%D0%B2%D0%BD%D0%B0%D1%8F_%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F) и чтобы для различных точек и были различными соответствующие точки и .

Примером простой поверхности является полусфера. Вся же [сфера](https://ru.wikipedia.org/wiki/%D0%A1%D1%84%D0%B5%D1%80%D0%B0) не является простой поверхностью. Это вызывает необходимость дальнейшего обобщения понятия поверхности.

Подмножество пространства, у каждой точки которого есть окрестность, являющаяся простой поверхностью, называется правильной поверхностью.

* + 1. В [дифференциальной геометрии](https://ru.wikipedia.org/wiki/%D0%94%D0%B8%D1%84%D1%84%D0%B5%D1%80%D0%B5%D0%BD%D1%86%D0%B8%D0%B0%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D0%B3%D0%B5%D0%BE%D0%BC%D0%B5%D1%82%D1%80%D0%B8%D1%8F) исследуемые поверхности обычно подчинены условиям, связанным с возможностью применения методов дифференциального исчисления. Как правило, это — условия гладкости поверхности, то есть существования в каждой точке поверхности определённой [касательной](https://ru.wikipedia.org/wiki/%D0%9A%D0%B0%D1%81%D0%B0%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D0%BF%D1%80%D1%8F%D0%BC%D0%B0%D1%8F) плоскости, кривизны и т. д. Эти требования сводятся к тому, что функции, задающие поверхность, предполагаются однократно, дважды, трижды, а в некоторых вопросах — неограниченное число раз [дифференцируемыми](https://ru.wikipedia.org/wiki/%D0%94%D0%B8%D1%84%D1%84%D0%B5%D1%80%D0%B5%D0%BD%D1%86%D0%B8%D1%80%D1%83%D0%B5%D0%BC%D0%B0%D1%8F_%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F) или даже [аналитическими функциями](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B0%D0%BB%D0%B8%D1%82%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B5_%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%B8). При этом дополнительно накладывается условие регулярности.

Случай неявного задания. Поверхность, заданная уравнением , является гладкой регулярной поверхностью, если , функция F [непрерывно дифференцируема](https://ru.wikipedia.org/wiki/%D0%94%D0%B8%D1%84%D1%84%D0%B5%D1%80%D0%B5%D0%BD%D1%86%D0%B8%D1%80%D1%83%D0%B5%D0%BC%D0%B0%D1%8F_%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F) в своей области определения Ω, а её частные производные одновременно не обращаются в нуль (условие правильности) на всём множестве Ω:

(3)

Случай параметрического задания. Зададим поверхность [векторным уравнением](https://ru.wikipedia.org/wiki/%D0%92%D0%B5%D0%BA%D1%82%D0%BE%D1%80-%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F) , или, что то же самое, тремя уравнениями в координатах:

(4)

Эта система уравнений задаёт гладкую регулярную поверхность, если выполнены условия:

* система устанавливает взаимно однозначное соответствие между образом и прообразом Ω;
* функции непрерывно дифференцируемы в Ω;
* выполнено условие не вырожденности:

(5)

Геометрически последнее условие означает, что векторы нигде не параллельны. Параметры  можно рассматривать как внутренние координаты точек поверхности. Фиксируя одну из координат, выводятся два семейства координатных кривых, покрывающих поверхность координатной сеткой.

Случай явного задания. Поверхность S может быть определена как график функции  тогда S является гладкой регулярной поверхностью, если функция f дифференцируема. Этот вариант можно рассматривать как частный случай параметрического задания: ; ; [1].

* 1. Вычисление площади поверхности
     1. Если везде в области *D* на координатной плоскости для формулы

(6)

положить , то, в соответствии со своим геометрическим смыслом, двойной интеграл будет численно равен площади S области интегрирования D, то есть

(7)

В полярной системе координат эта же самая формула приобретает вид

(8)

* + 1. Численное интегрирование методом трапеций.

Метод трапеций − метод численного интегрирования функции одной переменной, заключающийся в замене на каждом элементарном отрезке подынтегральной функции на многочлен первой степени, то есть линейную функцию. Площадь под графиком функции аппроксимируется прямоугольными трапециями. Алгебраический порядок точности равен 1.

Если отрезок {\displaystyle \left[a,b\right]}[a,b] является элементарным и не подвергается дальнейшему разбиению, значение интеграла можно найти по формуле:

Это простое применение формулы для площади трапеции − произведение полусуммы оснований, которыми в данном случае являются значения функции в крайних точках отрезка, на высоту (длину отрезка интегрирования) [3]. Схема алгоритма представлена в приложении Б (схема Б.1).

* + 1. Вывод с помощью одномерных интегралов. Так как уже известны формулы прямоугольников для интегралов от функций одной переменной, то удобно представить двойной интеграл через два интеграла, каждый из которых будет вычисляться от функции одной переменной и может быть численно найден с помощью уже известной нам одномерной формулы прямоугольников. С этой целью введем вспомогательную функцию g(x):

(10)

Каждый из интегралов

можно вычислить с помощью формул численного интегрирования для одномерных интегралов [2].

* 1. Построение поверхности

Для построения поверхности был выбран язык *Python*.

*Python* −  [высокоуровневый язык программирования](https://ru.wikipedia.org/wiki/%D0%92%D1%8B%D1%81%D0%BE%D0%BA%D0%BE%D1%83%D1%80%D0%BE%D0%B2%D0%BD%D0%B5%D0%B2%D1%8B%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F) общего назначения, ориентированный на повышение производительности разработчика и читаемости кода. [Синтаксис](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D0%BD%D1%82%D0%B0%D0%BA%D1%81%D0%B8%D1%81_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) ядра *Python* минималистичен.

В то же время [стандартная библиотека](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D0%B0%D0%BD%D0%B4%D0%B0%D1%80%D1%82%D0%BD%D0%B0%D1%8F_%D0%B1%D0%B8%D0%B1%D0%BB%D0%B8%D0%BE%D1%82%D0%B5%D0%BA%D0%B0_Python) включает большой объём полезных функций [5].

Основными архитектурными чертами языка являются: [динамическая типизация](https://ru.wikipedia.org/wiki/%D0%94%D0%B8%D0%BD%D0%B0%D0%BC%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B0%D1%8F_%D1%82%D0%B8%D0%BF%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F), [автоматическое управление памятью](https://ru.wikipedia.org/wiki/%D0%A1%D0%B1%D0%BE%D1%80%D0%BA%D0%B0_%D0%BC%D1%83%D1%81%D0%BE%D1%80%D0%B0_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)), механизм [обработки исключений](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B0_%D0%B8%D1%81%D0%BA%D0%BB%D1%8E%D1%87%D0%B5%D0%BD%D0%B8%D0%B9), поддержка [многопоточных вычислений](https://ru.wikipedia.org/wiki/%D0%9C%D0%BD%D0%BE%D0%B3%D0%BE%D0%BF%D0%BE%D1%82%D0%BE%D1%87%D0%BD%D0%BE%D1%81%D1%82%D1%8C), высокоуровневые [структуры данных](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85).

Поддерживается разбиение программ на [модули](https://ru.wikipedia.org/wiki/%D0%9C%D0%BE%D0%B4%D1%83%D0%BB%D1%8C_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)), которые, в свою очередь, могут объединяться в пакеты.

Построения поверхности в этом языке программирования самая тривиальная задача, которая может быть.

Для ее выполнения нужно следующие:

* версия *Matplotlib* не ниже 0.99;
* математическая библиотека *numpy*.

Давайте рассмотри самый простой пример:

Возьмем следующий пример:

Для начала нужно подготовить данные для рисования. Нам понадобятся три двумерные матрицы: матрицы *X* и *Y* будут хранить координаты сетки точек, в которых будет вычисляться приведенная выше функция, а матрица *Z* будет хранить значения этой функции (11) в соответствующей точке.

Если необходимо нарисовать трехмерный график на эквидистантной сетке (у которой расстояние между точками одинаковое), то для создания матриц, которые будут хранить координаты, поможет функция *meshgrid()* из библиотеки *numpy*.

Эта функция создает двумерные матрицы сеток по одномерным массивам. Работа этой функции очень наглядно показана в документации к *numpy*.

Чтобы отделить подготовку данных от самого рисования, создание сетки и расчет функции выделим в отдельную функцию:

def makeData ():

# Строим сетку в интервале от -10 до 10 с шагом 0.1 по обоим координатам

x = numpy.arange (-10, 10, 0.1)

y = numpy.arange (-10, 10, 0.1)

# Создаем двумерную матрицу-сетку

xgrid, ygrid = numpy.meshgrid(x, y)

# В узлах рассчитываем значение функции

zgrid = numpy.sin (xgrid) \* numpy.sin (ygrid) / (xgrid \* ygrid)

return xgrid, ygrid, zgrid

Эта функция возвращает три двумерные матрицы: *x, y, z*. Координаты x и y лежат в интервале от -10 до 10 с шагом 0.1.

Теперь возвращаемся непосредственно к рисованию. Чтобы отобразить наши данные, достаточно вызвать метод *plot\_surface()* экземпляра класса *Axes3D*, в который передадим полученные с помощью функции *makeData()* двумерные матрицы.

Теперь наш пример выглядит следующим образом:

import pylab

from mpl\_toolkits.mplot3d import Axes3D

import numpy

def makeData ():

x = numpy.arange (-10, 10, 0.1)

y = numpy.arange (-10, 10, 0.1)

xgrid, ygrid = numpy.meshgrid(x, y)

zgrid = numpy.sin (xgrid) \* numpy.sin (ygrid) / (xgrid \* ygrid)

return xgrid, ygrid, zgrid

x, y, z = makeData()

fig = pylab.figure()

axes = Axes3D(fig)

axes.plot\_surface(x, y, z)

pylab.show()

Если запустить этот скрипт, то появится окно с синей поверхностью, представленной на рисунке 1.1:

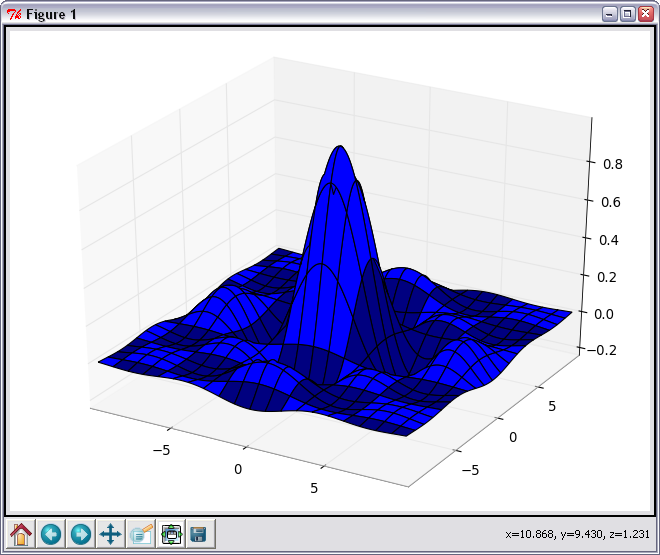


Рисунок 1.1 – График поверхности, построенный в *Python*

Отсюда сделаем вывод, что скрипт *Python* будет использоваться для построения поверхности, так как он прост и предоставляют стандартный шаблон, что очень удобно.

* 1. Используемые средства разработки

В ходе выполнения работы был использован вышеупомянутый язык *Python*, а так же объектно-ориентированный язык *С#* и технология *Windows Forms.*

Выбор проводился из двух языков: *C#* и *Java*.

Язык *Java* разработан компанией *Sun Microsystems*, создателем которого был Джеймс Гослинг, и выпущен в 1995 году в качестве основных компонентов компании *Sun Microsystems* − *Java* платформ (*Java 1.0 [J2SE]*).

По состоянию на 2018 год последней версией *Java Standard Edition* является 10 (*J2SE*). С развитием *Java*, и её широкой популярностью, несколько конфигураций были построены для различных типов платформ. Например: *J2EE* − приложения для предприятий, *J2ME* − для мобильных приложений.

*Sun Microsystems* переименовала прежнюю версию J2 и ввела новые: *Java SE, Java EE* и *Java* *ME*. Введение в программирование *Java* различных версий подтверждало знаменитый слоган компании «Напиши один раз, запускай везде».

Язык программирования *Java* имеет следующий список достоинств:

* Объектно−ориентированный: в *Java* все является объектом. Дополнение может быть легко расширено, так как он основан на объектной модели.
* Платформонезависимый: в отличие от многих других языков, включая *C* и *C++,* *Java*, когда был создан, он не компилировался в платформе конкретной машины, а в независимом от платформы байт−коде. Этот байт код распространяется через интернет и интерпретируется в *Java Virtual Machine (JVM)*, на которой он в настоящее время работает.
* Простой: процессы изучения и введение в язык программирования *Java* остаются простыми. Если основные концепции объектно-ориентированного программирования понятны, то он будет прост в освоении.
* Безопасный: методы проверки подлинности основаны на шифровании с открытым ключом.
* Архитектурно-нейтральный: компилятор генерирует архитектурно-нейтральные объекты формата файла, что делает скомпилированный код исполняемым на многих процессорах, с наличием системе *Java Runtime*.
* Портативный: архитектурно-нейтральный и не имеющий зависимости от реализации аспектов спецификаций − все это делает *Java* портативным. Компилятор в *Java* написан на *ANSI* C с чистой переносимостью, который является подмножеством *POSIX*.
* Прочный: выполняет усилия, чтобы устранить ошибки в различных ситуациях, делая упор в основном на время компиляции, проверку ошибок и проверку во время выполнения.
* Многопоточный: функции многопоточности, можно писать программы, которые могут выполнять множество задач одновременно. Введение в язык *Java* этой конструктивной особенности позволяет разработчикам создавать отлаженные интерактивные приложения.
* Интерпретированный: *Java* байт−код переводится на лету в машинные инструкции и нигде не сохраняется. Делая процесс более быстрым и аналитическим, поскольку связывание происходит как дополнительное с небольшого весом процесса.
* Высокопроизводительный: введение *Just-In-Time* компилятора, позволило получить высокую производительность.
* Распространенный: предназначен для распределенной среды интернета.
* Динамический: программирование на *Java* считается более динамичным, чем на *C* или *C++*, так как он предназначен для адаптации к меняющимся условиям. Программы могут выполнять обширное количество во время обработки информации, которая может быть использована для проверки и разрешения доступа к объектам на время выполнения.

*C#* − элегантный, типобезопасный объектно−ориентированный язык, предназначенный для разработки разнообразных безопасных и мощных приложений, выполняемых в среде *.NET Framework*. С помощью языка *C#* можно создавать обычные приложения Windows, приложения "клиент-сервер", приложения баз данных и т. д. *Visual C#* предоставляет развитый редактор кода, конструкторы с удобным пользовательским интерфейсом, встроенный отладчик и множество других средств, упрощающих разработку приложений на базе языка *C#* и *.NET Framework*.

Синтаксис *C#* очень выразителен, но прост в изучении. Все, кто знаком с языками *C, C++* или *Java* с легкостью узнают синтаксис с фигурными скобками, характерный для языка C#. Разработчики, знающие любой из этих языков, как правило, смогут добиться эффективной работы с языком *C#* за очень короткое время. Синтаксис *C#* делает проще то, что было сложно в *C++*, и обеспечивает мощные возможности, такие как типы значений *Nullable*, перечисления, делегаты, лямбда-выражения и прямой доступ к памяти, чего нет в *Java*. *C#* поддерживает универсальные методы и типы, обеспечивая более высокий уровень безопасности и производительности, а также итераторы, позволяющие при реализации коллекций классов определять собственное поведение итерации, которое может легко использоваться в клиентском коде. Выражения *LINQ* делают строго типизированный запрос очень удобной языковой конструкцией [4].

Как объектно−ориентированный язык, *C#* поддерживает понятия инкапсуляции, наследования и полиморфизма. Все переменные и методы, включая метод *Main* − точку входа приложения − инкапсулируются в определения классов. Класс может наследовать непосредственно из одного родительного класса, но может реализовывать любое число интерфейсов. Для методов, которые переопределяют виртуальные методы в родительском классе, необходимо ключевое слово *override*, чтобы исключить случайное повторное определение. В языке *C#* структура похожа на облегченный класс: это тип, распределяемый в стеке, реализующий интерфейсы, но не поддерживающий наследование.

В дополнение к основным описанным объектно-ориентированным принципам, язык *C#* упрощает разработку компонентов программного обеспечения благодаря нескольким инновационным конструкциям языка, в число которых входят следующие:

* инкапсулированные сигнатуры методов, называемые делегатами, которые поддерживают типобезопасные уведомления о событиях;
* свойства, выступающие в роли методов доступа для закрытых переменных−членов;
* *LINQ*, предлагающий встроенные возможности запросов в различных источниках данных.

Если потребуется обеспечить взаимодействие с другим программным обеспечением *Windows*, таким как объекты *COM* или собственные библиотеки *DLL Win32*, в языке *C#* можно использовать процесс, который называется "*Interop*". Процесс *Interop* позволяет программам на *C#* выполнять практически любые действия, которые может выполнять исходное приложение на *C++*. Язык *C#* поддерживает даже указатели и понятие "небезопасного" кода для тех случаев, когда прямой доступ к памяти имеет крайне важное значение.

Процесс построения *C#* по сравнению с *C* и *C++* прост и является более гибким, чем в *Java*. Нет отдельных файлов заголовка, а методы и типы не требуется объявлять в определенном порядке. В исходном файле *C#* может быть определено любое число классов, структур, интерфейсов и событий.

Таким образом остановимся на языке *C#.*

* 1. Получение статистических данных

В процессе анализа информации часто возникает потребность в обработке данных из документов в формате *MS Excel*.

В ходе выполнения курсового проекта будут получены статистические данные, представляющие собой зависимость времени выполнения вычислений от количества потоков. Возникает необходимость проанализировать и продемонстрировать где-либо эти данные. Исходя из задания, был сделан вывод, что статистику удобнее отобразить при помощи графика в *Excel*.

Для работы с *Excel* была выбрана библиотека *Microsoft.Office.Interpop.Excel*, предоставляющая программный интерфейс для взаимодействия с объектами *MS Excel*.

Преимущества данной библиотеки:

* создана корпорацией *Microsoft*, следовательно, взаимодействие с объектами программ пакета *MS Office* реализовано наиболее оптимально;
* нужный пакет *Visual Studio Tool for Office* поставляется вместе с *Visual Studio* (достаточно отметить его при установке *VS*).

Недостаток:

* для того, чтобы написанная программа работала на компьютере пользователя, необходимо, чтобы на нем была установлены программа *MS Excel*. Поэтому такой подход плохо подходит для серверных решений. Программа не будет являться кроссплатформенной.
  1. Синхронизация потоков

Нередко в потоках используются некоторые разделяемые ресурсы, общие для всей программы. Это могут быть общие переменные, файлы, другие ресурсы. Например:

class Program

{

static int x=0;

static void Main(string[] args)

{

for (int i = 0; i < 5; i++)

{

Thread myThread = new Thread(Count);

…

}

Console.ReadLine();

}

public static void Count()

{

x = 1;

for (int i = 1; i < 9; i++)

{

Console.WriteLine("{0}: {1}", Thread.CurrentThread.Name, x);

x++;

Thread.Sleep(100);

}

}

}

Здесь запускаются пять потоков, которые работают с общей переменной *x*, и предполагается, что метод выведет все значения *x* от 1 до 8. И так для каждого потока. Однако в реальности в процессе работы будет происходить переключение между потоками, и значение переменной x становится непредсказуемым.

Решение проблемы состоит в том, чтобы синхронизировать потоки и ограничить доступ к разделяемым ресурсам на время их использования каким-нибудь потоком. Для этого используется ключевое слово *lock*. Оператор *lock* определяет блок кода, внутри которого весь код блокируется и становится недоступным для других потоков до завершения работы текущего потока. И можно переделать предыдущий пример следующим образом:

class Program

{

static int x=0;

static object locker = new object();

static void Main(string[] args)

{

…

}

public static void Count()

{

lock (locker)

{

…

}

}

}

Для блокировки с ключевым словом *lock* используется объект-заглушка, в данном случае это переменная *locker*. Когда выполнение доходит до оператора *lock*, объект *locker* блокируется, и на время его блокировки монопольный доступ к блоку кода имеет только один поток. После окончания работы блока кода, объект *locker* освобождается и становится доступным для других потоков.

* 1. Распределение вычислений

Основной целью парадигмы параллелизма данных является полное устранение задач из поля зрения и их замена высокоуровневой абстракцией - параллельными циклами. Иными словами, распараллеливается не реализация алгоритма, а данные, которыми она оперирует. Библиотека *Task Parallel Library* предлагает несколько вариантов поддержки параллелизма данных.

***1.7.1*** Циклы *for* и *foreach* часто являются отличными кандидатами для распараллеливания. В действительности, еще на заре развития параллельных вычислений предпринимались попытки автоматического распараллеливания таких циклов. Некоторые попытки воплотились в языковые конструкции или расширения, такие как стандарт *OpenMP* (описывающий такие директивы, как *#pragma omp parallel for*, обеспечивающую распараллеливание циклов *for*). Библиотека *Task Parallel Library* предоставляет поддержку распараллеливания циклов посредством явных методов, очень близких своим языковым эквивалентам. Речь идет о методах *Parallel.For()* и *Parallel.ForEach(),* максимально близко имитирующих поведение циклов *for* и *foreach*.

В примере поиска простых чисел имелся цикл, который выполнял итерации по большому диапазону чисел, проверял каждое из них на принадлежность к категории простых чисел и добавлял их в коллекцию, как показано ниже:

for (uint number = start; number < end; ++number)

{

if (IsPrime(number))

{

primes.Add(number);

}

}

Преобразование этого кода, с целью задействовать в нем метод *Parallel.For()* - по большей степени механическая задача, если не считать необходимость синхронизации доступа к коллекции простых чисел (существуют более удачные решения, такие как агрегирование, которые представлено ниже):

Parallel.For(start, end, number => {

if (IsPrime(number)) {

lock (primes)

{

primes.Add(number);

}

}

});

Заменив языковую инструкцию цикла вызовом метода, автоматически обеспечивается параллельное выполнение итераций цикла. Кроме того, метод *Parallel.For* – это не простой цикл, генерирующий задачи в каждой итерации или для каждого фрагмента данных определенного размера. Вместо этого *Parallel.For* не спеша приспосабливается к темпу выполнения отдельных итераций, учитывая количество задач, выполняющихся в каждый момент, и исключает вероятность дробления диапазона на слишком мелкие фрагменты, производя его деление динамически.

Реализовать подобное поведение вручную весьма непросто, но вам доступны некоторые настройки (такие как управление максимальным количеством выполняемых задач), которые можно выполнить с помощью перегруженной версии метода *Parallel.For*, принимающей объект *ParallelOptions*, или используя собственный метод, выполняющий деление диапазона между задачами.

Существует похожий метод и для цикла *foreach*, который с успехом можно использовать, когда объем источника данных заранее неизвестен и может даже оказаться бесконечным. Представьте, что нам потребовалось загрузить из Интернета множество полос *RSS*, объявленных как *IEnumerable<string>.* В этом случае общая структура цикла могла бы иметь следующий вид:

IEnumerable<string> rssFeeds = // ...;

WebClient webClient = new WebClient();

foreach (string url in rssFeeds) {

Process(webClient.DownloadString(url));

}

Данный цикл легко можно распараллелить механической заменой инструкции foreach вызовом метода *Parallel.ForEach*. Обратите внимание, что источник данных (коллекция *rssFeeds*) необязательно должен быть потокобезопасным, потому что *Parallel.ForEach* автоматически синхронизирует операции обращения к нему из разных потоков:

IEnumerable<string> rssFeeds = // ... не должен быть потокобезопасным

WebClient webClient = new WebClient();

Parallel.ForEach(rssFeeds, url => {

Process(webClient.DownloadString(url));

});

Однако, распараллелить цикл совсем не просто, как могло бы показаться из предыдущего обсуждения. Существует ряд «недостающих» особенностей, которые необходимо рассмотреть, прежде чем подвесить этот инструмент на пояс. Для начинающих отметим, что в языке *C#* существует ключевое слово *break*, которое может вызывать преждевременное завершение циклов. Но как завершить цикл, параллельно выполняемый несколькими потоками, когда неизвестно, какая итерация в данный момент выполняется в других потоках?

Класс *ParallelLoopState* представляет состояние параллельного цикла и позволяет прервать его. Например:

int invitedToParty = 0;

Parallel.ForEach(customers, (customer, loopState) =>

{

if (customer.Orders.Count > 10 && customer.City == "Москва")

{

if (Interlocked.Increment(ref invitedToParty) >= 25)

{

// прервать выполнение итераций

loopState.Stop();

}

}

});

Обратите внимание: метод *Stop()* не гарантирует, что итерация, вызвавшая его, будет последней - итерации, уже запущенные к этому моменту, будут выполнены до конца (если они не проверяют свойство *ParallelLoopState.ShouldExitCurrentIteration*). Но гарантирует, что никакие другие итерации не будут запланированы на выполнение.

Одним из недостатков метода *ParallelLoopState.Stop()* является отсутствие гарантий, что все итерации, предшествующие данной, будут выполнены. Например, при обработке списка из 1000 заказчиков таким способом может получиться так, что заказчики с 1 по 100 будут обработаны полностью, заказчики с 101 по 110 вообще не будут обработаны, и заказчик 111 окажется последним обработанным перед вызовом *Stop().* Если необходимо обеспечить выполнение всех итераций, предшествующих данной (даже если они еще не были запущены!), следует использовать метод *ParallelLoopState.Break().*

***1.7.2*** Пожалуй, самой высокоуровневой абстракцией параллельных вычислений является возможность переложить все хлопоты по рапределенному выполнению на используемый фреймворк. Именно это позволяет фреймворк *Parallel LINQ*. Но сначала вспомним, что такое *LINQ*.

*LINQ* (*Language Integrated Query* - язык интегрированных запросов) - это фреймворк и набор расширений языка, появившийся в версии *C#* 3.0 и *.NET* 3.5, стирающий грань между императивным и декларативным программированием там, где требуется выполнять итерации через данные. Например, следующий запрос *LINQ* извлекает из источника *customers* - который может быть обычной коллекцией в памяти, таблицей в базе данных или иметь более экзотическое происхождение - имена и возраст клиентов, проживающих в Москве, сделавших не менее трех покупок на сумму более 100 руб. за последние десять месяцев, и выводит эти данные в консоль:

public class Customer

{

public int Id { get; set; }

public string City { get; set; }

public string Name { get; set; }

public byte Age { get; set; }

}

public class Order

{

public int Id { get; set; }

public DateTime Date { get; set; }

public int Amount { get; set; }

}

public class Program

{

static void Main(string[] args)

{

List<Customer> customers = new List<Customer>();

List<Order> orders = new List<Order>();

var results = from customer in customers

where customer.City == "Москва"…;

foreach (var result in results)

{

Console.WriteLine("{0} {1}", result.Name, result.Age);

}

}

}

Первое, на что следует обратить внимание, - большая часть запроса определена декларативно, подобно запросу на языке *SQL*. Здесь не используются циклы для фильтрации объектов или группировки объектов из разных источников. Часто вам не придется даже волноваться о синхронизации итераций, выполняемых запросом, потому что запросы *LINQ* являются исключительно функциональными и не имеют побочных эффектов - они преобразуют одну коллекцию (*IEnumerable<T>*) в другую, не изменяя никакие объекты в процессе работы.

Чтобы распараллелить запрос, представленный выше, достаточно лишь изменить тип коллекции источника с обобщенного *IEnumerable<T>* на *ParallelQuery<T>.* Для этого можно воспользоваться методом *AsParallel()* расширения и получить в результате следующий элегантный код:

var results = from customer in customers.AsParallel()…;

Параллельная обработка запросов выполняется фреймворком *PLINQ* в три этапа, как показано на рисунке ниже. Сначала *PLINQ* решает, сколько потоков следует использовать для выполнения запроса. Затем рабочие потоки извлекают свои фрагменты их исходной коллекции, под защитой блокировок. Все потоки выполняют свои задания независимо и помещают результаты в свои локальные очереди. В заключение, локальные результаты объединяются в единую коллекцию, которая подается в цикл foreach, в примере выше.

Серые прямоугольники представляют результаты завершенных заданий, помещенные в локальные очереди потоков, откуда они последовательно перемещаются в общий выходной буфер, доступный вызывающей программе. Заштрихованные прямоугольники представляют задания, выполняющиеся в данный момент.

Главное преимущество *PLINQ* перед методом *Parallel.ForEach()* заключается в автоматическом объединении результатов, полученных разными потоками. В примере поиска простых чисел с использованием *Parallel.ForEach* было необходимо вручную добавлять результаты работы каждого потока в глобальную коллекцию. При этом необходимо было использовать механизм синхронизации и тем самым увеличивать накладные расходы. Тот же результат легко можно получить с помощью *PLINQ*:

List<int> primes = (from n in Enumerable.Range(3, 100000).AsParallel()

where IsPrime(n)

select n).ToList();

Параллельные циклы (*Parallel.For* и *Parallel.ForEach*) и *PLINQ* поддерживают несколько методов для выполнения настройки, которые делают эти инструменты чрезвычайно гибкими и близкими в богатстве и выразительности к механизму параллельных задач, обсуждавшемуся выше. Методы параллельных циклов принимают объект *ParallelOptions* с различными свойствами, определяющими дополнительные параметры, а фреймворк *PLINQ* - дополнительные методы объектов *ParallelQuery<T>.* В число настраиваемых параметров входят:

* ограничение степени параллелизма (максимально возможное количество задач, выполняемых параллельно);
* передача признака отмены для остановки параллельных задач;
* принудительное определение порядка получения результатов параллельных запросов;
* управление буферизацией вывода (режимом слияния) результатов параллельных запросов.

При использовании параллельных циклов чаще всего ограничивают максимально возможное количество задач, выполняемых одновременно, тогда как при использовании *PLINQ* обычно настраивают режим слияния результатов и порядок их вывода.

***1.7.3*** Достаточно часто возникает необходимость связать с некоторой задачей продолжение (*continuation*), или обратный вызов (*callback*), то есть некоторый код, который должен быть выполнен по завершении задачи. Имея контроль над задачей, можно встроить обратный вызов в саму задачу, но когда задача получается от какого-то другого метода, необходимо использовать явный API продолжения.

Библиотека *TPL* предлагает метод экземпляра *ContinueWith* и статические методы *ContinueWhenAll()* и *ContinueWhenAny()* (их имена говорят сами за себя) для управления продолжениями в некоторых ситуациях. Используя класс *TaskScheduler* можно запланировать выполнение продолжения только при определенных обстоятельствах (например, только когда задача завершилась успешно или только когда в задаче возникло исключение) и в определенном потоке (группе потоков).

Продолжения – удобный способ программирования асинхронных приложений и очень ценный, при выполнении асинхронных операций ввода/вывода в приложениях с графическим интерфейсом. Например, чтобы обеспечить высокую отзывчивость приложений для *Windows* 8 с интерфейсом в стиле Метро (*Metro*), *WinRT* (*Windows* *Runtime*) *API* в *Windows* 8 поддерживает только асинхронные версии всех операций, длительность которых может составить больше 50 миллисекунд. При наличии множества асинхронных вызовов, выполняемых друг за другом, вложенные продолжения могут стать неудобными в использовании.

Асинхронный метод должен быть отмечен ключевым словом async и может возвращать значение типа *void*, *Task* или *Task<T>.* Внутри асинхронного метода можно использовать оператор await, чтобы реализовать продолжение без использования метода *ContinueWith().* Взгляните на следующий пример:

private async void updateButton\_Clicked()

{

using (LocationService location = new LocationService())

{

Task<Location> locTask = location.GetCurrentLocationAsync();

Location loc = await locTask;

cityTextBox.Text = loc.City.Name;

}

}

Здесь выражение "*await* *locTask*" реализует продолжение для задачи, возвращаемой вызовом *GetCurrentLocationAsync().* Собственно продолжением является оставшаяся часть тела метода (начиная с инструкции присваивания переменной *loc*), а значением выражения await является результат выполнения задачи, в данном случае - объект *Location*. Кроме того, продолжение неявно планируется для выполнения в потоке управления пользовательским интерфейсом, о чем, при использовании *TaskScheduler*, необходимо было позаботиться явно.

Заботу обо всех синтаксических особенностях тела метода берет на себя компилятор *C#*. Например, в только что написанном методе имеется блок *try...finally*, спрятанный под покровом инструкции using. Компилятор переделает продолжение так, что метод *Dispose()* переменной *location* будет вызван независимо от успешности завершения задачи. Эта особенность делает замену вызовов синхронных методов их асинхронными аналогами почти тривиальным делом. Компилятор поддерживает обработку исключений, сложные циклы, рекурсивные вызовы методов - языковые конструкции, которые плохо сочетаются с явным механизмом продолжений.

Всего лишь две языковые особенности (не очень сложные в реализации!) существенно снизили порог вхождения в асинхронное программирование и упростили работу с методами, возвращающими задачи и управляющими ими. Кроме того, реализация оператора *await* несовместима с библиотекой *Task Parallel Library;* низкоуровневый *WinRT* *API* в *Windows* 8 возвращает экземпляры типа *IAsyncOperation<T>,* а не задачи *Task* (которые являются управляемой концепцией), которые, тем не менее, с успехом можно передавать оператору *await*.

***1.7.4*** Первым приемом оптимизации, который может использоваться при распараллеливании циклов с общим состоянием, является агрегирование (иногда называется сверткой (*reduction*)). Когда в параллельном цикле используется общее состояние, масштабируемость часто утрачивается из-за необходимости синхронизировать доступ к общим данным; чем больше ядер в процессоре оказывается доступно, тем меньше выигрыш из-за синхронизации (этот эффект является прямым следствием закона Амдала (*Amdahl Law*), который часто называют законом убывающей отдачи (*The Law of Diminishing Returns*)). Значительный прирост производительности часто достигается за счет создания локальных состояний потоков или задач, выполняющих параллельные итерации цикла, и их объединения в конце. Методы из библиотеки *TPL*, используемые для организации циклов, имеют перегруженные версии, обслуживающие такого рода локальные агрегаты.

Одной из основных помех масштабированию в нем является необходимость добавления новых обнаруженных простых чисел в совместно используемый список, для чего требуется использовать механизм синхронизации. Вместо этого можно использовать в каждом потоке свой, локальный список и объединить их по завершении цикла:

public class Program

{

static void Main(string[] args)

{

IEnumerable<int> counters = new List<int>();

Stopwatch timer = Stopwatch.StartNew();

counters = PrimesInRange(100, 200000);

timer.Stop();

Console.WriteLine("Прошло: {0} мс., найдено {1} простых чисел",

timer.ElapsedMilliseconds,

counters.Count());

}

public static IEnumerable<int> PrimesInRange(uint start, uint end)

{

List<int> primes = new List<int>();

Parallel.For(start, end,

() => new List<int>(),

(i, pls, localPrimes) => /

{

if (IsPrime(i))

{

localPrimes.Add((int)i);

}

return localPrimes;

},

localPrimes =>

{

lock (primes)

{

primes.AddRange(localPrimes);

}

}

);

return primes;

}

private static bool IsPrime(long number)

{

…

}

}

В примере выше количество попыток приобрести блокировку значительно меньше, чем в предыдущих примерах - блокировку требуется приобрести лишь один раз для каждого потока, а не для каждого найденного простого числа. Добавляются накладные расходы на объединение списков, но эта цена незначительна, в сравнении с увеличившейся масштабируемостью.

Еще одно место, где можно применить оптимизацию, - итерации цикла, слишком короткие, чтобы их эффективно можно было распараллелить. Даже при том, что механизм параллелизма данных объединяет несколько итераций иногда тело цикла может выполняться настолько быстро, по скорости превосходят вызов делегата, необходимый для вызова тела цикла в каждой итерации. В этом случае можно использовать класс Partitioner и с его помощью вручную группировать итерации, уменьшая количество вызовов делегатов:

Parallel.For(

Partitioner.Create(start, end),

range => { // range - это тип Tuple<int, int>

for (int i = range.Item1; i < range.Item2; ++i)

{

// тело цикла без вызова делегата

}

});

За дополнительной информацией о разбиении циклов на фрагменты, являющемся важным способом оптимизации, обращайтесь к статье «Пользовательские разделители для *PLINQ* и *TPL*» на сайте *MSDN*.

Наконец, существуют приложения, в которых могут пригодиться собственные планировщики задач. В качестве примеров можно привести планирование заданий в потоке управления пользовательским интерфейсом, назначение приоритетов задачам, планируя их с помощью высокоприоритетного планировщика, и связывание задач с определенным процессором, планируя их с помощью планировщика, использующего потоки, привязанные к определенному процессору. Реализовать собственные планировщики можно путем наследования класса *TaskScheduler*. Пример такой реализации можно найти в статье «Практическое руководство. Создание планировщика заданий, ограничивающего степень параллелизма» на сайте *MSDN*.

**2 АЛГОРИТМИЧЕСКИЙ АНАЛИЗ**

**2.1 Постановка задачи**

Необходимо провести анализ заданной поверхности в математическом пакете. Анализ включает: построение графика поверхности, вычисления площади поверхности, используя аналитическую формулу

(12)

Вид поверхности:

(13)

Метод численного интегрирования – метод трапеций. Значение коэффициентов и пределы интегрирования представлены в таблице 2.1.

Таблица 2.1 – Значения коэффициентов и пределы интегрирования

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Значения коэффициентов** | | | | **Пределы интегрирования** | | | |
| *A* | *B* | *C* | *D* | *Xнач* | *Xкон* | *Yнач* | *Yкон* |
| -3 | -1 | -5 | 5 | -100 | 100 | -20 | 20 |

Необходимо провести численный анализ заданной поверхности, используя формулы численного интегрирования. Программный код оформить в виде компонента, позволяющий принимать в качестве параметров границы области интегрирования, вычислять площадь заданной поверхности в пределах области интегрирования, готовить данные для построения графика в математическом пакете.

Также необходимо провести анализ скорости обработки заданной поверхности и погрешности численных методов в зависимости от количества потоков, используемых для расчетов.

**2.2 Построение функциональной модели**

Для выполнения поставленной задачи, программа должна реализовывать следующий функционал:

* расчет площади заданной поверхности заданным численным методом (методом трапеций);
* построение графика поверхности;
* анализ влияния распараллеливания вычисления площади поверхности заданным численным методом.

Структурная схема программы представлена на рисунке 2.1.

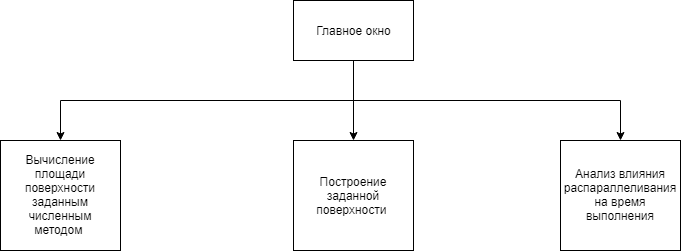


Рисунок 2.1 – Структурная схема программы

Приложение представляет собой комплекс классов для расчета площади, отображения поверхности и анализа влияния распараллеливания подсчета площади. При этом под анализом будет пониматься наглядное сравнение погрешности вычисления и затраченного времени на выполнение задачи параллельным и линейным методами.

**3 РАЗРАБОТКА ПРИЛОЖЕНИЯ**

**3.1 Разработка программного кода**

Приложение можно условно поделить на 2 части:

* программа на языке *Python* для отображения поверхности;
* проект *C#* для вычисления площади и проведения анализа;
* вывод статистики в *Excel*.

В файле *CourseProj.py* описан процесс создания, отображения поверхности по данным, полученным из проекта *C#.*

Список функций и их описание отображено в таблице 3.1.

Таблица 3.1 – Функции файла *CourseProj.py*

|  |  |
| --- | --- |
| **Функция** | **Описание** |
| *makeData* | Построение поверхности |

Функция *makeData* необходима для построения поверхности по данным, полученным из C# проекта и не принимает параметров.

В классе *CheckData.cs* описан алгоритм проверки введенных на форму данных. Список методов и их описание отображено в таблице 3.2.

Таблица 3.2 – Методы файла *CheckData.cs*

|  |  |
| --- | --- |
| **Функция** | **Описание** |
| *CheckAsync* | Проверка введенных данных |
| *CheckData* | Конструктор класса |

Функция *public async System.Threading.Tasks.Task CheckAsync()* необходима для проверки входных данных и не принимает параметров.

Конструктор *public CheckData(MainForm mainForm)* необходим для создания поля главной формы и принимает параметры:

* *MainForm mainForm* – главное окно.

В классе *ExportToExcel.cs* описан алгоритм создания статистики и экспорт ее в *Excel*. Методы класса описаны в таблице 3.3.

Таблица 3.3 – Методы класса *ExportToExcel.cs*

|  |  |
| --- | --- |
| **Функция** | **Описание** |
| *ExportToExcel* | Конструктор класса |
| *Function* | Задание функции |
| *Export* | Экспорт данных |

Конструктор *public CheckData(MainForm mainForm)* необходим для создания поля главной формы и принимает параметры:

* *MainForm mainForm* – главное окно.

Функция *public double Function(double x, double y)* используется для построения уравнения поверхности. Принимает параметры:

* *double x* – значение аргумента;
* *double y* – значение функции при заданном аргументе.

Функция *public void Export()* необходима для отправки данных в *Excel* и не принимает параметров.

В классе *Processing.cs* описан алгоритм создания статистики и экспорт ее в *Excel*. Методы класса описаны в таблице 3.4.

Таблица 3.4 – Методы класса *Processing.cs*

|  |  |
| --- | --- |
| **Функция** | **Описание** |
| *Processing* | Конструктор класса |
| *Function* | Задание функции |
| *DerivativeFunction* | Задание функции в аналитическом виде |
| *Integrate* | Линейный метод подсчета площади для исходной функции |
| *ParallelIntegrate* | Распараллеленый метод подсчета площади для исходной функции |
| *IntegrateDerivative* | Линейный метод подсчета площади для функции в аналитическом виде |
| *ParallelIntegrateDerivative* | Распараллеленый метод подсчета площади для функции в аналитическом виде |

Конструктор *public Processing (int a, int b, int c, int d, MainForm mainForm)* инициализирует класс и принимает параметры:

* *int a* – коэффициент *A*;
* *int b* – коэффициент *B*;
* *int c* – коэффициент *C*;
* *int d* – коэффициент *D*;
* *MainForm mainForm* – главное окно.

Функция *public double Function(double x, double y)* используется для создания уравнения поверхности. Принимает параметры:

* *double x* – значение аргумента;
* *double y* – значение функции при заданном аргументе.

Функция *public void DerivativeFunction(double x, double y)* необходима для создания функции в аналитическом виде. Принимает параметры:

* *double x* – значение аргумента;
* *double y* – значение функции при заданном аргументе.

Функция *double Integrate (double xn, double xk, double yn, double, yk, double n)* − функция, которая линейно рассчитывает площадь поверхности. Принимает параметры:

* *xn* − начальное значение *x*;
* *xk* − конечное значение *х*;
* *yn* − начальное значение *у*;
* *yk* − конечное значение *у*;
* *h1* − количество точек для расчета *x*;
* *h2* − количество точек для расчета *y*.

Функция *double IntegrateDerivative (double xn, double xk, double yn, double, yk, double n)* − функция, которая линейно рассчитывает площадь поверхности по аналитической формуле. Принимает параметры:

* *xn* − начальное значение *x*;
* *xk* − конечное значение *х*;
* *yn* − начальное значение *у*;
* *yk* − конечное значение *у*;
* *h1* − количество точек для расчета *x*;
* *h2* − количество точек для расчета *y*.

Функция *double ParallelIntegrate (double xn, double xk, double yn, double, yk, double n)* − функция, которая распределенно рассчитывает площадь поверхности. Принимает параметры:

* *xn* − начальное значение *x*;
* *xk* − конечное значение *х*;
* *yn* − начальное значение *у*;
* *yk* − конечное значение *у*;
* *h1* − количество точек для расчета *x*;
* *h2* − количество точек для расчета *y*.

Функция *double ParallelIntegrateDerivative (double xn, double xk, double yn, double, yk, double n)* − функция, которая распределенно рассчитывает площадь поверхности по аналитической формуле. Принимает параметры:

* *xn* − начальное значение *x*;
* *xk* − конечное значение *х*;
* *yn* − начальное значение *у*;
* *yk* − конечное значение *у*;
* *h1* − количество точек для расчета *x*;
* *h2* − количество точек для расчета *y*.

В классе *MainForm.cs* описаны в основном обработчики событий для каких-либо действий. Методы класса описаны в таблице 3.5.

Таблица 3.5 – Методы класса *MainForm.cs*

|  |  |
| --- | --- |
| **Функция** | **Описание** |
| *MainForm* | Конструктор класса |
| *ProcessButton*\_*Click* | Нажатие на кнопку «обработать» |
| *ExitButton*\_*Click* | Нажатие на кнопку «выход» |
| *AValue*\_*Click* | Нажатие на поле с коэффициентом A |
| *BValue\_Click* | Нажатие на поле с коэффициентом B |
| *CValue\_Click* | Нажатие на поле с коэффициентом C |
| *DValue\_Click* | Нажатие на поле с коэффициентом D |
| *ExportToExcelBtn\_Click* | Нажатие на кнопку «статистика» |
| *ShowSurfaceBtn\_Click* | Нажатие на кнопку «построить поверхность» |

Конструктор *public MainForm ()* инициализирует форму и не принимает параметров.

Метод *private void ProcessButton\_Click (object sender, EventArgs e)* обрабатывает нажатие на кнопку обработки. В результате нажатия создается объект класса *CheckData* и происходит проверка введенных данных. Если проверка успешна, производится вычисление.

Метод *private void ExitButton\_Click (object sender, EventArgs e)* обрабатывает нажатие на кнопку выхода. Приложение закрывается при нажатии.

Метод *private void AValue\_Click (object sender, EventArgs e)* обрабатывает нажатие на поле с коэффициентом *A*. Поле очищается при нажатии.

Метод *private void BValue\_Click (object sender, EventArgs e)* обрабатывает нажатие на поле с коэффициентом *B*. Поле очищается при нажатии.

Метод *private void CValue\_Click (object sender, EventArgs e)* обрабатывает нажатие на поле с коэффициентом *C*. Поле очищается при нажатии.

Метод *private void DValue\_Click (object sender, EventArgs e)* обрабатывает нажатие на поле с коэффициентом *D*. Поле очищается при нажатии.

Метод *private void ExportToExcelBtn\_Click (object sender, EventArgs e)* обрабатывает нажатие на кнопку экспорта. В результате нажатия создается объект класса *ExportToExcel* и происходит экспорт данных.

Метод *private void ShowSurfaceBtn\_Click (object sender, EventArgs e)* обрабатывает нажатие на кнопку отображения. В результате нажатия происходит проверка введенных данных. Если проверка успешна, производится построения.

**3.2 Разработка графического интерфейса**

Программа состоит из одного графического окна. В данном окне имеется ряд полей ввода информации, а именно – поле выбора количества потоков, и поля ввода коэффициентов Также присутствуют элементы управления программой в виде кнопок – кнопки для построения поверхности, вычисления площади, создания статистики и выхода.

Также присутствуют поля для отображения результатов.

Главная форма представлена на рисунке 3.1. Представленная поверхность – на рисунке 3.2. Результат выполнения программы представлен на рисунке 3.3.

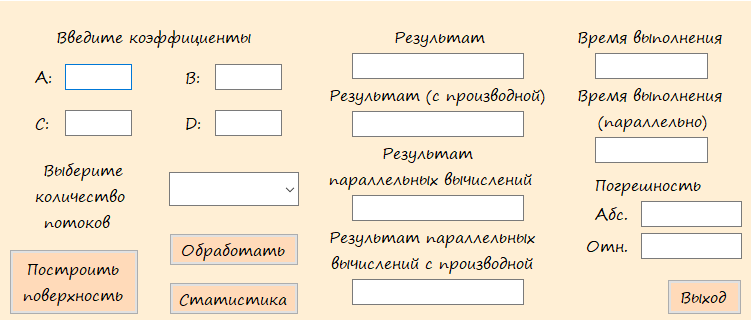


Рисунок 3.1 – Главное окно приложения

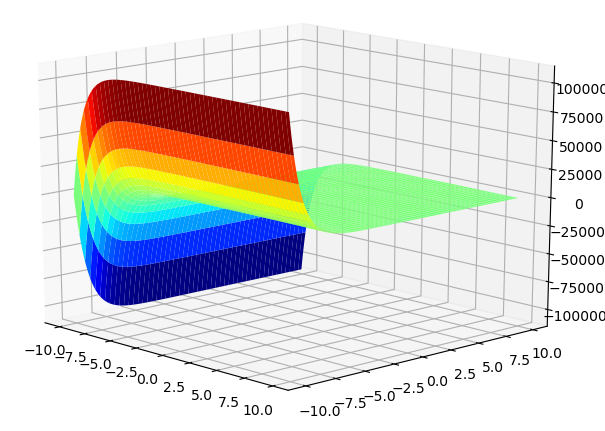


Рисунок 3.2 – Построенная поверхность

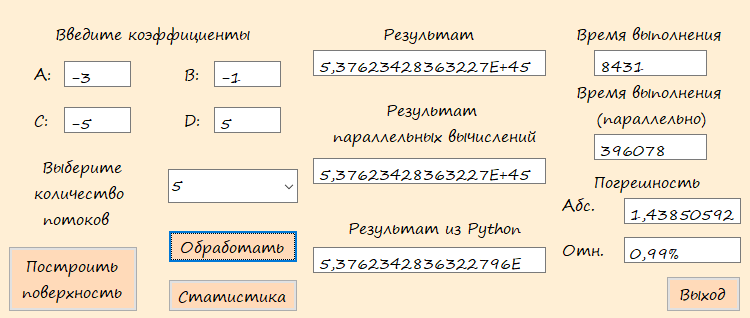


Рисунок 3.3 – Результат выполнения программы

**3.3 Определение погрешности численного метода**

Абсолютная погрешность некоторого числа равна разности между его истинным значением и приближенным значением, полученным в результате вычисления или измерения:

           (13)

где *а* – приближенное значение числа *х.*

Относительная погрешность – это отношение абсолютной погрешности к приближенному значению числа:

           (14)

Для определения погрешности вычислений численного метода интегрирования (метод трапеций), были вычислены значения площади поверхности по аналитической формуле 12, используя функцию *dblquad* из модуля *scipy.integrate* в *Python*.

Общая форма *dblquad* – это *scipy.integrate.dblquad (func, a, b, gfun, hfun).* Где *func* – это имя интегрируемой функции, *a* и *b* – это нижний и верхний пределы переменной *x* соответственно, а *gfun* и *hfun* – это имена функций, которые определяют нижний и верхний пределы. переменной *у*.

В качестве примера приведем метод двойного интеграла.

(15)

(16)

(17)

Функции *f* (15), *g* (17) и *h* (16) определяются, используя лямбда-выражения. Даже если *g* и *h* являются константами, как они могут быть во многих случаях, они должны быть определены как функции, как сделано здесь для нижнего предела.

import scipy.integrate

from numpy import exp

from math import sqrt

f = lambda x, y : 16\*x\*y

g = lambda x : 0

h = lambda y : sqrt(1-4\*y\*\*2)

i = scipy.integrate.dblquad(f, 0, 0.5, g, h)

print (i)

Вышеуказанная программа сгенерирует следующий вывод.

(0.5, 1.7092350012594845e-14)

Используя формулы 13 и 14, было проведено определение погрешностей, представленное на рисунке 3.3.

****

Рисунок 3.4 – Вычисление погрешности

**4 АНАЛИЗ РЕЗУЛЬТАТОВ РАБОТЫ И ВЫВОДЫ**

В качестве используемого оборудования выступает компьютер со следующими характеристиками:

* процессор *Intel(R) Core(TM) i5-4288U CPU @ 2.60GHz 2.61GHz;*
* ОС *Windows* 10;
* 4.00 ГБ ОЗУ.

Процессор имеет 4 ядра, что позволит использовать распределенные вычисления.

Для реализации данной задачи были использованы стандартные механизмы работы с многопоточностью в C#, а именно Parallel.For.

Метод *Parallel.For* позволяет выполнять итерации цикла параллельно. Он имеет следующее определение: *For(int, int, Action<int>),* где первый параметр задает начальный индекс элемента в цикле, а второй параметр - конечный индекс. Третий параметр – делегат *Action* – указывает на метод, который будет выполняться один раз за итерацию.

В ходе анализа выполнения вычислений линейным методом трапеций и распределенным методом трапеций была получена статистика, анализируя которую, можно сказать, что с увеличением числа потоков время вычисления уменьшается. Сравнивая время линейного и распределенного методов, можно сказать, что при малых объемах данных распределенный вариант занимает больше времени за счет выполнения алгоритма распараллеливания.

Далее приведена подробная статистическая информация.

Рассмотрим зависимость скорости выполнения вычислений от количества потоков (таблица 4.1).

Таблица 4.1 – Время выполнения вычислений

|  |  |
| --- | --- |
| **Количество потоков** | **Время выполнения (10-7 мс)** |
| 1 | 13 |
| 2 | 8 |
| 3 | 7 |
| 4 | 5 |
| 5 | 5 |
| 6 | 4 |
| 7 | 5 |
| 8 | 4 |
| 9 | 5 |
| 10 | 5 |
| 11 | 6 |
| 12 | 6 |
| 13 | 7 |
| 14 | 5 |
| 15 | 6 |

Таблица получена путем запуска функции с различным количеством потоков. Данные о количестве и времени записываются в коллекцию *Dictionary* и передаются в *Excel*.

Проанализировав данные в таблице, можно заметить, что с увеличением количества потоков уменьшается время выполнения вычислений.

Далее в *Excel* строится график зависимости, расположенный на рисунке 4.1, наглядно иллюстрирует это.

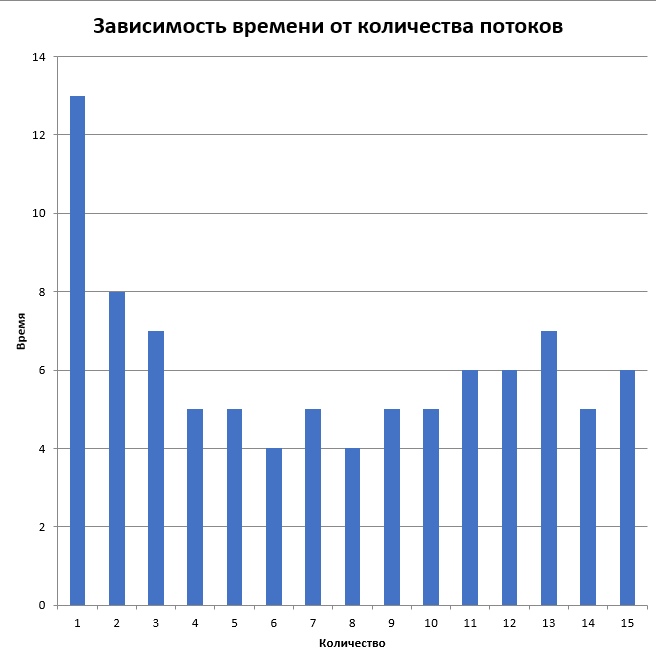


Рисунок 4.1 – Зависимость скорости вычислений от количества потоков

**5 ТЕСТИРОВАНИЕ И ВЕРИФИКАЦИЯ**

**5.1 Тестирование пользовательского интерфейса**

Весь необходимый функционал приложения был реализован с помощью графических элементов. Все действия сопровождаются соответствующими уведомлениями.

Для каждого уведомления был подобран соответствующий дизайн: шрифт, цвет и содержание. Для приложения были подобраны стили, которые делают приложение более привлекательным, но не мешают читабельности.

Приложение имеет удобный интерфейс. Все элементы выполнены в одном стиле.

Отсутствуют орфографические и пунктуационные ошибки.

При запуске программы открывается окно, которое представлено на рисунке 5.1:

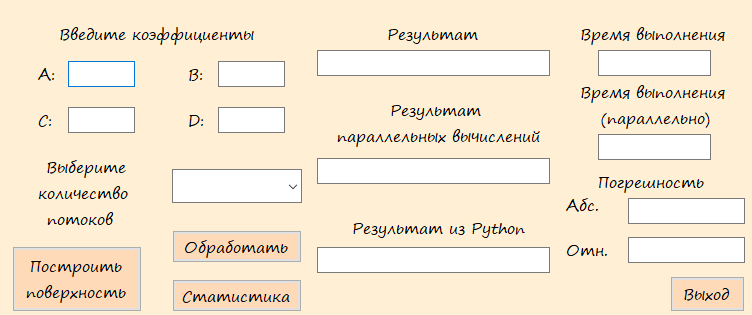


Рисунок 5.1 – Главное окно программы

В этом окне пользователь может ввести количество потоков для распределенного вычисления, коэффициенты для функции поверхности, которая будет построена.

После ввода коэффициентов при нажатии на кнопку «Построить поверхность», запускается файл Python, строящий поверхность по введенным данным.

Если возникают исключения (например, не введены коэффициенты), пользователь получает уведомление об этом.

Если коэффициенты получены успешно, открывается графическое окно (рисунок 5.2).

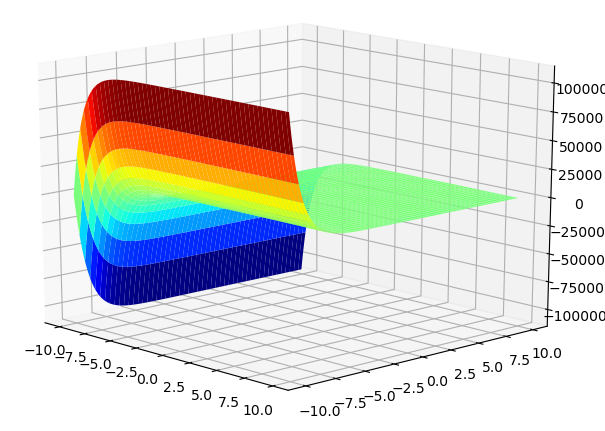


Рисунок 5.2 – Отображение поверхности

При вводе данных, прошедших валидацию, пользователь получает результаты, полученные четырьмя методами, а также погрешность вычислений.

Результат представлен на рисунке 5.4.

Если по случайности пользователь вводит неверные значения в поля ввода количества потоков или коэффициентов, он получает уведомление об этом с кнопкой «Повтор», при нажатии на которую пользователь может вновь ввести желаемые значения.

Пример уведомления представлен на рисунке 5.3.

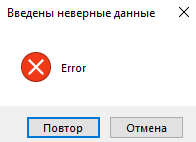


Рисунок 5.3 – Уведомление о вводе неверных данных

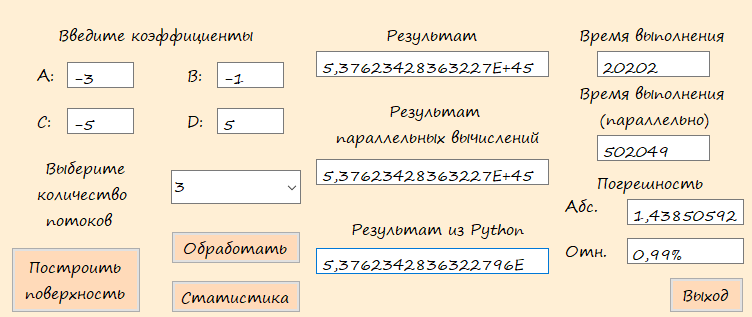


Рисунок 5.4 – Результат выполнения программы

Основные тесты интерфейса представлены в таблице 5.1.

Таблица 5.1 – Тестирование пользовательского интерфейса

|  |  |  |
| --- | --- | --- |
| **№** | **Вид проверки** | **Результат** |
| 1 | Реализуется ли функционал приложения при помощи графических элементов | 10 |
| 2 | Реализуется ли размещение сообщений об ошибках | 10 |
| 3 | Отсутствуют ли орфографически, пунктуационные ошибки | 10 |
| 4 | Читабелен ли используемый шрифт | 9 |

**5.2 Тестирование функционала**

Результат тестирования функционала программы представлен в таблице 5.2.

Таблица 5.2 – Тестирование функционала

|  |  |  |  |
| --- | --- | --- | --- |
| **№** | **Имя элемента** | **Тест** | **Результат** |
| 1 | *IntegrateDerivative* | Линейный метод подсчета площади для исходной функции | Исключительных ситуаций нет |
| 2 | *ParallelIntegrateDerivative* | Распараллеленый метод подсчета площади для исходной функции | Исключительных ситуаций нет |

**ЗАКЛЮЧЕНИЕ**

В результате выполнения данного курсового проекта было разработано приложение, предназначенное для вычисления площади заданной поверхности в заданных пределах, построен график данной поверхности и исследование влияния распределения вычисления площади на время выполнения и точность результатов. Были проведены такие этапы разработки, как изучение теоретического материала, алгоритмический анализ задачи и составление алгоритма работы приложения.

В течение выполнения данного курсового проекта был изучен теоретический материал о численном методе трапеций для вычисления интеграла, на основе которого в дальнейшем были разработаны алгоритмы решения поставленной задачи.

Поставленная задача решена на языке программирования *C#* в среде разработки *Microsoft Visual Studio Community* 2019.

При использовании приложения пользователь может вычислить площадь данной по варианту поверхности, построить график этой поверхности и провести анализ влияния распределения вычисления площади поверхности на время выполнения. В итоге был приобретен опыт разработки подобных приложений с помощью языка программирования *C#* и *Python*.

Получены характеристики скорости вычисления, на основании которого сделан следующий вывод об эффективности распараллеливания.

Поставленные задачи в курсовом проекте решены в полном объеме.

**Список использованных источников**

1. Ильин В. А., Позняк Э. Г. Аналитическая геометрия. — М.: ФИЗМАТЛИТ, 2002. — 240 с.

2. Калиткин, Н.Н. Численные методы / Н.Н. Калиткин. – СПб.:BHУ, 2014. – 202 с.

3. Основные понятия. Численные методы линейной алгебры и анализа [Электронные ресурс].: Свободная энциклопедия. – Электронные данные. – Режим доступа: https://slemeshevsky.github.io/python-num-pde/term1/build/html/\_computing-integrals/double-triple.html – Дата доступа: 29.04.2020.

4. Рихтер. Программирование на платформе Microsoft .NET/ В.Л.Григо-рьев. – М.: Энергоатомиздат, 2010. – 209 с.

5. Бизли. Python. Подробный справочник / В.И.Юров. – Спб: Питер, 2014.– 624 с.

**Приложение А**

**(обязательное)**

**CheckData.cs**

using System;

using System.Diagnostics;

using System.IO;

using System.Windows.Forms;

namespace CourseProject2.\_0

{

public class CheckData

{

public MainForm MainForm { get; set; }

public CheckData(MainForm mainForm)

{

MainForm = mainForm;

}

public async System.Threading.Tasks.Task CheckAsync()

{

try

{

int a = Convert.ToInt32(MainForm.aValue.Text);

int b = Convert.ToInt32(MainForm.bValue.Text);

int c = Convert.ToInt32(MainForm.cValue.Text);

int d = Convert.ToInt32(MainForm.dValue.Text);

Processing processing = new Processing(a, b, c, d, MainForm);

Stopwatch.StartNew();

long startTick = Stopwatch.GetTimestamp();

var squarePar = processing.ParallelIntegrate(-100, 100, 200, -20, 20, 40);

long stopTick = Stopwatch.GetTimestamp();

MainForm.timeParTextBox.Text = (stopTick - startTick).ToString();

MainForm.resultParTextBox.Text = squarePar.ToString();

startTick = Stopwatch.GetTimestamp();

var square = processing.Integrate(-100, 100, 200, -20, 20, 40);

stopTick = Stopwatch.GetTimestamp();

MainForm.timeTextBox.Text = (stopTick - startTick).ToString();

MainForm.resultTextBox.Text = square.ToString();

var square2 = processing.IntegrateDerivative(-100, 100, 200, -20, 20, 40);

MainForm.result2TextBox.Text = square2.ToString();

var square2Par = processing.ParallelIntegrateDetivate(-100, 100, 200, -20, 20, 40);

MainForm.result2ParTextBox.Text = square2Par.ToString();

MainForm.absoluteErrorTextBox.Text = Math.Abs(square2 - square).ToString();

MainForm.relativeErrorTextBox.Text = ((Math.Abs(square2 - square) / square) \* 100).ToString() + "%";

}

catch (FormatException)

{

MessageBox.Show("Error", "Введены неверные данные", MessageBoxButtons.RetryCancel, MessageBoxIcon.Error);

}

catch (InvalidCastException)

{

MessageBox.Show("Error", "Ошибка преобразования типов", MessageBoxButtons.RetryCancel, MessageBoxIcon.Error);

}

catch (ArgumentOutOfRangeException)

{

MessageBox.Show("Error", "Выход за пределы допустимых значений", MessageBoxButtons.RetryCancel, MessageBoxIcon.Error);

}

catch (NullReferenceException)

{

MessageBox.Show("Error", "Введена пустая строка", MessageBoxButtons.RetryCancel, MessageBoxIcon.Error);

}

}

}

}

**Processing.cs**

using System;

using System.IO;

using System.Threading;

using System.Threading.Tasks;

namespace CourseProject2.\_0

{

public class Processing

{

public int A { get; set; }

public int B { get; set; }

public int C { get; set; }

public int D { get; set; }

public MainForm MainForm { get; set; }

public Processing(int a, int b, int c, int d, MainForm mainForm)

{

A = a;

B = b;

C = c;

D = d;

MainForm = mainForm;

}

public virtual double DetivativeFunction(double x, double y)

{

return Math.Sqrt(1 +

Math.Pow(2 \* A \* x + C \* Math.Exp(-x), 2) +

Math.Pow(2 \* B \* y + D \* Math.Exp(-y), 2));

}

public double IntegrateDerivative(int xMin, int xMax, int nX, int yMin, int yMax, int nY)

{

int xIter = (int)((xMax - xMin) / nX);

int yIter = (int)((yMax - yMin) / nY);

double x, y, w, sum = 0;

for (var i = 0; i <= xIter; i++)

{

x = xMin + i \* nX;

for (var j = 0; j <= yIter; j++)

{

if (i > 0 && i < xIter && j > 0 && j < yIter)

{

w = 1;

}

else if ((i == 0 || i == yIter) && (j == 0 || j == xIter))

{

w = 0.25;

}

else

{

w = 0.5;

}

y = yMin + j \* nY;

sum += w \* DetivativeFunction(x, y);

}

}

return Math.Abs(nX \* nY \* sum);

}

public double ParallelIntegrateDetivate(int xMin, int xMax, int nX, int yMin, int yMax, int nY)

{

object lockObject = new object();

string filePath = string.Empty;

int xIter = (int)((xMax - xMin) / nX);

int yIter = (int)((yMax - yMin) / nY);

double x, y, w, sum = 0;

//ThreadPool.SetMinThreads(Convert.ToInt32(MainForm.threadCount.SelectedItem.ToString()), Convert.ToInt32(MainForm.threadCount.SelectedItem.ToString()));

Parallel.For(0, xIter, i =>

{

x = xMin + i \* nX;

for (var j = 0; j <= yIter; j++)

{

if (i > 0 && i < xIter && j > 0 && j < yIter)

{

w = 1;

}

else if ((i == 0 || i == yIter) && (j == 0 || j == xIter))

{

w = 0.25;

}

else

{

w = 0.5;

}

y = yMin + j \* nY;

sum += w \* DetivativeFunction(x, y);

}

lock (lockObject)

{

StreamWriter streamWriter = new StreamWriter("Thread" + i + ".txt");

streamWriter.WriteLine(Math.Abs(sum));

streamWriter.Dispose();

}

});

return Math.Abs(nX \* nY \* sum);

}

}

}

**MainForm.cs**

using System;

using System.Diagnostics;

using System.IO;

using System.Windows.Forms;

namespace CourseProject2.\_0

{

public partial class MainForm : Form

{

public MainForm()

{

InitializeComponent();

}

private void ProcessButton\_Click(object sender, EventArgs e)

{

CheckData checkData = new CheckData(this);

checkData.CheckAsync();

}

private void ExitButton\_Click(object sender, EventArgs e)

{

Close();

}

private void AValue\_Click(object sender, EventArgs e)

{

aValue.Clear();

}

private void BValue\_Click(object sender, EventArgs e)

{

bValue.Clear();

}

private void CValue\_Click(object sender, EventArgs e)

{

cValue.Clear();

}

private void DValue\_Click(object sender, EventArgs e)

{

dValue.Clear();

}

private void ExportToExcelBtn\_Click(object sender, EventArgs e)

{

ExportToExcel exportToExcel = new ExportToExcel(this);

exportToExcel.Export();

}

private void ShowSurfaceBtn\_Click(object sender, EventArgs e)

{

try

{

int a = Convert.ToInt32(aValue.Text);

int b = Convert.ToInt32(bValue.Text);

int c = Convert.ToInt32(cValue.Text);

int d = Convert.ToInt32(dValue.Text);

Process pythonProcess = new Process();

string fileName = "course.txt";

using (FileStream fstream = new FileStream(fileName, FileMode.OpenOrCreate))

{

byte[] array = System.Text.Encoding.Default.GetBytes(a.ToString() + "\n" + b.ToString() + "\n" +

c.ToString() + "\n" + d.ToString());

fstream.WriteAsync(array, 0, array.Length);

}

pythonProcess.StartInfo.FileName = Application.StartupPath + @"\dist\CourseProj.exe";

pythonProcess.Start();

}

catch (FormatException)

{

MessageBox.Show("Error", "Введены неверные данные", MessageBoxButtons.RetryCancel, MessageBoxIcon.Error);

}

catch (InvalidCastException)

{

MessageBox.Show("Error", "Ошибка преобразования типов", MessageBoxButtons.RetryCancel, MessageBoxIcon.Error);

}

catch (ArgumentOutOfRangeException)

{

MessageBox.Show("Error", "Выход за пределы допустимых значений", MessageBoxButtons.RetryCancel, MessageBoxIcon.Error);

}

catch (NullReferenceException)

{

MessageBox.Show("Error", "Введена пустая строка", MessageBoxButtons.RetryCancel, MessageBoxIcon.Error);

}

}

}

}

**ExportToExcel.cs**

using Microsoft.Office.Interop.Excel;

using System;

using System.Collections.Generic;

using System.Diagnostics;

using System.Threading.Tasks;

namespace CourseProject2.\_0

{

public class ExportToExcel

{

public MainForm MainForm { get; set; }

public ExportToExcel(MainForm mainForm)

{

MainForm = mainForm;

}

public double Funcion(double x, double y)

{

return Math.Sqrt(1 +

Math.Pow(2 \* (Convert.ToInt32(MainForm.aValue.Text)) \* x + (Convert.ToInt32(MainForm.cValue.Text)) \* Math.Exp(-x), 2) +

Math.Pow(2 \* (Convert.ToInt32(MainForm.bValue.Text)) \* y + Convert.ToInt32(MainForm.dValue.Text) \* Math.Exp(-y), 2));

}

public void Export()

{

Dictionary<int, int> statistic = new Dictionary<int, int>();

int xMin = -100, xMax = 100, nX = -20, yMin = 20, yMax = 200, nY = 40;

for (var i = 1; i <= 15; i ++)

{

//ThreadPool.SetMinThreads(i, i);

int xIter = (int)((xMax - xMin) / nX);

int yIter = (int)((yMax - yMin) / nY);

double x, y, w, sum = 0;

var stopwatch = new Stopwatch();

stopwatch.Start();

Parallel.For(0, xIter, k =>

{

x = xMin + k \* nX;

for (var j = 0; j <= yIter; j++)

{

if (k > 0 && k < xIter && j > 0 && j < yIter)

{

w = 1;

}

else if ((k == 0 || k == yIter) && (j == 0 || j == xIter))

{

w = 0.25;

}

else

{

w = 0.5;

}

y = yMin + j \* nY;

sum += w \* Funcion(x, y);

}

});

stopwatch.Stop();

statistic.Add(i, Convert.ToInt32(stopwatch.ElapsedTicks));

stopwatch.Reset();

}

Application ObjExcel = new Application();

Workbook ObjWorkBook;

Worksheet ObjWorkSheet;

ObjWorkBook = ObjExcel.Workbooks.Add(System.Reflection.Missing.Value);

ObjWorkSheet = (Worksheet)ObjWorkBook.Sheets[1];

foreach (var singleResult in statistic)

{

ObjWorkSheet.Cells[1, singleResult.Key] = singleResult.Key;

ObjWorkSheet.Cells[2, singleResult.Key] = singleResult.Value;

}

ObjExcel.Visible = true;

ObjExcel.UserControl = true;

var excelcells = ObjWorkSheet.get\_Range("A2", "O2");

excelcells.Select();

Chart excelchart = (Chart)ObjExcel.Charts.Add(Type.Missing, Type.Missing, Type.Missing, Type.Missing);

excelchart.Activate();

excelchart.Select(Type.Missing);

ObjExcel.ActiveChart.HasTitle = true;

ObjExcel.ActiveChart.HasLegend = false;

ObjExcel.ActiveChart.ChartTitle.Text

= "Зависимость времени от количества потоков";

((Axis)ObjExcel.ActiveChart.Axes(XlAxisType.xlCategory,

XlAxisGroup.xlPrimary)).HasTitle = true;

((Axis)ObjExcel.ActiveChart.Axes(XlAxisType.xlCategory,

XlAxisGroup.xlPrimary)).AxisTitle.Text = "Количество ";

((Axis)ObjExcel.ActiveChart.Axes(XlAxisType.xlValue,

XlAxisGroup.xlPrimary)).HasTitle = true;

((Axis)ObjExcel.ActiveChart.Axes(XlAxisType.xlValue,

XlAxisGroup.xlPrimary)).AxisTitle.Text = "Время";

ObjExcel.ActiveChart.Location(XlChartLocation.xlLocationAsObject, "Лист1");

var excelsheets = ObjWorkBook.Worksheets;

ObjWorkSheet = (Worksheet)excelsheets.get\_Item(1);

ObjWorkSheet.Shapes.Item(1).IncrementLeft(-201);

ObjWorkSheet.Shapes.Item(1).IncrementTop((float)20.5);

ObjWorkSheet.Shapes.Item(1).Height = 500;

ObjWorkSheet.Shapes.Item(1).Width = 500;

}

}

}

**CourseProj.py**

import pylab

from mpl\_toolkits.mplot3d import Axes3D

import numpy

from matplotlib import cm

import sys

import argparse

args = []

f = open("course.txt")

args = [line.strip() for line in f]

def makeData ():

x = numpy.arange (-10, 10, 0.1)

y = numpy.arange (-10, 10, 0.1)

xgrid, ygrid = numpy.meshgrid(x, y)

zgrid = int(args[0]) \* numpy.power(xgrid, 2) + int(args[1]) \* numpy.power(ygrid, 2) + int(args[2]) \* numpy.exp(-xgrid) + int(args[3]) \* numpy.exp(-ygrid)

return xgrid, ygrid, zgrid

x, y, z = makeData()

fig = pylab.figure()

axes = Axes3D(fig)

axes.plot\_surface(x, y, z, rstride=4, cstride=4, cmap= cm.jet )

pylab.show()

**CheckSquare.py**

import numpy

import sympy

import sympy.parsing

import pylab

from scipy.integrate import dblquad

import math

args = numpy.loadtxt("C:\\Users\masha\OneDrive\Учеба\Третий курс\CourseProject2.0\CourseProject2.0\\bin\Debug\course.txt", delimiter='\n', dtype= numpy.int)

a = int(args[0])

b = int(args[1])

c = int(args[2])

d = int(args[3])

startX = int(args[4])

finishX = int(args[5])

startY = int(args[6])

finishY = int(args[7])

function = lambda y, x: math.sqrt(1 + math.pow(2 \* a \* x + c \* math.exp(-x), 2) + math.pow(2 \* b \* y + d \* math.exp(-y), 2))

square = dblquad(function, startX, finishX, lambda x: startY, lambda x: finishY)[0]

file = open('square.txt', 'w')

file.write(str(square))

**Приложение Б**

**(обязательное)**

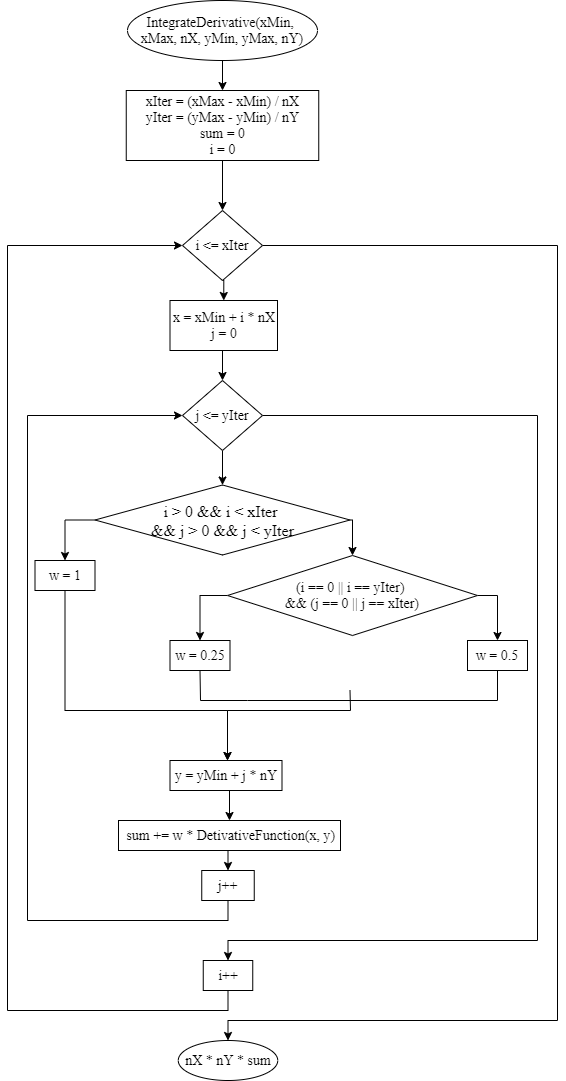
****

Рисунок Б.1 – Блок-схема метода трапеций