

Оглавление

Теоретическая часть.....	2
Базовые представления о моделировании аномалии силы тяжести над уступом [1].....	2
Базовые сведения о применяемых средствах и инструментах программной инженерии.....	3
Введение.....	3
Начальные сведения об объектно-ориентированном программировании (ООП) на примере языка программирования C#.....	4
Начальные сведения о системе построения клиентских приложений WPF.....	7
Начальные сведения о паттерне проектирования MVVM.....	7
Практическая часть	9
Постановка задачи.....	9
Программирование предварительной версии визуального представления приложения (модуль “V”)	10
Программирование предварительной версии абстракции графического интерфейса приложения (модуль “VM”).....	12
Программирование предварительной версии модели обработки и расчёта данных (модуль “M”).....	15
Привязка данных из модуля “VM” к элементам визуального представления.....	16
Доработка модуля M и связывание его с абстракцией графического интерфейса приложения	25
Проверка работоспособности приложения	31
Вывод.....	32
Литература	33
Интернет-ресурсы	33

Теоретическая часть

Базовые представления о моделировании аномалии силы тяжести над уступом [1]

Аномалии силы тяжести, вызванные притяжением тел известной формы, размера и избыточной плотности, рассчитывают на основе закона всемирного тяготения (закона Ньютона). Для этого гравитирующее тело разбивают на элементарные массы dm ; рассчитывают аномалию такой точечной массы Δg_1 , которая равна вертикальной составляющей силы ньютоновского притяжения F_1 этой массой массы 1 г, находящейся в точке наблюдения А, т. е. берут составляющую силы притяжения по направлению действия силы тяжести Земли g ; наконец, используя принцип суперпозиции, определяют аномалию за счет притяжения всем телом Δg_T , как сумму притяжения всех элементарных точечных масс, которыми можно представить аномалиеобразующее тело.

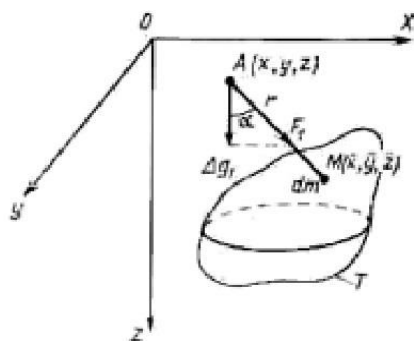


Рис. 1. Схема определения аномалий силы тяжести от элементарной массы dm и гравитирующего тела.

Аналитические решения с помощью уравнения (1.1) получаются для тел простой геометрической формы (шар, цилиндр и др.) с постоянной избыточной плотностью. Для тел более сложной формы, а особенно с переменной плотностью, возможны лишь численные решения интеграла (1.1) с помощью ЭВМ. Анализ решений прямых задач служит основой при

2, Контрольная работа на тему “Разработка оконного приложения для решения прямой задачи гравиразведки для уступа”

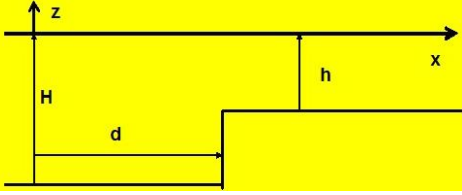
разработке приемов решения обратных задач гравиразведки для типовых геологических структур и объектов. Рассмотрим несколько примеров решения прямых и обратных задач для тел правильной геометрической формы.

$$\Delta g = \frac{G \cdot \Delta \sigma \cdot dV}{r^3}$$

$$\Delta g = G \int_V \frac{\Delta \sigma (\bar{z} - z) dV}{\left[\sqrt{(\bar{x} - x)^2 + (\bar{y} - y)^2 + (\bar{z} - z)^2} \right]^3}$$

$$\Delta \sigma = \sigma - \sigma_0 \quad \text{избыточная плотность}$$
(1.1)

Так, существует специальная формула для расчёта аномалии силы тяжести над уступом, выведенная из более общей формулы (1.1), которую мы отобразим в формуле 1.2.

$$\Delta g(x) = G\Delta\sigma \left[\pi(H-h) + 2H \operatorname{arctg} \frac{x-d}{H} - 2h \operatorname{arctg} \frac{x-d}{h} + (x-d) \ln \frac{(x-d)^2 + H^2}{(x-d)^2 + h^2} \right]$$

(1.2)

Базовые сведения о применяемых средствах и инструментах программной инженерии

Введение

Данное приложение будет разрабатываться средствами платформы для разработки прикладного программного обеспечения .NET, разработанная компанией Microsoft. Особенностью данной платформы является кроссплатформенность и поддержка множества языков программирования в одном проекте.

Кроссплатформенность (способность программного обеспечения работать с несколькими аппаратными платформами или операционными

3, Контрольная работа на тему “Разработка оконного приложения для решения прямой задачи гравиразведки для уступа”

системами) обеспечивается выполнением разработанных средствами вышеупомянутой платформы для разработки прикладного ПО в среде исполнения CLR (общезыковая среда выполнения – Common Language Runtime), которая может быть установлена на большинство имеющихся на текущий момент времени операционных систем, так как поддерживаемые в .NET компиляторы вместо компиляции кода непосредственно в машинный код генерируют управляемые модули, которые и могут быть запущены в вышеупомянутой среде выполнения [2].

Вышеупомянутая CLR способна поддерживать множество языков программирования также и из-за того, что компиляторы, работающие под .NET генерируют IL код, который может быть интерпретирован средой выполнения [2].

При разработке будем использовать систему построения клиентских приложений WPF, реализующую паттерн проектирования MVVM (о данной системе построения приложений и паттерне проектировании будет более подробно рассказано позже).

Начальные сведения об объектно-ориентированном программировании (ООП) на примере языка программирования C#

Парадигма ООП подразумевает под собой представление кодовой базы как некоторой совокупности объектов, являющихся экземплярами описывающих их классов. В наиболее типичном представлении, объект может содержать поля и методы, при этом поля описывают данные об объекте, а методы – функции, описывающие способы манипуляции представленными в классе полями.

Как поля, так и методы могут иметь модификаторы доступа, модификаторы принадлежности поля/метода к классу. Наиболее часто применяемые модификаторы доступа к полям это private, protected и public.

4, Контрольная работа на тему “Разработка оконного приложения для решения прямой задачи гравиразведки для уступа”

Модификатор `public` означает что поле/метод доступны для использования/изменения в любой части программы. Модификатор `private` означает что к методу или полю есть доступ исключительно внутри описанного класса, т.е. значения полей с таким модификатором смогут изменить извне исключительно методы с модификатором `public`.

Чтобы понять значение модификатора `protected` введём новое понятие – Наследование. Данная концепция объектно-ориентированного программирования подразумевает, что один класс может “унаследовать” поля и методы целевого. Стоит при этом отметить, что некоторые языки программирования как C++ позволяют применять множественное наследование, но большинство разработчиков языков программирования, в том числе и C#, отказались от того, чтобы давать возможность применять программисту множественное наследование, так как это приводило и к ухудшению поддерживаемости кода, и к ошибкам, не зависящим от действий программиста при некоторых исключительных случаях.

Вернёмся к описанию значения `protected`. Данный модификатор доступа подразумевает, что доступ к полям какого-либо класса есть исключительно у дочернего класса.

Пример: допустим есть изначальный класс (Main.cs), из которого запускается проект и класс, описывающий перемещение некоторого объекта по карте (ObjectInsideMap.cs) (в примере используется целочисленный тип данных `int`, потому применение параметров с плавающей точкой вызовет исключение (ошибку)):

Main.cs:

```
// В C# также существует такое понятие как пространства имён, на котором останавливаться не будем
namespace MapExample
{
    public class Program
    {
        static void Main(string[] args)
        {
            ObjectInsideMap obj_1 = new ObjectInsideMap();
            obj_1.ShowObjectLocation();
            obj_1.Move(-1, 4);
            obj_1.ShowObjectLocation();
        }
    }
}
```

5, Контрольная работа на тему “Разработка оконного приложения для решения прямой задачи гравиразведки для уступа”

ObjectInsideMap.cs:

```
namespace MapExample
{
    public class ObjectInsideMap {
        private int X_Location;
        private int Y_Location;

        // Сигнатура метода Move с необходимыми параметрами смещения объекта
        (offset)
        public void Move (int x_offset, int y_offset) {
            X_Location = X_Location + x_offset;
            Y_Location = Y_Location + y_offset;
        }

        // Метод необязательно может иметь какие-либо входные параметры
        public void ShowObjectLocation ()
        {
            Console.WriteLine("X coord: " + X_Location + ", Y coord: " +
                Y_Location);
        }
    }
}
```

Данный класс содержит два приватных поля, которые отвечают за местоположение объекта на карте, а также два публичных метода, один из которых описывает по какому принципу в зависимости от входных параметров “offset” изменяется местоположение объекта, второй выводит значения местоположения на монитор.

Помимо модификаторов доступа для полей и методов существует модификатор `static`, определяющий принадлежность поля/метода к классу либо к экземпляру класса, который удобен, когда, например необходимо создать метод, который бы не был привязан к конкретному типу объектов и который можно было бы вызвать, не создавая экземпляр класса.

Пример: допустим есть уже упомянутый изначальный класс (Main.cs), из которого запускается проект и класс (MathClass.cs), который содержит метод возведения целого числа во вторую степень:

Main.cs:

```
namespace MapExample
{
    public class Program
    {
        static void Main(string[] args)
        {
            ObjectInsideMap obj_1 = new ObjectInsideMap();
            obj_1.ShowObjectLocation();
            obj_1.Move(-1, 4);
            obj_1.ShowObjectLocation();
        }
    }
}
```

```

        // Произойдёт перемещение объекта на 3^2 = 9 шагов по оси X и 4 по оси Y
        // После этого действия объект придёт на координаты (8,8)
        obj_1.Move(MathClass.RaiseToSecondDegree(3), 4);
        obj_1.ShowObjectLocation();
    }
}

MathClass.cs:

// Оператор импорта
using System;

namespace MapExample
{
    public class MathClass
    {
        public static int RaiseToSecondDegree (int value)
        {
            return Convert.ToInt32(Math.Pow(value, 2));
        }
    }
}

```

Начальные сведения о системе построения клиентских приложений WPF.

Уже упомянутая система построения клиентских приложений WPF, реализующая паттерн проектирования MVVM, представляет из себя аналог своего предшественника WinForms, ориентированный под Microsoft .NET. Одним из главных преимуществ WPF над WinForms является обеспечение большей гибкости разработки, в том числе за счёт использования языка XAML для вёрстки интерфейса приложения и привязок данных.

Привязки данных позволяют через расширения разметки XAML связывать с элементами окна различные данные из классов используемого языка программирования.

Начальные сведения о паттерне проектирования MVVM

Данный способ организации кода предусматривает разделение приложения на 3 модуля:

M – model. Содержит в себе логику работы с данными.

V – view. Представляет из себя визуальное представление разрабатываемого приложения. В случае с .NET описывается языком разметки XAML.

VM – viewModel. Абстракция графического интерфейса, содержащая в себе фундаментальные данные приложения. Эти данные впоследствии будут связаны с view посредством привязок. Данный модуль так же содержит описание того, как именно будет применяться логика работы с данными из модуля model.

В интернете существует ошибочная трактовка назначения модулей паттерна проектирования MVVM (рис. 2) (подобная трактовка подразумевает под собой нарушение одного из принципов SOLID - принципа единственной ответственности).

Шаблон MVVM делится на три части:

- Модель (англ. Model) (так же, как в классической MVC) представляет собой логику работы с данными и описание фундаментальных данных, необходимых для работы приложения.
- Представление (англ. View) — графический интерфейс (окна, списки, кнопки и т. п.). Выступает подписчиком на событие изменения значений свойств или команд, предоставляемых Моделью Представления. В случае, если в Модели Представления изменилось какое-либо свойство, то она оповещает всех подписчиков об этом, и Представление, в свою очередь, запрашивает обновлённое значение свойства из Модели Представления. В случае, если пользователь воздействует на какой-либо элемент интерфейса, Представление вызывает соответствующую команду, предоставленную Моделью Представления.
- Модель Представления (англ. ViewModel) — с одной стороны, абстракция Представления, а с другой — обёртка данных из Модели, подлежащих связыванию. То есть, она содержит Модель, преобразованную к Представлению, а также команды, которыми может пользоваться Представление, чтобы влиять на Модель.

Рис. 2. Пример ошибочной трактовки назначения модулей паттерна проектирования MVVM [интернет-ресурс 1].

Практическая часть

Постановка задачи

В ходе данной контрольной работы необходимо разработать оконное приложение с графическим интерфейсом для отображения уступа и аномалии над ним.

Для построения модели уступа необходимо знать следующие параметры модели среды (рис. 3):

- 1) Глубина залегания верхней кромки уступа h ;
- 2) Глубина залегания нижней кромки уступа H ;
- 3) Расположение выступа d относительно нулевой оси измерения;
- 4) Плотность вмещающих пород $\sigma_{\text{вмещ}}$ и плотность пород, составляющих рассматриваемый уступ $\sigma_{\text{уст}}$.

Также необходимо знать значение гравитационной постоянной, которая будет взята за $G = 0,00667$ в силу того, что значения составляющие гравитационную аномалию над уступом необходимо измерять в мГалл.

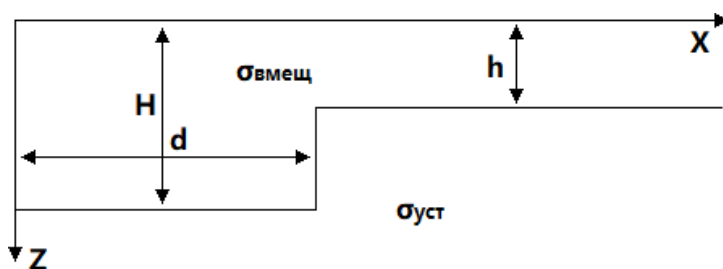


Рис. 3. Элементы уступа, значения которых необходимо знать для расчёта значений аномалии силы тяжести над ним.

При расчёте аномалии необходимо пользоваться формулой (1.2).

Для выполнения поставленной задачи необходимо создать предварительный интерфейс приложения со следующими составляющими:

1) Шесть элементов типа “TextBox” для введения численных параметров моделируемой среды и отображения сообщений для пользователя о выполнении или невыполнении операции по введенным значениям. Поле должно быть доступным только для чтения;

2) Восемь элементов типа “Label” для подписи каждого из вышеприведенных элементов;

3) Два элемента типа “Rectangle” для обозначения границ элементов типа “Canvas”, которые будут добавлены позже в составе ViewBox;

4) Один элемент типа “Button” для подтверждения расчёта аномалии по введенным параметрам рассматриваемой среды.

Также необходимо определиться с проверкой введенных параметров среды на корректность, для этого введём ряд первоначальных условий:

1) $h > 0$;

2) $H > 0 \text{ \& } H \geq h$;

3) $d > 1$;

4) $\sigma_{\text{вмещ}} > 0$; $\sigma_{\text{уст}} > 0$;

5) $\sigma_{\text{уст}} > \sigma_{\text{вмещ}}$.

Программирование предварительной версии визуального представления приложения (модуль “V”)

Последовательно добавим в XAML код визуального представления все вышеописанные элементы. При этом будем использовать следующий шаблон для каждого из элементов (шаблон написан на псевдокоде):

```
<ElementType Margin="X,Y,0,0" Height="value" Width="value"
HorizontalAlignment="Left" VerticalAlignment="Top"/>
```

Необходимость применения такого шаблона обусловлена необходимостью обеспечения большей гибкости разработки, таким образом добавив в описание элемента параметры `HorizontalAlignment="Left"` и `VerticalAlignment="Top"` у нас появляется возможность размещать верхний

10, Контрольная работа на тему “Разработка оконного приложения для решения прямой задачи гравirazведки для уступа”

левый край элемента в зависимости от введенных координат в следующем параметре: `Margin="X,Y,0,0"`, при этом отсчет будет вестись от верхнего левого края. Параметры “Height” и “Width”, обозначают соответственно высоту и ширину объекта.

Таким образом в блоке `Grid` имеем следующий, сформированный на языке разметки XAML интерфейс приложения. Визуальный вид полученного приложения отобразим на рис. 4.

```
<Grid>
    <Rectangle Margin="500,40,0,0" Height="350" Width="1000" HorizontalAlignment="Left"
VerticalAlignment="Top" Stroke="Black" />
    <Rectangle Margin="500,400,0,0" Height="250" Width="1000" HorizontalAlignment="Left"
VerticalAlignment="Top" Stroke="Black" />
    <Label Margin="10,40,0,0" Height="30" Width="140" HorizontalAlignment="Left"
VerticalAlignment="Top" Content="Панель сообщений:" />
    <TextBox Margin="10,75,0,0" Height="30" Width="420" HorizontalAlignment="Left"
VerticalAlignment="Top" IsReadOnly="True" />
    <Label Margin="10,120,0,0" Height="30" Width="231" HorizontalAlignment="Left"
VerticalAlignment="Top" Content="Глубина залегания нижней кромки:" />
    <TextBox Margin="10,155,0,0" Height="30" Width="180" HorizontalAlignment="Left"
VerticalAlignment="Top" />
    <Label Margin="250,120,0,0" Height="30" Width="217" HorizontalAlignment="Left"
VerticalAlignment="Top" Content="Глубина залегания верхней кромки:" />
    <TextBox Margin="250,155,0,0" Height="30" Width="180" HorizontalAlignment="Left"
VerticalAlignment="Top" />
    <Label Margin="10,200,0,0" Height="30" Width="231" HorizontalAlignment="Left"
VerticalAlignment="Top" Content="Плотность вмещающих пород:" />
    <TextBox Margin="10,235,0,0" Height="30" Width="180" HorizontalAlignment="Left"
VerticalAlignment="Top" />
    <Label Margin="250,200,0,0" Height="30" Width="217" HorizontalAlignment="Left"
VerticalAlignment="Top" Content="Плотность пород уступа:" />
    <TextBox Margin="250,235,0,0" Height="30" Width="180" HorizontalAlignment="Left"
VerticalAlignment="Top" />
    <Label Margin="10,280,0,0" Height="30" Width="370" HorizontalAlignment="Left"
VerticalAlignment="Top" Content="Расположение выступа относительно нулевой оси измерения:" />
    <TextBox Margin="10,315,0,0" Height="30" Width="180" HorizontalAlignment="Left"
VerticalAlignment="Top" />
    <Button Margin="10,385,0,0" Height="50" Width="210" HorizontalAlignment="Left"
VerticalAlignment="Top" Content="Рассчитать аномалию над уступом" />
</Grid>
```

При этом в тэге описания параметров самого окна имеем:

```
WindowStartupLocation="CenterScreen" Title="Решение прямой задачи гравиразведки
для уступа" Height="730" Width="1550" ResizeMode="NoResize"
```

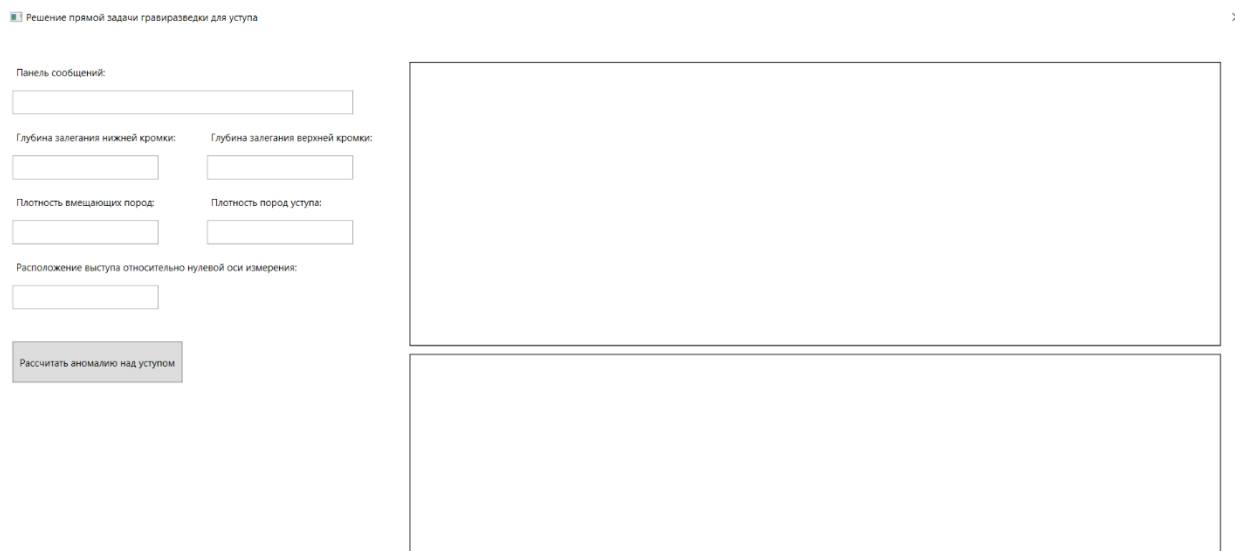


Рис. 4. Итоговый интерфейс приложения на данной стадии разработки.

Программирование предварительной версии абстракции графического интерфейса приложения (модуль “VM”)

Для нормального функционирования разрабатываемого приложения добавим два класса, один будет содержать алгоритм обновления данных по ходу работы приложения посредством событий, от данного класса впоследствии будет наследоваться класс абстракции графического интерфейса самого приложения. Располагаться данный класс будет в файле “NotifyPropertyChanged.cs”.

Второй класс будет также посредством событий предоставлять возможность манипулировать управляющими элементами таких типов, как “Button”. Данный класс будет располагаться в файле “Command.cs”.

NotifyPropertyChanged.cs :

```
using System.ComponentModel;
using System.Runtime.CompilerServices;

namespace GFDirectTasksSolver.ViewModelService
{
    // Интерфейс INotifyPropertyChanged не стоит путать с классом.
    public class NotifyPropertyChanged : INotifyPropertyChanged
    {
        public event PropertyChangedEventHandler? PropertyChanged;
```

```

        public virtual void CheckChanges([CallerMemberName] string property = "")
        {
            PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(property));
        }
    }
}

Command.cs :
using System.Windows.Input;
using System;

namespace GFDirectTasksSolver.ViewModelService
{
    public class Command : ICommand
    {
        private Action<object> execute;
        private Func<object, bool> canExecute;

        public event EventHandler? CanExecuteChanged
        {
            add { CommandManager.RequerySuggested += value; }
            remove { CommandManager.RequerySuggested -= value; }
        }

        public Command(Action<object> _execute, Func<object, bool> _canExecute = null)
        {
            execute = _execute;
            canExecute = _canExecute;
        }

        public bool CanExecute(object parameter)
        {
            return canExecute == null || canExecute(parameter);
        }

        public void Execute(object parameter)
        {
            execute(parameter);
        }
    }
}

```

Оба файла разместим в директории “ViewModelService”.

Приступим к программированию класса абстракции визуального интерфейса, который разместим в файле “MainViewModel.cs”.

Сам класс унаследуем от уже добавленного класса “NotifyPropertyChanged” и для удобства разобьём директивой #region класс на 3 составляющие:

- 1) Переменные для взаимодействия с графическими элементами приложения;
- 2) Переменные для взаимодействия с основным интерфейсом приложения;

3) Переменные для обмена данными между моделью и визуальным представлением приложения.

Итоговое содержимое полученного класса отобразим ниже:

MainWindowViewModel.cs:

```
using GFDirectTasksSolver.ViewModelService;
using System.Windows.Controls;

namespace GFDirectTasksSolver
{
    public class MainWindowViewModel : NotifyPropertyChanged
    {
        #region переменные для взаимодействия с графическими элементами приложения (not
        implemented)

        #endregion

        #region переменные для взаимодействия с основным интерфейсом приложения

        // Переменная для отображения пользователю сообщений об ошибках / успешно
        завершённых операциях
        public string infoPanel;
        public string InfoPanel
        {
            get { return infoPanel; }
            set
            {
                infoPanel = value;
                CheckChanges();
            }
        }

        // Для ввода информации о глубине залегания нижней кромки
        public decimal depthLowerEdge;
        public decimal DepthLowerEdge
        {
            get { return depthLowerEdge; }
            set
            {
                depthLowerEdge = value;
                CheckChanges();
            }
        }

        // Для ввода информации о глубине залегания верхней кромки
        public decimal depthHigherEdge;
        public decimal DepthHigherEdge
        {
            get { return depthHigherEdge; }
            set
            {
                depthHigherEdge = value;
                CheckChanges();
            }
        }

        // Для ввода информации о плотности вмещающих пород
        public decimal hostRocksDensity;
        public decimal HostRocksDensity
        {
            get { return hostRocksDensity; }
            set
            {
                hostRocksDensity = value;
            }
        }
    }
}
```

14, Контрольная работа на тему “Разработка оконного приложения для решения прямой задачи гравиразведки для уступа”

```

        CheckChanges();
    }
}

// Для ввода информации о плотности пород, слагающих уступ
public decimal ledgeRocksDensity;
public decimal LedgeRocksDensity
{
    get { return ledgeRocksDensity; }
    set
    {
        ledgeRocksDensity = value;
        CheckChanges();
    }
}

// Для ввода информации о расположении выступа относительно нулевой оси
измерения
public decimal overhangLocation;
public decimal OverhangLocation
{
    get { return overhangLocation; }
    set
    {
        overhangLocation = value;
        CheckChanges();
    }
}

// Для начала расчёта аномалии силы тяжести по клику на элемент типа "Button"
(not implemented)
public Command CalculateAnomaly
{
    get
    {
        return new Command(
            (obj) =>
            {
                // Здесь будет содержаться логика работы по отрис-е аномалий
            }
        );
    }
}

#endregion

#region переменные для передачи данных по аномалии между моделью и визуальным
представлением приложения (not implemented)

#endregion
}
}

```

Программирование предварительной версии модели обработки и расчёта данных (модуль “М”)

В папке с разрабатываемым решением создадим новый проект типа “библиотека”, который назовём “Model”. В нём создадим директорию “CalculateAnomalyService” (подобного рода разделение необходимо для удобства дальнейшего расширения приложения, например если потребуется

добавить расчёт гравитационной аномалии над шаром, цилиндром и т.п.), в которую добавим класс для расчёта аномалии силы тяжести над уступом для отдельной точки X, назовём класс “GravityOverLedgeCalculator”. В нём, согласно формуле 1.2 реализуем вышеуказанный функционал. Полученный код отобразим ниже:

GravityOverLedgeCalculator.cs :

```
using System;

namespace Model.CalculateAnomalyService
{
    public class GravityOverLedgeCalculator
    {
        private const decimal G = 0.00667M;
        private const decimal PI = 3.141593M;

        // Сигнатура метода для расчёта аномалии над уступом в конкретной точке X
        public static decimal CalculateGravityOverLedge (
            decimal X,
            decimal depthLowerEdge,
            decimal depthHigherEdge,
            decimal hostRocksDensity,
            decimal ledgeRocksDensity,
            decimal overhangLocation
        )
        {
            return G*(ledgeRocksDensity - hostRocksDensity) * (PI*(depthLowerEdge -
            depthHigherEdge)
                + 2 * depthLowerEdge * Convert.ToDecimal(Math.Atan(Convert.ToDouble((X -
            overhangLocation) / depthLowerEdge)))
                - 2 * depthHigherEdge * Convert.ToDecimal(Math.Atan(Convert.ToDouble((X -
            overhangLocation) / depthHigherEdge)))
                + (X - overhangLocation) * Convert.ToDecimal (Math.Log (Convert.ToDouble (
                ((X - overhangLocation) * (X - overhangLocation) + depthLowerEdge *
            depthLowerEdge)
                /
                ((X - overhangLocation) * (X - overhangLocation) + depthHigherEdge
            * depthHigherEdge)
                )))
        );
    }
}
```

Привязка данных из модуля “VM” к элементам визуального представления

В данной части будет произведена привязка переменных из VM с элементами окна, описанными на языке разметки XAML в модуле V, будут настроены расчёт и отображение гравитационных аномалий над уступом, для чего в Grid часть разметки XAML будет добавлено два Viewbox контейнера, содержащие информацию об используемых для отрисовки ресурсов

(посредством привязок данных, указывающий на список точек из VM) и панелей визуального представления, в нашем случае это 2 элемента Canvas. Также упомянутый контейнер будет содержать указание на тип отрисовываемых фигур, в нашем случае это будут элементы типа Line (XAML).

Обобщение принципа работы приложения: алгоритм работы приложения будет выглядеть следующим образом: после нажатия пользователем нужного элемента типа “Button”, сначала на вход поступают параметры моделируемой среды и проверяются на корректность, после по введенным данным определяются максимальные значения для осей и создается список, отражающий значения аномалии силы тяжести для каждой отдельно взятой координаты X, по полученным значениям создается список линий для отрисовки аномалий, при этом координаты линий будут адаптироваться под отрисовку в элементе “Canvas”.

Первым шагом создадим новый проект “Core”, в котором будем хранить все необходимые для работы приложения абстракции и сущности. В созданном проекте добавим папку “Entities” (сущности), в которой создадим новый класс для абстракции отрисовываемых линий, данный класс будет содержать свойства, отражающие координаты начальной и конечной точек и цвет заданной линии. Описание вышеупомянутой абстракции из файла “Line.cs” отобразим ниже:

Line.cs:

```
using System.Windows.Media;

namespace Core.Entities
{
    public class Line
    {
        public int X1 { get; init; }
        public int Y1 { get; init; }
        public int X2 { get; init; }
        public int Y2 { get; init; }
        public SolidColorBrush Color { get; init; }
    }
}
```

Примечание: пространство имён System.Windows.Media изначально недоступно для проектов, создаваемых в среде разработки Microsoft Visual Studio, потому для его использования необходимо скачать из сети “Интернет” файл “PresentationCore.dll” и в зависимостях отдельно взятого проекта указать путь до данного файла способом, указанным в рис. 5 – 6.

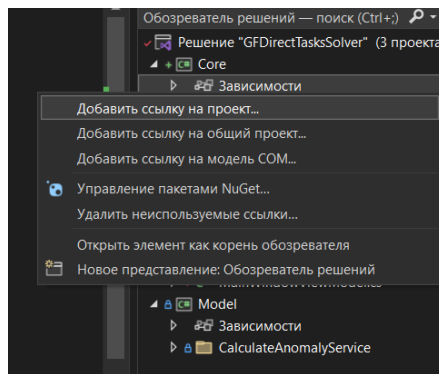


Рис. 5. Переход в меню зависимостей проекта.

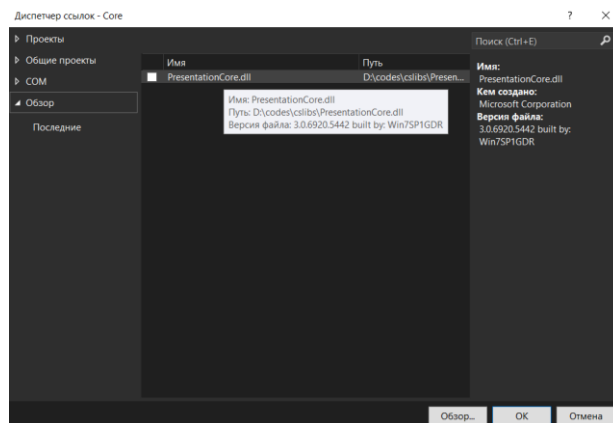


Рис. 6. Меню зависимостей проекта. Для добавления нужного файла необходимо указать путь до целевого файла перейдя в меню выбора через кнопку “Обзор”.

Также необходимо в тот же проект добавить абстракцию полученных в результате расчёта аномалии данных, данная абстракция будет содержать два поля, для координаты X и полученного значения силы тяжести над X . Полученный код из файла “AbstractPoint.cs” отобразим ниже:

AbstractPoint.cs :

```
namespace Core.Entities
{
    public class AbstractPoint
    {
        public decimal X;
        public decimal Y;
    }
}
```

Далее добавим привязки к каждому из элементов. Для этого необходимо указать компилятору откуда брать данные для визуального представления приложения, что обеспечивает добавление следующего тега в XAML код:

18, Контрольная работа на тему “Разработка оконного приложения для решения прямой задачи гравиразведки для уступа”

```
<Window.DataContext>
    <local:MainWindowViewModel/>
</Window.DataContext>
```

Совместно с одним из элементов тега описания окна, который автоматически генерируется при создании нового WPF проекта в среде разработки Microsoft Visual Studio он указывает из какого конкретного пространства имён (namespace) ожидать свойства для привязки:

```
xmlns:local="clr-namespace:GFDirectTasksSolver"
```

Примечание: после добавления данного кода в XAML разметку будет возникать ошибка о том, что среда разработки не может найти целевое пространство имён вследствие того, что конфигурация проекта всё ещё не содержит указания на контекст данных, для устранения ошибки достаточно через среду разработки запустить проект для обновления конфигурации проекта.

Перед тем, как начать привязку свойств к визуальному представлению приложения, добавим указание на пространство имён ранее созданного проекта “Core” в разметку XAML для использования созданного класса в отрисовке линий, для этого необходимо сначала указать в зависимостях основного проекта ссылку на проект “Core”, для этого по аналогии с рис. 5. добавим ссылку на нужный проект, интерфейс диспетчера ссылок отобразим на рис. 7.

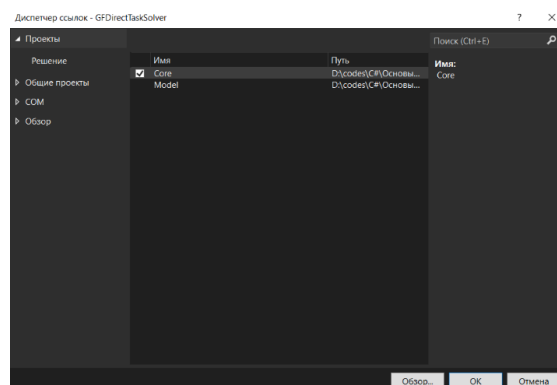


Рис. 7. Интерфейс диспетчера ссылок.

Далее в свойства окна добавим переменную ссылки на необходимое пространство имён:

```
xmlns:core="clr-namespace:Core.Entities;assembly=Core"
```

Далее последовательно привяжем элементы окна к соответствующим им свойствам из VM, также для последующего отображения модели уступа и модели аномалии силы тяжести над ним, в VM добавим свойства типа “List<Line>” для использования списка отрисовываемых линий визуальным представлением приложения, также добавим свойство типа “List<AbstractPoint>” для расчёта точек силы тяжести над каждой из координат X.

Также добавим переменные для отображения на координатной плоскости численные значения делений, создав свойства и переменные для отображения делений новые пары свойство/переменная. Кодовую базу модуля V (без прописанного в свойстве “CalculateAnomaly” алгоритма обработки данных) отобразим ниже:

MainWindow.cs:

```
using Core.Entities;
using GFDirectTasksSolver.ViewModelService;
using System;
using System.Collections.Generic;
using System.Windows.Media;

namespace GFDirectTasksSolver
{
    public class MainWindowViewModel : NotifyPropertyChanged {
        #region переменные для взаимодействия с графическими элементами приложения
        public List<Line> anomalyModelLines = new List<Line>();
        public List<Line> AnomalyModelLines
        {
            get { return anomalyModelLines; }
            set {
                anomalyModelLines = value;
                CheckChanges();
            }
        }
        public List<Line> environmentModelLines = new List<Line>();
        public List<Line> EnvironmentModelLines
        {
            get { return environmentModelLines; }
            set {
                environmentModelLines = value;
                CheckChanges();
            }
        }
        #endregion
        #region переменные для взаимодействия с основным интерфейсом приложения
        // Переменная для отображения пользователю сообщений об ошибках / успешно
        // завершённых операциях
        public string infoPanel;
        public string InfoPanel
        {
            get { return infoPanel; }
            set {
```

```

        infoPanel = value;
        CheckChanges();
    }
}
// Для ввода информации о глубине залегания нижней кромки
public decimal depthLowerEdge;
public decimal DepthLowerEdge
{
    get { return depthLowerEdge; }
    set {
        depthLowerEdge = value;
        CheckChanges();
    }
}
// Для ввода информации о глубине залегания верхней кромки
public decimal depthHigherEdge;
public decimal DepthHigherEdge
{
    get { return depthHigherEdge; }
    set {
        depthHigherEdge = value;
        CheckChanges();
    }
}
// Для ввода информации о плотности вмещающих пород
public decimal hostRocksDensity;
public decimal HostRocksDensity
{
    get { return hostRocksDensity; }
    set {
        hostRocksDensity = value;
        CheckChanges();
    }
}
// Для ввода информации о плотности пород, слагающих уступ
public decimal ledgeRocksDensity;
public decimal LedgeRocksDensity
{
    get { return ledgeRocksDensity; }
    set {
        ledgeRocksDensity = value;
        CheckChanges();
    }
}
// Для ввода информации о расположении выступа относительно нулевой оси
измерения
public decimal overhangLocation;
public decimal OverhangLocation
{
    get { return overhangLocation; }
    set {
        overhangLocation = value;
        CheckChanges();
    }
}
// Для начала расчёта аномалии силы тяжести по клику на элемент типа "Button"
public Command CalculateAnomaly
{
    get {
        return new Command(
            (obj) => {
                // Not implemented
                throw new NotImplementedException();
            }
        );
    }
}
}
#endregion

```

21, Контрольная работа на тему “Разработка оконного приложения для решения прямой задачи гравиразведки для уступа”

```

        #region переменные для передачи данных по аномалии между моделью и визуальным
представлением приложения
        public List<AbstractPoint> anomalyPoints;
        public List<AbstractPoint> AnomalyPoints
        {
            get { return anomalyPoints; }
            set {
                anomalyPoints = value;
                CheckChanges();
            }
        }
    #endregion

    #region переменные для отображения в визуальном представлении численных значений
делений на координатной плоскости

    public decimal dg_max;
    /// <summary>
    /// Максимальное значение по оси dg
    /// </summary>
    public decimal dg_Max
    {
        get { return dg_max; }
        set {
            dg_max = value;
            CheckChanges();
        }
    }

    public decimal dg_med;
    /// <summary>
    /// Среднее значение по оси dg
    /// </summary>
    public decimal dg_Med
    {
        get { return dg_med; }
        set {
            dg_med = value;
            CheckChanges();
        }
    }

    public decimal x_max;
    /// <summary>
    /// Максимальное значение по оси X
    /// </summary>
    public decimal x_Max
    {
        get { return x_max; }
        set {
            x_max = value;
            CheckChanges();
        }
    }

    public decimal x_med;
    /// <summary>
    /// Среднее значение по оси X
    /// </summary>
    public decimal x_Med
    {
        get { return x_med; }
        set {
            x_med = value;
            CheckChanges();
        }
    }
}

```

```

    public decimal z_max;
    /// <summary>
    /// Максимальное значение по оси Z
    /// </summary>
    public decimal z_Max
    {
        get { return z_max; }
        set {
            z_max = value;
            CheckChanges();
        }
    }

    public decimal z_med;
    /// <summary>
    /// Среднее значение по оси Z
    /// </summary>
    public decimal z_Med
    {
        get { return z_med; }
        set {
            z_med = value;
            CheckChanges();
        }
    }
}
#endregion
}
}

```

В код визуального представления приложения добавим контейнер ViewBox, содержащий как главный элемент рендеринга, так и список ресурсов, которые будут использоваться приложением, также добавим подписи для осей координат и численные значения делений, полученную кодовую базу из файла “MainWindow.xaml”, отобразим ниже:

MainWindow.xaml:

```

<Window x:Class="GFDirectTasksSolver.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:GFDirectTasksSolver"
        xmlns:core="clr-namespace:Core.Entities;assembly=Core"
        mc:Ignorable="d"
        WindowStartupLocation="CenterScreen"
        Title="Решение прямой задачи гравirazведки для уступа" Height="750" Width="1550"
        ResizeMode="NoResize">
    <Window.DataContext>
        <local:MainWindowViewModel/>
    </Window.DataContext>
    <Grid>
        <!-- Элементы для обозначения границ графика аномалии и её отрисовки -->
        <Rectangle Margin="500,40,0,0" Height="350" Width="1000"
            HorizontalAlignment="Left" VerticalAlignment="Top" Stroke="Black" />
        <Viewbox Stretch="Uniform" HorizontalAlignment="Left" VerticalAlignment="Top"
            Margin="501,47,58,0">
            <ItemsControl ItemsSource="{Binding AnomalyModelLines}" Width="1000"
                Height="350" >
                <ItemsControl.ItemsPanel>
                    <ItemsPanelTemplate>
                        <Canvas Background="#fff" />

```

```

        </ItemsPanelTemplate>
    </ItemsControl.ItemsPanel>
    <ItemsControl.Resources>
        <Style TargetType="ContentPresenter">
            <Setter Property="Canvas.Left" Value="0" />
            <Setter Property="Canvas.Top" Value="0" />
        </Style>
        <!-- В DataTemplate прописываем путь к классу абстракции
отрисовываемых линий -->
        <DataTemplate DataType="{x:Type core:Line}">
            <Line X1="{Binding X1}" X2="{Binding X2}" Y1="{Binding Y1}"
Y2="{Binding Y2}" Stroke="{Binding Color}" StrokeThickness="1"/>
        </DataTemplate>
    </ItemsControl.Resources>
</ItemsControl>
</Viewbox>

<!-- Элементы для обозначения границ модели среды и её отрисовки -->
<Rectangle Margin="500,420,0,0" Height="250" Width="1000"
HorizontalAlignment="Left" VerticalAlignment="Top" Stroke="Black" />
<Viewbox Stretch="Uniform" HorizontalAlignment="Left" VerticalAlignment="Top"
Margin="501,421,58,0">
    <ItemsControl ItemsSource="{Binding EnvironmentModelLines}" Width="1000"
Height="250">
        <ItemsControl.ItemsPanel>
            <ItemsPanelTemplate>
                <Canvas Background="#fff" />
            </ItemsPanelTemplate>
        </ItemsControl.ItemsPanel>
        <ItemsControl.Resources>
            <Style TargetType="ContentPresenter">
                <Setter Property="Canvas.Left" Value="0" />
                <Setter Property="Canvas.Top" Value="0" />
            </Style>
            <DataTemplate DataType="{x:Type core:Line}">
                <Line X1="{Binding X1}" X2="{Binding X2}" Y1="{Binding Y1}"
Y2="{Binding Y2}" Stroke="{Binding Color}" StrokeThickness="1"/>
            </DataTemplate>
        </ItemsControl.Resources>
    </ItemsControl>
</Viewbox>

<!-- Элементы для ввода параметров среды -->
<Label Margin="10,40,0,0" Height="30" Width="140" HorizontalAlignment="Left"
VerticalAlignment="Top" Content="Панель сообщений:" />
<TextBox Margin="10,75,0,0" Height="30" Width="420" HorizontalAlignment="Left"
VerticalAlignment="Top" IsReadOnly="True" Text="{Binding InfoPanel}"/>

<Label Margin="10,120,0,0" Height="30" Width="231" HorizontalAlignment="Left"
VerticalAlignment="Top" Content="Глубина залегания нижней кромки:" />
<TextBox Margin="10,155,0,0" Height="30" Width="180" HorizontalAlignment="Left"
VerticalAlignment="Top" Text="{Binding DepthLowerEdge}"/>
<Label Margin="250,120,0,0" Height="30" Width="217" HorizontalAlignment="Left"
VerticalAlignment="Top" Content="Глубина залегания верхней кромки:" />
<TextBox Margin="250,155,0,0" Height="30" Width="180" HorizontalAlignment="Left"
VerticalAlignment="Top" Text="{Binding DepthHigherEdge}"/>

<Label Margin="10,200,0,0" Height="30" Width="231" HorizontalAlignment="Left"
VerticalAlignment="Top" Content="Плотность вмещающих пород:" />
<TextBox Margin="10,235,0,0" Height="30" Width="180" HorizontalAlignment="Left"
VerticalAlignment="Top" Text="{Binding HostRocksDensity}"/>
<Label Margin="250,200,0,0" Height="30" Width="217" HorizontalAlignment="Left"
VerticalAlignment="Top" Content="Плотность пород уступа:" />
<TextBox Margin="250,235,0,0" Height="30" Width="180" HorizontalAlignment="Left"
VerticalAlignment="Top" Text="{Binding LedgeRocksDensity}"/>

```

24, Контрольная работа на тему “Разработка оконного приложения для решения прямой задачи гравиразведки для уступа”


```

        <Label Margin="10,280,0,0" Height="30" Width="370" HorizontalAlignment="Left"
VerticalAlignment="Top" Content="Расположение выступа относительно нулевой оси
измерения:"/>
        <TextBox Margin="10,315,0,0" Height="30" Width="180" HorizontalAlignment="Left"
VerticalAlignment="Top" Text="{Binding OverhangLocation}"/>

        <!-- Элемент для запуска отрисовки графиков аномалии и модели среды -->
        <Button Margin="10,385,0,0" Height="50" Width="210" HorizontalAlignment="Left"
VerticalAlignment="Top" Content="Рассчитать аномалию над уступом" Command="{Binding
CalculateAnomaly}"/>

        <!-- Элементы для подписи графиков -->
        <Label Margin="500,10,0,0" Height="30" Width="231" HorizontalAlignment="Left"
VerticalAlignment="Top" Content="dg, мГалл"/>
        <Label Margin="480,40,0,0" Height="30" Width="40" HorizontalAlignment="Left"
VerticalAlignment="Top" Content="{Binding dg_Max}" FontSize="8"/>
        <Label Margin="480,205,0,0" Height="30" Width="40" HorizontalAlignment="Left"
VerticalAlignment="Top" Content="{Binding dg_Med}" FontSize="8"/>
        <Label Margin="480,380,0,0" Height="30" Width="40" HorizontalAlignment="Left"
VerticalAlignment="Top" Content="0" FontSize="8"/>

        <Label Margin="500,670,0,0" Height="30" Width="40" HorizontalAlignment="Left"
VerticalAlignment="Top" Content="Z, м"/>
        <Label Margin="480,410,0,0" Height="30" Width="40" HorizontalAlignment="Left"
VerticalAlignment="Top" Content="0" FontSize="8"/>
        <Label Margin="480,535,0,0" Height="30" Width="40" HorizontalAlignment="Left"
VerticalAlignment="Top" Content="{Binding z_Med}" FontSize="8"/>
        <Label Margin="480,660,0,0" Height="30" Width="40" HorizontalAlignment="Left"
VerticalAlignment="Top" Content="{Binding z_Max}" FontSize="8"/>

        <Label Margin="1500,370,0,0" Height="30" Width="40" HorizontalAlignment="Left"
VerticalAlignment="Top" Content="X, м"/>
        <Label Margin="1500,415,0,0" Height="30" Width="40" HorizontalAlignment="Left"
VerticalAlignment="Top" Content="X, м"/>
        <Label Margin="1490,387,0,0" Height="30" Width="40" HorizontalAlignment="Left"
VerticalAlignment="Top" Content="{Binding x_Max}" FontSize="8"/>
        <Label Margin="1490,403,0,0" Height="30" Width="40" HorizontalAlignment="Left"
VerticalAlignment="Top" Content="{Binding x_Max}" FontSize="8"/>
        <Label Margin="1000,387,0,0" Height="30" Width="40" HorizontalAlignment="Left"
VerticalAlignment="Top" Content="{Binding x_Med}" FontSize="8"/>
        <Label Margin="1000,403,0,0" Height="30" Width="40" HorizontalAlignment="Left"
VerticalAlignment="Top" Content="{Binding x_Med}" FontSize="8"/>
    </Grid>
</Window>

```

Доработка модуля М и связывание его с абстракцией графического интерфейса приложения

Добавим в приложение проверку введенных параметров на корректность согласно условиям, описанным в главе “Постановка задачи”. Для этого в проекте “Model” создадим новую папку “CheckEnvironmentModelService” и добавим туда класс “EnvironmentParamsChecker”. Данный класс будет содержать статический метод, который проверяет введенные значения на корректность, и, если

данные введены верно, возвращает true. Код, позволяющий решить данную задачу отобразим ниже:

EnvironmentParamsChecker.cs :

```
namespace Model.CheckEnvironmentModelService
{
    public class EnvironmentParamsChecker
    {
        public static bool CheckEnvironmentParams(
            decimal depthLowerEdge,
            decimal depthHigherEdge,
            decimal hostRocksDensity,
            decimal ledgeRocksDensity,
            decimal overhangLocation
        )
        {
            return (depthLowerEdge > 0)
                && (depthHigherEdge > 0)
                && (depthLowerEdge >= depthHigherEdge)
                && (hostRocksDensity > 0)
                && (ledgeRocksDensity > 0)
                && (ledgeRocksDensity > hostRocksDensity)
                && (overhangLocation > 1);
        }
    }
}
```

Добавим в данный модуль ещё один класс “ AnomalyPointsClass” в папку “CalculateAnomalyService”, создав в нём два статических метода: для генерации по заданной частоте измерений по оси X динамического массива типа “List<AbstractPoint>” и для определения наибольшего значения Y во входном массиве. Код из файла “AnomalyPointsClass.cs” код отобразим ниже:

AnomalyPointsClass.cs :

```
using Core.Entities;
using System;
using System.Collections.Generic;

namespace Model.CalculateAnomalyService {
    public class AnomalyPointsClass
    {
        // Для создания динамического массива с данными по аномалии
        // Генерируется по частоте дискретизации и параметрам моделируемой среды
        public static List<AbstractPoint> GetAnomalyPoints(
            int samplingRate,
            decimal x_Max,
            decimal depthLowerEdge,
            decimal depthHigherEdge,
            decimal hostRocksDensity,
            decimal ledgeRocksDensity,
            decimal overhangLocation
        )
        {
            List<AbstractPoint> anomalyPoints= new List<AbstractPoint>();
            decimal increment_X = x_Max / samplingRate;
            for (decimal X = 0; X <= x_Max; X += increment_X)
            {

```

```

        anomalyPoints.Add(
            new AnomalyPoint()
            {
                X = X,
                Y = GravityOverLedgeCalculator.CalculateGravityOverLedge(
                    X,
                    depthLowerEdge,
                    depthHigherEdge,
                    hostRocksDensity,
                    ledgeRocksDensity,
                    overhangLocation
                )
            }
        );
    }
    return anomalyPoints;
}

public static decimal GetMaxFromAnomalyList (List<AbstractPoint> anomalyPoints)
{
    decimal answer = Decimal.MinValue;
    foreach (var point in anomalyPoints)
    {
        if (answer < point.Y)
        {
            answer = point.Y;
        }
    }
    return answer;
}
}
}
}

```

Добавим в проект “Model” новую директорию “LineDrawService”, в которую добавим новый класс “LineCoordsCalculator”, по задумке содержащий статический метод. Данный метод будет принимать как входные данные вычисленные максимальные значения для осей, размеры полотна, в котором будет производиться отрисовка линий и набор точек для отрисовки и также содержать в своей сигнатуре параметр “IsMustInverted” в случае, если рисунок необходимо инвертировать по вертикали (о необходимости реализации такого функционала в методе речь зайдёт чуть позже) и параметр, задающий цвет для всех отрисовываемых линий. Возвращать данный метод будет набор линий для отрисовки с рассчитанными координатами для отрисовки на конкретном полотне. Данный метод будет использоваться как для расчёта расположения линий на графике аномалии, так и на полотне, которое визуализирует модель среды.

Содержимое вышеуказанного файла отобразим ниже:

27, Контрольная работа на тему “Разработка оконного приложения для решения прямой задачи гравиразведки для уступа”

LineCoordsCalculator.cs :

```
using Core.Entities;
using System;
using System.Collections.Generic;
using System.Windows.Media;

namespace Model.LineDrawService
{
    public class LineCoordsCalculator
    {
        public static List<Line> CalculateLinesCoords (
            decimal X_Max,
            decimal Y_Max,
            int CanvasHeight,
            int CanvasWidth,
            List<AbstractPoint> points,
            bool IsMustInverted,
            SolidColorBrush Color
        )
        {
            List<Line> lines = new List<Line>();

            // количество итераций "points.Count - 2" обусловлено тем, что по
            // n точкам можно построить только n-1 линий
            if ( IsMustInverted )
            {
                for (int i = 0; i <= points.Count - 2; i++)
                {
                    lines.Add(
                        new Line ()
                        {
                            X1 = Convert.ToInt32(points[i].X * (CanvasHeight / X_Max)),
                            Y1 = Convert.ToInt32(CanvasWidth-points[i].Y*(CanvasWidth /
Y_Max)),
                            X2 = Convert.ToInt32(points[i+1].X * (CanvasHeight /
X_Max)),
                            Y2 = Convert.ToInt32(CanvasWidth - points[i+1].Y *
(CanvasWidth / Y_Max)),
                            Color = Color
                        }
                    );
                }
            }
            else
            {
                for (int i = 0; i <= points.Count - 2; i++)
                {
                    lines.Add(
                        new Line()
                        {
                            X1 = Convert.ToInt32(points[i].X * (CanvasHeight / X_Max)),
                            Y1 = Convert.ToInt32(points[i].Y * (CanvasWidth / Y_Max)),
                            X2 = Convert.ToInt32(points[i+1].X * (CanvasHeight / X_Max)),
                            Y2 = Convert.ToInt32(points[i+1].Y * (CanvasWidth / Y_Max)),
                            Color = Color
                        }
                    );
                }
            }

            return lines;
        }
    }
}
```

Примечание: необходимость добавления инвертирования координат, по которым предполагается строить линии для отрисовки заключается в том, что координатные оси для графика аномалии и изображения модели уступа направлены в разные стороны: для графика аномалии снизу-вверх, для модели уступа сверху-вниз (рис. 8). При этом стоит отметить, что в элементе типа “Canvas” отсчёт координат всегда ведётся от верхнего левого угла, поэтому координаты линий для построения модели аномалии предполагается инвертировать.

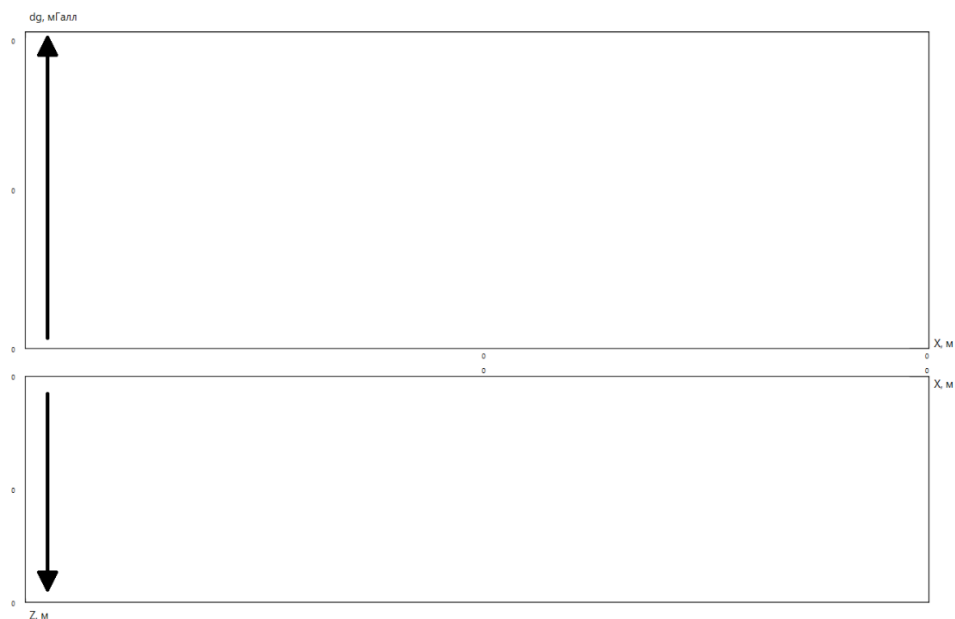


Рис. 8.

Завершающим шагом будет написание логики расчёта и отображения аномалии по нажатию кнопки “Рассчитать аномалию”. Связан данный элемент окна со свойством “CalculateAnomaly” типа “Command”, в котором соответственно и пропишем алгоритм обработки данных и их передачи в нужные свойства.

Вышеупомянутый алгоритм средствами условной конструкции “if-else” будет проверять на корректность введённые пользователем параметры моделируемой среды и в зависимости от значения, возвращённого из метода проверки данных выдавать в консоль сообщений результат проверки данных.

Если введены корректные данные, алгоритм по введённым значениям сначала рассчитает максимальные и средние значения для осей X и Z, после с

учётom принятой внутри алгоритма частоты дискретизации расчёта значений в различных точках X, произведёт расчёт точек для отрисовки аномалии над уступом, по данным точкам будет рассчитано максимальное и среднее значения для аномалии силы тяжести. Список точек будет передан в метод расчёта координат для линий, которые будут отрисованы на графике аномалии силы тяжести, при этом координаты для линий аномалии силы тяжести будут инвертированы, так как отсчёт координат в элементе “Canvas” ведётся от верхнего левого угла. Список линий будет передан в свойство, привязанное к элементу типа “Canvas”.

Отдельно, по максимальным и средним значениям X и Z, будут построены точки, по которым предполагается отображение модели среды. Список точек будет также передан в метод расчёта линий. Список линий для отрисовки модели среды также будет передан в метод расчёта координат линий для элемента типа “Canvas”. При этом инвертирование координат для линий не предполагается. Логика работы отобразим ниже:

```
public Command CalculateAnomaly
{
    get
    {
        return new Command(
            (obj) =>
            {
                if
                (EnvironmentParamsChecker.CheckEnvironmentParams(DepthLowerEdge, depthHigherEdge,
                    HostRocksDensity, LedgeRocksDensity, OverhangLocation))
                {
                    // Определение максимального значения X
                    x_Max = 2 * OverhangLocation;
                    x_Med = x_Max / 2;

                    // Определение максимального значения Z
                    z_Max = Math.Round(DepthLowerEdge * (1 +
                        (DepthLowerEdge/10)/(DepthLowerEdge) ), 2);
                    z_Med = z_Max / 2;

                    // Определение частоты измерений (или более обще –
                    // дискретизации) по оси X и расчёт значений аномалии для каждого из
                    // значений X с интервалом равным частоте измерений.
                    int SamplingRate = 50;
                    AnomalyPoints = AnomalyPointsClass.GetAnomalyPoints(
                        SamplingRate, x_Max, DepthLowerEdge, DepthHigherEdge,
                        HostRocksDensity, LedgeRocksDensity, OverhangLocation
                    );
                    dg_Max =
                    Math.Round(AnomalyPointsClass.GetMaxFromAnomalyList(AnomalyPoints), 2);
                    dg_Med = dg_Max / 2;
                }
            }
        );
    }
}
```

```

        // Задание набора линий для модели среды
        EnvironmentModelLines =
LineCoordsCalculator.CalculateLinesCoords(
        x_Max,
        z_Max,
        1000,
        250,
        new List<AbstractPoint> ()
        {
            new AbstractPoint () {X = 0, Y = DepthLowerEdge},
            new AbstractPoint () {X = OverhangLocation, Y =
DepthLowerEdge},
            new AbstractPoint () {X = OverhangLocation, Y =
DepthHigherEdge},
            new AbstractPoint () {X = x_Max , Y =
DepthHigherEdge}
        },
        false,
        Brushes.Black
    );

    // Задание набора линий для модели аномалии
    AnomalyModelLines =
LineCoordsCalculator.CalculateLinesCoords(
        x_Max,
        dg_Max,
        1000,
        350,
        AnomalyPoints,
        true,
        Brushes.Red
    );

    InfoPanel = "Успешно.";
}
else
{
    InfoPanel = "Введены некорректные параметры моделируемой
среды";
}
}
}
);
}
}
}

```

Проверка работоспособности приложения

Для проверки работоспособности приложения сначала введём заведомо некорректные параметры уступа, например при которых значение глубины залегания верхней кромки будет выше значения глубины залегания нижней кромки. После введём корректные параметры моделируемой среды. Интерфейс приложения после введения вышеуказанных вариантов параметров моделируемой среды отобразим на рис. 9 для некорректных данных и рис. 10 для корректных данных.

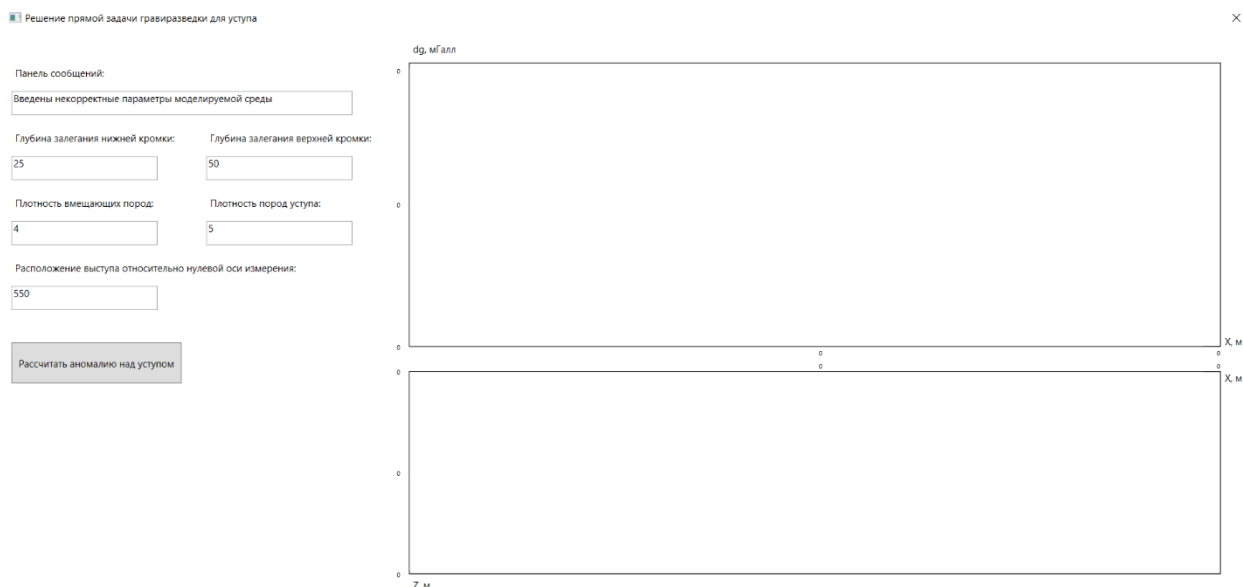


Рис. 9. Работа приложения при некорректных данных.

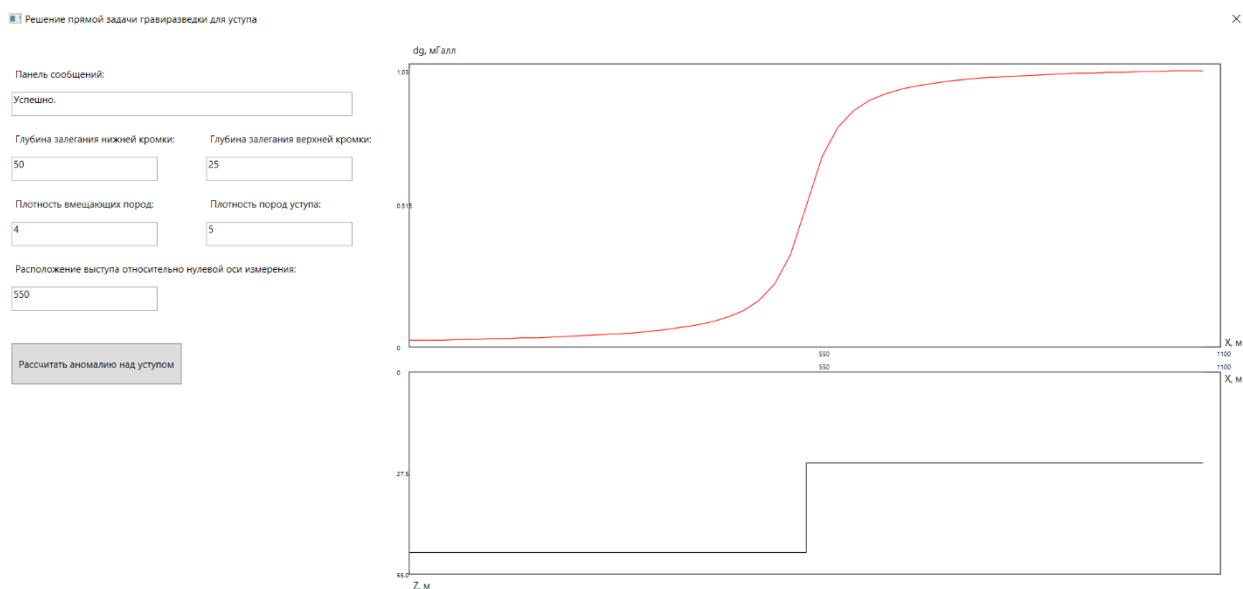


Рис. 10. Работа приложения при корректных данных.

Полученные изображения аномалии над уступом и модели уступа после введения корректных параметров моделируемой среды свидетельствуют о корректной работе разработанного приложения.

Вывод

В ходе данной лабораторной работы средствами системы построения клиентских приложений WPF на языке программирования C# с применением языка разметки XAML для вёрстки пользовательского интерфейса, было

разработано оконное приложение для расчёта и отображения графика гравитационной аномалии над уступом, а также моделируемой среды.

Литература

1. Серков В.А. Практическое руководство по математическому моделированию геоданных.
2. Джеффри Рихтер. CLR via C#. - Изд. - СПб.: Питер, 2013. - 896 с.

Интернет-ресурсы

1. <https://ru.wikipedia.org/wiki/Model-View-ViewModel>