**МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ**

**УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ**

**ГОМЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ**

**УНИВЕРСИТЕТ ИМЕНИ П. О. СУХОГО**

Факультет автоматизированных и информационных систем

Кафедра «Информационные технологии»

направление специальности 1-40 05 01 Информационные системы

и технологии (в проектировании и производстве)

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовому проекту

по дисциплине «Компьютерные системы конечноэлементных расчетов»

на тему: **«АНАЛИЗ НАПРЯЖЕННО-ДЕФОРМИРОВАННОГО**

**СОСОТОЯНИЯ ПЛОСКОЙ КОНСТРУКЦИИ – КРОНШТЕЙН»**

Исполнитель: студент гр. ИТП-31

Петрушкин А.А.

Руководитель: доцент Токочаков В.И.

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

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

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

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

Подписи членов комиссии

по защите курсового проекта: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Гомель 2020

**ОГЛАВЛЕНИЕ**

[Введение](#_Toc26404826) 4

[1 Обзор численных методов моделирования в механике](#_Toc26404827) 5

[1.1 Существующие численные методы](#_Toc26404828) 5

[1.2 Метод конечных разностей](#_Toc26404830) 5

[1.3 Метод граничных элементов](#_Toc26404830) 6

[1.4 Метод конечных элементов](#_Toc26404830) 7

[2 Алгоритмический анализ задачи](#_Toc26404827) 12

[2.1 Постановка задачи](#_Toc26404830) 12

[2.2 Метод конечных элементов для решения двумерных задач теории упру-гости](#_Toc26404830) 12

[3 Программная реализация задачи](#_Toc26404827) 19

[3.1 Структура разрабатываемого приложения](#_Toc26404828) 19

[4 Верификация полученных результатов](#_Toc26404827) 24

[4.1 Основы модульного тестирования](#_Toc26404828) 24

[4.2 Тестирование пользовательского интерфейса](#_Toc26404828) 24

[4.3 Сравнение полученных результатов](#_Toc26404828) 28

Заключение 32

Список использованных источников33

Приложение А Листинг программы 34

Приложение Б Чертеж кронштейна 43

**ВВЕДЕНИЕ**

В XXI веке, когда идет бурное развитие как науки, так и техники, уже сложно, а порой и нереально представить проектирование изделий и конструкций без САПР. Трудности в разработке промышленных изделий поставили инженеров перед необходимостью сочетания эффективных методов изучения особенностей поведения изделий с сочетанием реального прототипа. Необходимость внедрения в производство сложнейшей техники в короткие сроки приводит к созданию систем автоматизированного проектирования. Важную роль в этих системах играет расчет на прочность.

В основе любого расчета на прочность лежит расчетная схема, включающая в себя геометрию конструкции и действующие на нее нагрузки (механические и температурные). Естественно, что при создании расчетной схемы сложной конструкции прибегают к некоторой идеализации ее формы, при этом степень этой идеализации влияет на достоверность результатов расчета.

Метод конечных элементов (МКЭ) превратился в ключевую, незаменимую технологию в моделировании и симуляции современных инженерных систем в различных областях, таких как жилье, транспорт, связь и т.д. В строительстве таких инженерных систем, инженеры и дизайнеры проходят через сложный процесс моделирования, симуляции, визуализации, анализа, проектирования, создания прототипов, тестирования и, наконец, изготовления.

Достаточно много работы требуется проделать до изготовления конечного продукта или системы. Например, обеспечить работоспособность готового продукта, также хорошо, как экономическую эффективность. Поэтому техники, связанные с моделированием и симуляцией, быстро и эффективно набирают все более важную роль, в результате чего увеличивается количество отраслей, в которых применяется МКЭ.

Хорошее понимание этих методов играет важную роль в построении передовых инженерных систем быстрым и экономически эффективным способом.

**1 ОБЗОР ЧИСЛЕННЫХ МЕТОДОВ МОДЕЛИРОВАНИЯ В**

**МЕХАНИКЕ**

* 1. **Существующие численные методы**

При выполнении различных расчётов в математике, механике и других областях на практике применяют как аналитические, так и численные методы решения поставленных задач. Применение аналитических методов требует высокого уровня математической подготовки специалистов, в частности инженеров. Преимуществом аналитических методов является их точность, однако их применение ограничено относительно простыми математическими моделями, то есть в задачах механики применение ограничено достаточно простыми геометрическими моделями.

Численные методы (вычислительные методы, методы вычислений) –раздел вычислительной математики, изучающий приближенные способы решения типовых математических задач, которые либо не решаются, либо трудно решаются точными аналитическими методами. Примерами типовых задач являются численное решение уравнений, численные дифференцирование и интегрирование и др.

Выделяют следующие численные методы моделирования в механике:

– метод конечных разностей;

– метод граничных элементов;

– метод конечных элементов.

**1.2 Метод конечных разностей**

Метод конечных разностей (МКР) является одним из методов, используемых для решения дифференциальных уравнений, которые трудно или невозможно решить аналитически. Основная формула:

 (1.1)

Можно использовать вышеприведенное уравнение для дискретизации уравнения в частных производных (ДУЧП) и реализовать численный метод для решения ДУЧП. Например, если требуется вычислить численные решения уравнения конвекции-дисперсии-сорбции:

 (1.2)

где эффективность столбца бесконечна (то есть с *Di* равным ноль и *cF* = *fi*(*c1*, *c2*, ...)), уравнение может быть заменено уравнением конечных разностей (уравнение 2.3):

 (1.3)

где *n* и *j* - временные и пространственные индексы соответственно, а *Δt* и *Δx* - временные и пространственные приращения. Для этого преобразования область решения рассматривается как неограниченная прямоугольная область размером 0 ≤ *x* ≤ 1 и *t* ≥ 0, как показано на рисунке 1.1[1].

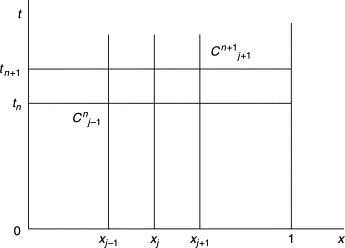


Рисунок 1.1 – Рассматриваемая прямоугольная область

МКР – это очень универсальный метод моделирования, который сравнительно прост для понимания и реализации пользователями.

**1.3 Метод граничных элементов**

Граничные интегральные уравнения являются классическим инструментом для анализа краевых задач для дифференциальных уравнений в частных производных. Термин «метод граничных элементов» (МГЭ) обозначает любой метод приближенного численного решения этих граничных интегральных уравнений. Приближенное решение краевой задачи, полученное с помощью МГЭ, отличается тем, что оно является точным решением дифференциального уравнения в области и параметризовано конечным набором параметров, находящихся на границе.

***1.3.1*** МГЭ имеет некоторые преимущества перед другими численными методами, такими как МКЭ или МКР:

* только граница области должна быть дискретизирована, особенно в двух измерениях, где граница является просто кривой, это позволяет осуществить очень простой ввод и хранение данных для метода;.
* внешние проблемы с неограниченными областями, но ограниченными границами легко решаются;
* в некоторых ситуациях физические данные задаются не решением во внутренней части области, а скорее граничными значениями решения или его производными, эти данные могут быть получены непосредственно из решения граничного интегрального уравнения, тогда как граничные значения, полученные из решений МКЭ, в общем случае не являются очень точными;
* решение внутри области аппроксимируется с достаточно высокой скоростью сходимости и, более того, такая же скорость сходимости имеет место для всех производных любого порядка внутри области решения.

***1.3.2*** Некоторые основные трудности с МГЭ следующие:

* граничные интегральные уравнения требуют явного фундаментального знания решения дифференциальных уравнений – это доступно только для линейных уравнений в частных производных с постоянными или некоторыми конкретно переменными коэффициентами.
* для заданной краевой задачи существуют различные граничные интегральные уравнения и каждому из них соответствует несколько методов численного приближения. Таким образом, МГЭ требует выбора нескольких вариантов, чтобы оценить различные возможности, поэтому нужно много математического анализа;
* причиной сложности математического анализа является то, что граничный интегральные уравнения часто являются неординарными интегральными уравнениями Фредгольма второго рода[2].

**1.4 Метод конечных элементов**

***1.4.1*** Метод конечных элементов (МКЭ) – это метод приближённого численного решения физических задач. В его основе лежат две главные идеи: дискретизация исследуемого объекта на конечное множество элементов и кусочно-элементная аппроксимация исследуемых функций. Историческими предшественниками МКЭ были различные методы строительной механики и механики деформируемого твёрдого тела, использующие дискретизацию. Ещё Пуассон в начале 19 века предлагал рассматривать сплошную среду как систему конечных объёмов.

Основная идея метода конечных элементов – найти решение сложной задачи, заменив её более простой. Поскольку актуальная задача заменена на более простую в поиске решения находится только приблизительное решение, а не точное. Существующих математических инструментов будет недостаточно, чтобы найти точное решение (а иногда даже приблизительное решение) большинства практических задач. Таким образом, при отсутствии какого-либо другого удобного способа найти даже приближенное решение данной задачи, следует предпочесть метод конечных элементов. Более того, в МКЭ, часто будет возможность улучшить или уточнить приближенное решение.

Главные достоинства МКЭ:

– исследуемые объекты могут иметь любую форму и различную физическую природу – твёрдые деформируемые тела, жидкости, газы, электромагнитные среды;

– конечные элементы могут иметь различную форму, в частности криволинейную, и различные размеры;

– можно исследовать однородные и неоднородные, изотропные и анизотропные объекты с линейными и нелинейными свойствами;

– можно решать как стационарные, так и нестационарные задачи;

– можно решать контактные задачи;

– можно моделировать любые граничные условия;

– вычислительный алгоритм, представленный в матричной форме, формально единообразен для различных физических задач и для задач различной размерности, что удобно для компьютерного программирования;

– на одной и той же сетке конечных элементов можно решать различные физические задачи, что облегчает анализ связанных задач;

– разрешающая система уравнений имеет экономичную разреженную симметричную ленточную матрицу «жёсткости», что ускоряет вычислительный процесс на ЭВМ;

– удобно осуществляется иерархическая дискретизация исследуемой области на подобласти с образованием суперэлементов, что позволяет эффективно использовать параллельное решение задачи.

***1.4.2*** Процедура вычислительного моделирования с использованием МКЭ, в целом, состоит из четырех шагов:

* моделирование геометрии;
* построение сетки (дискретизация);
* определение свойств материала;
* указание граничных, начальных и нагрузочных условий**.**

Построение сетки выполняется для дискретизации геометрии, то есть, разбиение на маленькие кусочки, называемые элементами. Рациональное обоснование этого может быть объяснено очень простым и логичным образом. Решение инженерной задачи будет очень сложным процессом, и будет меняться непредсказуемым образом, если использовать одну функцию по всей области задачи. Если заданная область может быть разделена на маленькие элементы, использующие набор сеток или узлов, решение внутри элемента может быть очень легко аппроксимировано с использованием простых функций, таких как полиномы. Таким образом, формируется решение для всей области.

Создание сетки – очень важная задача. Это может быть очень трудоемким процессом для аналитика, и, как правило, опытный аналитик создаст более надежную сетку для сложной задачи. Область должна быть правильно разбита на элементы конкретной формы, такие как треугольники и четырехугольники. Информация, такая как связь элементов, должна быть создан во время создания сетки для использования в дальнейшем, при формировании уравнений МКЭ.

Многие анализируемые системы состоят из более чем одного материала. Свойства материалов могут определяется либо для группы элементов, либо для каждого отдельного элемента в отдельности, если это необходимо. Для моделирования требуются различные наборы свойств материалов. Например, Модуль Юнга и модуль сдвига требуются для анализа напряжений твердых тел и структур, тогда как коэффициент теплопроводности потребуется для термического анализа.

Ввод свойств материала в препроцессор обычно прост; аналитик должен ввести данные о свойствах материала и указать, к какому региону геометрии или к каким элементам данные применяются. Однако получение этих свойств не всегда легко[3].

***1.4.3*** В методе конечных элементов исходная расчётная область дискретизируется конечными элементами. В силу размерности рассматриваемой области конечные элементы могут быть соответственно одномерными, двумерными и трёхмерными. Примеры конечных элементов расположены на рисунке 1.2. Поведение искомой функции на конечном элементе выражается через известные базисные функции (функции формы), которые приведены в формуле (1.4):

|  |  |  |
| --- | --- | --- |
|  |  | (1.4) |

где *F* – искомая функция;

*aij* – неизвестные постоянные множители, подлежащие определению;

*Nij\**– базисные функции;

*n* – общее число узлов элемента,

*si* – число степеней свободы в *i-*ом узле.

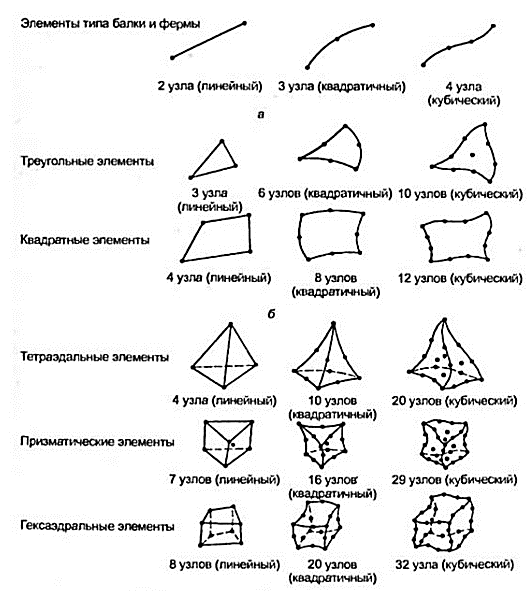


Рисунок 1.2 – Виды конечных элементов

Неизвестные постоянные множители *aij* выражаются через значения в узлах искомой функции. Данные значения называются узловыми степенями свободы, или просто степенями свободы конечного элемента. В зависимости от того, содержат ли узловые степени свободы только значения функции или ещё дополнительно содержат и значения производных, различают соответственно лагранжевы и эрмитовы конечные элементы.

Таким образом, выражая постоянные множители *aij* через значения в узлах искомой функции, формулу (1.2) можно переписать в виде формулы (1.3):

|  |  |  |
| --- | --- | --- |
|  |  | (1.5) |

где *Ni* – функция формы конечного элемента;

*Fie* – значение искомой функции в узле конечного элемента;

e – общее количество степеней свободы конечного элемента.

При выборе функций формы *Ni* необходимо руководствоваться следующим условиями:

– функции формы должны непрерывно дифференцируемы внутри, конечно, элемента столько раз, каков порядок производной, входящей в исходное дифференциальное уравнение или применяемый вариационный принцип (условие дифференцируемости или условие существования функционала вариационной задачи);

– при переходе через границы смежных элементов должна обеспечиваться непрерывность искомой функции и существование её производных (условие межэлементной непрерывности);

– при стремлении размера конечного элемента к нулю функции формы должны обеспечить возможность получения конечных значений искомой функции и её производных, входящих в вариационный функционал;

– кроме того, для существования решения, основанного на принципе минимума энергии, необходимо, чтобы реализовывалось состояние постоянной деформации, а при перемещении тела как твёрдого целого должна обеспечиваться нулевая энергия деформации;

– из соотношений (1.3) следует, что *Ni* =1 только в *i-*ом узле, и нулю – во всех остальных узлах [4].

**2 АЛГОРИТМИЧЕСКИЙ АНАЛИЗ ЗАДАЧИ**

**2.1 Постановка задачи**

Для решения поставленной задачи необходимо разработать приложение для вычисления перемещений плоского кронштейна заданной ширины, высоты и толщины при воздействии силы, направленной на узел, под углом 45 градусов. Для проверки правильности вычислений необходимо произвести аналогичный расчет в программном комплексе *ANSYS.* Необходимыми требованиями к разрабатываемому приложению являются:

– наличие графического пользовательского интерфейса, содержащий поля ввода для начальных параметров кронштейна и позволяющий выводить результаты расчета в виде графиков;

– наличие алгоритма дискретизации поверхности кронштейна на конечные элементы, другими словами, создание сетки из конечных элементов;

– наличие алгоритма, реализующий метод конечных элементов, для вычисления перемещений узлов элементов.

В качестве исходных данных в программе выступают:

– плоский кронштейн, чертеж которого приведен в приложении Б;

– начальные характеристики кронштейна, включающие в себя ширину, длину, толщину, модуль Юнга (модуль упругости) и коэффициент Пуассона;

– граничные условия в виде закреплений кронштейна в определенных узлах;

– воздействие внешних сил.

Так как кронштейн представляет собой плоскую фигуру, а закрепления и воздействие силы осуществляется в направлении только двух осей – *OX* и *OY*, то рассматриваемая задача представляет собой двумерную задачу теории упругости.

**2.2 Метод конечных элементов для решения двумерных задач теории упругости**

***2.2.1*** Для представления системы в виде двухмерного тела в основном необходимо удалить одну координату (обычно ось *z*), если, все зависимые переменные не зависят от оси *Z*, и все внешние нагрузки не зависят от координаты *Z*, и применяется только в плоскости *OX*-*OY*. Таким образом, осталась система только с двумя координатами *х* и *у*. Есть в основном два типа 2*D* тел. Один из них представляет собой твердое тело с плоским напряжением, а другой – это тело с плоской деформацией. Напряжённое состояние тела является плоским, если толщина в направлении *z* очень мала по сравнению с размерами в направлении *х* и *у*. Внешние силы прикладываются только в плоскости *x*-*y*, а напряжения – в направлении *Oz* равны нулю, как показано на рисунке 2.1. Деформация называется плоской если, толщина тела в направлении *z* очень велика по сравнению с размерами в направлениях *х* и *у*. Внешние силы прикладываются равномерно вдоль оси *z*, а движение по оси *z* в любой точке ограничено. Компоненты деформации в направлении z равны нулю, как показано на рисунке 2.2

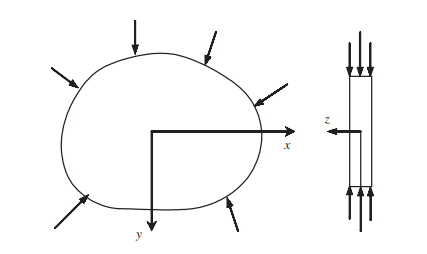


Рисунок 2.1 – Плоское напряжённое состояние

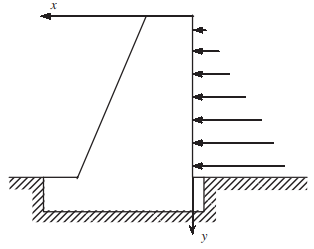


Рисунок 2.2 – Плоское деформированное состояние

В любой точке двухмерных тел имеется три соответствующих компонента деформации, которые также могут быть записаны в аналогичной векторной форме (2.1)

 (2.1)

Соотношения деформации и смещения приведены в 2.2.

 (2.2)

где *u*, *v* - компоненты смещения в направлениях *x*, *y* соответственно.

Смещения также могут быть записаны в следующей матричной форме (2.3):

 (2.3)

где вектор смещения имеет вид (2.4):

 (2.4)

и матрица дифференциального оператора получается просто из уравнения (2.2) как:

 (2.5)

Для плоско напряженных, изотропных материалов, Закон Гука для двумерных тел имеет следующую матричную форму (2.6):

 (2.6)

где *v –* коэффициент Пуассона,

E – модуль Юнга.

***2.2.2*** На следующем шаге решения находятся значения коэффициентов матриц, которые определяют связь геометрических характеристик внутри конечных элементов, вследствие чего для каждого элемента существует уникальная матрица [*B*]. Она состоит из разности координат узлов, входящих в конечный элемент (2.7).

 (2.7)

где *i*, *j*, *k* – узлы соответствующего конечного элемента.

Тогда матрицы жёсткости элементов запишутся в виде (2.8):

, (2.8)

где *i* – номер конечного элемента,

*Si* – площадь *i*-го элемента,

*h* – толщина элемента.

При перемножении следует учитывать, что данная операция не коммутативна, т.е. матрицы необходимо перемножать именно в том порядке, в котором они записаны. Крайне важно выдерживать высокую точность расчёта, в противном случае при математических операциях величина ошибки будет всё время нарастать. Итоговая матрица жёсткости каждого элемента может быть представлена в виде (2.9):

, (2.9)

где *kij* – коэффициенты влияния, а верхний индекс обозначает номер конечного элемента. Элементы справа и слева относительно главной диагонали должны быть одинаковы. Таким образом, левый нижний угол должен быть зеркальным отражением верхнего правого с осью симметрии, проходящей через главную диагональ (*…*). Так можно проверить правильность построения матрицы.

***2.2.3*** Для построения общей матрицы жёсткости приведён пример плоской конструкции на рисунке 2.3.

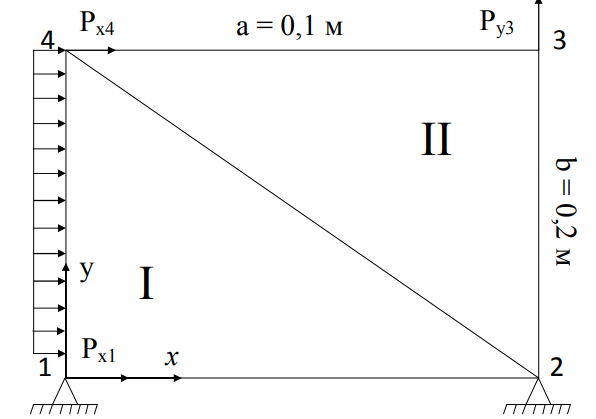


Рисунок 2.3 – Плоская конструкция

Для предоставленного примера *a* = 0,1 м; *b* = 0,2 м; *h* = 0,003 м; *E* = 2·1011 Па; *µ* = 0,33; *Py3* = 10000 Н (приложена к узлу 3); *p* = 80000 Н/м (приложена к линии 1-4); закреплены узлы 1 и 2.

Сила 

Вектора перемещений и нагрузок представлены в (2.10).

 (2.10)

Площади конечных элементов *S*1=*S*2=0.01 м2.

Для построения общей матрицы жёсткости [*K*], необходимо записать уравнения равновесия для каждого узла. Т.е. сумма всех внутренних реакций в каждом узле должна быть равна нулю. При этом следует учесть, что узлы №1 и №3 принадлежат только одному конечному элементу (соответственно первому и второму). А узлы №2 и №4 принадлежат обоим конечным элементам. В первом элементе порядок узлов №1-2-4. Поэтому для записи уравнения равновесия первого узла используются только первые две строки матрицы [*К*1]. Затем выбранные строки матрицы [*K*1] делятся на несколько квадратных матриц размерностью 2х2. Поскольку узел №1 принадлежит только одному элементу, то *kij*=*k*1, где *i*, *j* – номер строки и столбца матрицы жёсткости. Все силы, действующие в узле №1 (2.11).

 (2.11)

Нулевая матрица для перемещений третьего узла взята потому, что узел №3 не входит в первый конечный элемент. Примеры определения коэффициентов: 

Узел №2 принадлежит двум конечным элементам, поэтому его матрица жёсткости записывается сложнее. Для её записи используются третья и четвёртая строки матрицы [*К*1] и первая и вторая строки матрицы [K2], соответствующих узлу №2. Значения коэффициентов в матрицах складываются с учётом порядка узлов в элементах. Матрица в общем виде представлена в 2.12.

 (2.12)

Затем через коэффициенты матриц К1 и К2 матрица примет вид (2.13):

 (2.13)

Нули в первой матрице записаны потому, что узел №1 не входит во второй конечный элемент, аналогично, нули в третьей матрице объясняются тем, что узел №3 не входит в первый конечный элемент. Аналогично получаются уравнения для узлов №3 и №4. Тогда разрешающая система уравнений запишется в виде (2.14):

 (2.14)

Для получения разрешающей системы уравнений необходимо умножить матрицу [*K*] на вектор перемещений {*q*} и приравнять результат к вектору {*P*}. Из каждой строки матрицы [*K*] получается одно уравнение. Для этого нужно последовательно перемножить коэффициенты в одной строке матрицы [*K*] на значения строк вектора [*q*], сложить эти произведения и приравнять их к значению вектора {*P*} той же строки, что и в матрице [*K*]. Для нахождения неизвестных перемещений узлов достаточно решить уравнения, в которых перемещения не равны нулю. Для выбранной схемы (2.14) это последние четыре уравнения, поскольку в первых четырёх уравнениях перемещения равны нулю. А затем из первых четырёх уравнений, за счёт подстановки в них найденных перемещений, можно определить неизвестные реакции. Умножив по правилам матрицу [*К*] на вектор {*q*}, получим (2.15):

 (2.15)

Система из четырёх уравнений с четырьмя неизвестными может быть решена методом Гаусса. После решения системы уравнений, перемещения для системы найденых[5].

**3 ПРОГРАММНАЯ РЕАЛИЗАЦИЯ ЗАДАЧИ**

**3.1 Структура разрабатываемого приложения**

***3.1.1*** Для реализации анализа напряженно-деформированного состояния плоского кронштейна, а именно значений перемещения, в приложении разработаны необходимые классы. Полный код приложения приведен в приложении А.

Для полного и правильного функционирования программного комплекса реализованы следующие классы:

– *Node –*класс, реализующий такой объект, как «узел»;

– *TriangleStiffness –* класс, реализующий такой объект, как «конечный элемент» треугольной формы;

– *Finite2DGenerator –* класс, реализующий алгоритм метода конечных элементов;

– *MainWindow –*класс, служащий для отображения пользовательского графического интерфейса, обработки введенных данных и вывода полученных результатов в виде графиков.

– StiffnessData – класс, приводящий данные к приемлемому, для обработки МКЭ, виду, данный класс хранит информацию о узлах, элементах и свойствах материала.

***3.1.2*** Для реализации объекта «узел» реализован класс «*Node*». В нем выделяют определенные поля, однозначно характеризующие принадлежность класса к объекту «узел». Описание полей приведено в таблице 3.1.

Таблица 3.1 – **Поля, описывающие класс «*Node*»**

|  |  |  |
| --- | --- | --- |
| **Поле** | **Описание** | **Тип данных** |
| *num* | Уникальный индекс узла | *int* |
| *X* | Координаты узла по оси *X* | *float* |
| *Y* | Координаты узла по оси *Y* | *float* |
| *moving* | Определяет закреплен ли узел | *bool* |

***3.1.3*** Для реализации объекта «конечный элемент» реализован класс «*TriangleStiffness*». В нем выделяют ряд методов и определенные поля, однозначно характеризующие принадлежность класса к объекту «конечный элемент».

Класс *TriangleStiffness* содержит следующие методы:

– *\_\_define\_nodes\_order –* метод, определяющий порядок следования узлов (против часовой стрелки) для применения в матрице геометрических характеристик элемента;

– *get\_stiffness\_matrix –* метод, позволяющий определить локальную матрицу жесткости для данного конечного элемента;

– *nodes –*метод, возвращающий список узлов, принадлежащих текущему конечному элементу.

Описание полей класса «*TriangleStiffness*» приведено в таблице 3.2.

Таблица 3.2 – **Поля, описывающие класс «***TriangleStiffness***»**

|  |  |  |
| --- | --- | --- |
| **Поле** | **Описание** | **Тип данных** |
| *a* | 1 из 3 узлов конечного элемента | *Node* |
| *b* | 2 из 3 узлов конечного элемента | *Node* |
| *c* | 3 из 3 узлов конечного элемента | *Node* |
| *\_stiffness* | Давление, подаваемое на поверхность конечного элемента | *np.array* |

***3.1.4*** Для реализации алгоритма метода конечных элементов создан класс «*Finite2DGenerator*». В нем выделяют ряд методов и определенные поля, однозначно характеризующие принадлежность класса.

Класс *Finite2DGenerator* содержит следующие методы:

– *get\_stiffness\_matrix –*метод, реализующий создание глобальной матрицы жесткости, посредством итерации по узлам системы, определения элементов которые содержат данный узел и определения позиции строк в глобальной матрице жёсткости;

– *\_append\_to\_matrix –*приватный метод, реализующий добавление строк локальной матрицы жесткости каждого элемента к глобальной;

– \_get\_node\_in\_element\_indexes – статический метод, который находит все элементы содержащие узел, переданный в качестве первого параметра, итерация осуществляется по элементам, которые переданы вторым аргументом.

– \_extract\_triangles – статический метод, который формирует конечные элементы на основе переданных массивов координат и точек треугольника.

– calculate\_moving – метод, который вычисляет перемещения узлов, метод удаляет строки из глобальной матрицы, которые не требуются при расчётах, вычисление системы полученных линейных уравнений производится посредством библиотеки scipy и её модуля linalg который содержит функцию solve, применяющая метод гаусса для решения системы уравнений, поэтому необходимо учитывать сингулярность матрицы.

Описание полей класса «*Finite2DGenerator*» приведено в таблице 3.3.

Таблица 3.3 – **Поля, описывающие класс «***Finite2DGenerator***»**

|  |  |  |
| --- | --- | --- |
| **Поле** | **Описание** | **Тип данных** |
| *dimension* | Константа со значением «2», которое означает двумерное рассматриваемое пространство данной задачи. | *int* |
| *data* | Необходимые для расчёта данные – порядок узлов, нагрузки и т.д | *StiffnessData* |
| *sorted\_nodes* | Упорядоченный список узлов | *np.array* |
| *stiffness\_elements* | Список конечных элементов треугольной формы | *list* |
| *m\_s* | Число степеней свободы, которое определяется количеством узлов и рассматриваемого пространства | *int* |
| *stiffness\_matrix* | Матрица жёсткости | *np.array* |
| *calculated\_moving* | Рассчитанные перемещения | *np.array* |
| *D* | Матрица описывающая закон Гука для двумерных тел | *np.array* |
| *h* | Толщина кронштейна | *float* |

***3.1.5*** Для хранения и формирования входных данных создан класс «*StiffnessData*». В нем выделяют ряд методов и определенные поля, однозначно характеризующие принадлежность класса.

Класс *StiffnessData* содержит следующие методы:

– *is\_prepared –*метод, который проверяет наличие всех необходимых данных для рассчёта задачи, вызывается прежде, чем приступать к использованию данных внутри класса;

– *d* *–* метод который формирует матрицу закона Гука, при формировании матрицы сохраняет её в переменной класса, для того, чтобы избежать дальнейших однотипных перерасчётов;

– *set\_fixing* – метод который позволяет закрепить узлы системы, по умолчанию все узлы обрабатываемые классом, могут перемещатся.

– *set\_load* – метод который позволяет задать точное значение силы действующее на узел.

– *set\_undefined\_load* – метод который позволяет задать неопределённо значение силы, если оно неизвестно, то есть представлена, в виде буквы в векторе сил.

– *get\_tri\_x* – метод позволяющий извлечь все точки *x* узлов решаемой системы, в первоначальном порядке.

– *get\_tri\_y* – метод позволяющий извлечь все точки *y* узлов решаемой системы, в первоначальном порядке.

– *create\_nodes* – метод который создаёт объекты узлов, для набора точек содержащихся внутри геометрии, переданных аргументов в качестве списка для данного метода.

– *get\_raw\_nodes* – возвращает узлы системы в первоначальном порядке, для использования в графическом отображении узлов и формирования элементов.

– *\_find\_duplicated\_elements* – статический метод возвращающий индексы продублированных элементов в списке, используется для замены продублированных узлов на один объект в памяти компьютера.

Описание полей класса «*StiffnessData*» приведено в таблице 3.4.

Таблица 3.4 – **Поля, описывающие класс «***StiffnessData***»**

|  |  |  |
| --- | --- | --- |
| **Поле** | **Описание** | **Тип данных** |
| *loads* | Список сил действующих на узлы, в порядке следования узлов | *np.array* |
| *moving* | Список перемещений на узлах, в порядке следования узлов | *np.array* |
| *points* | Список точек, для формирования узлов и элементов | *list* |
| *nodes* | Отсортированный против часовой стрелки список узлов | *list* |
| *poisson* | Коэффициент Пуассона | *float* |
| *thickness* | Толщина элемента | *float* |
| *young* | Модуль Юнга | *float* |
| *tri* | Объект из библиотеки scipy, предоставляющий триангуляцию | *Delaunay* |
| *\_\_nodes* | Список узлов в порядке следования точек, используемых для формирования узлов | *list* |
| *D* | Матрица сформированная из закона Гука | *np.array* |

***3.1.6*** *Класс «MainWindow»* служит для отображения пользовательского графического интерфейса, считывания введенных пользователем в специальные поля данных и на их основе получения определенного результата с последующей его интерпретацией в виде различного рода графиков. Именно в данном классе создается объект класса «*Finite2DGenerator*» с последующим вызовом его методов для решения задачи анализа напряженно-деформированного состояния кронштейна.

Графический интерфейс реализован с использованием библиотеки *PyQt*5.

*PyQt* объединяет кроссплатформенную среду приложений *Qt C*++ и кроссплатформенный интерпретируемый язык *Python*.

*Qt* – это больше, чем инструментарий *GUI*. Он включает в себя абстракции сетевых сокетов, потоков, *Unicode*, регулярных выражений, баз данных *SQL*, *SVG*, *OpenGL*, *XML*, полнофункциональный веб-браузер, справочную систему, мультимедийную среду, а также богатую коллекцию графических виджетов.

Классы *Qt* используют механизм сигнал/слот для связи между объектами, который является безопасным по типу, но слабо связанным, что облегчает создание программных компонентов для многократного использования.

*Qt* также включает *Qt Designer*, дизайнер графического пользовательского интерфейса. *PyQt* может генерировать код *Python* из *Qt Designer*. Также возможно добавить новые элементы управления *GUI*, написанные на *Python*, в *Qt Designer*.

*Python* – это простой, но мощный объектно-ориентированный язык. Его простота облегчает изучение, но его мощь означает, что можно создавать большие и сложные приложения. Его интерпретируемая природа означает, что программисты на *Python* очень продуктивны, потому что нет цикла разработки / компиляции / компоновки / запуска.

Большая часть возможностей *Python* заключается в его всеобъемлющем наборе модулей расширения, обеспечивающих широкий спектр функций. Модули расширения обычно реализуются на *Python*, *C* или *C*++. Используя такие инструменты, как *SIP*, довольно просто создать модуль расширения, который инкапсулирует существующую библиотеку *C* или *C*++. При таком использовании *Python* может стать клеем для создания новых приложений из установленных библиотек.

*PyQt* сочетает в себе все преимущества *Qt* и *Python*. Программист обладает всеми возможностями *Qt*, но может использовать его с простотой *Python*.

**4 ВЕРИФИКАЦИЯ ПОЛУЧЕННЫХ РЕЗУЛЬТАТОВ**

**4.1 Основы модульного тестирования**

Модульное тестирование – это тестирование программы на уровне отдельно взятых модулей, функций или классов. Цель модульного тестирования состоит в выявлении локализованных в модуле ошибок в реализации алгоритмов, а также в определении степени готовности системы к переходу на следующий уровень разработки и тестирования. Модульное тестирование проводится по принципу "белого ящика", то есть основывается на знании внутренней структуры программы, и часто включает те или иные методы анализа покрытия кода.

Модульное тестирование обычно подразумевает создание вокруг каждого модуля определенной среды, включающей заглушки для всех интерфейсов тестируемого модуля. Некоторые из них могут использоваться для подачи входных значений, другие для анализа результатов, присутствие третьих может быть продиктовано требованиями, накладываемыми компилятором и сборщиком.

На уровне модульного тестирования проще всего обнаружить дефекты, связанные с алгоритмическими ошибками и ошибками кодирования алгоритмов, типа работы с условиями и счетчиками циклов, а также с использованием локальных переменных и ресурсов [6].

Так как доступ к приватным (*private*) методам класса извне не возможен, не используя механизмы рефлексии, то и тестировать такие методы модульными тестами нельзя. Остаются только публичные (*public*) методы. Не нужно знать о приватных методах, которые используются публичными. Это означает, что при правильном выполнении публичных методов все использующиеся скрытые внутри класса методы работают корректно и отдельно их тестировать не нужно и нельзя из-за модификаторов доступа.

**4.2 Тестирование пользовательского интерфейса**

***4.2.1*** Для работы приложения необходимо наличие интерпретатора *Python* с подключенными библиотеками. Для запуска программы необходимо запустить файл «*main.py*» с помощью интерпретатора

После запуска программы открывается главное окно программы, где пользователь предоставляется возможность ввести необходимые для анализа данные.

Пример внешнего вида основного пользовательского графического интерфейса программы показан на рисунке 4.1.

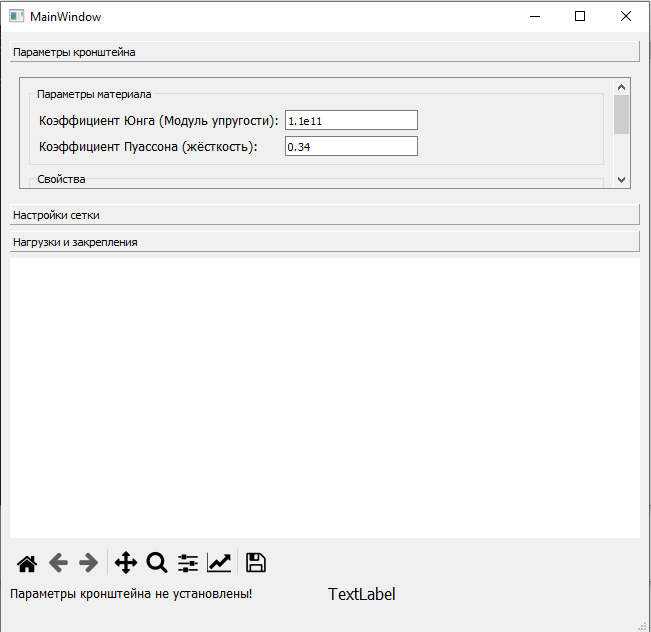


Рисунок 4.1 – Внешний вид основного окна программы

***4.2.2*** Вверху, на первой вкладке графического интерфейса расположены поля ввода начальных параметров кронштейна.

В данном окне пользователю доступны следующие поля ввода данных:

– «Толщина (мм)» – поле ввода толщины кронштейна в мм, принимающее в качестве верных значений – числа;

– «Модуль Юнга (Па)» – поле ввода модуля упругости для материала, из которого состоит кронштейн;

– «Коэффициент Пуассона» – поле ввода коэффициента Пуассона для материала, из которого состоит кронштейн;

– «Значение силы (Н)» – поле ввода значения силы, которое оказывает воздействие на узел кронштейна.

Внешний вид полей ввода начальных параметров плоского кронштейна представлен на рисунке 4.2.

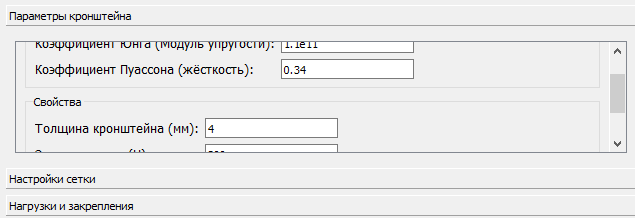


Рисунок 4.2 – Внешний вид полей ввода начальных параметров кронштейна

***4.2.3*** На второй вкладке графического интерфейса расположены поле ввода начальных параметров сетки, состоящей из конечных элементов, и кнопка расчёта.

В данном окне пользователю доступны следующие поля ввода данных:

– «Размер сетки» – поле ввода, отвечающее за количество конечных элементов, параметр задаёт максимальное значение длинны ребра конечного элемента;

Также здесь расположена следующая кнопка:

– кнопка «Сгенерировать и произвести расчёты» – кнопка, отвечающая за создание сетки из конечных элементов и ее последующего отображения, а также вывода карты деформаций, рассчитанных на основе метода конечных элементов;

Пример внешнего вида полей ввода параметров сетки и кнопок приведен на рисунке 4.3.

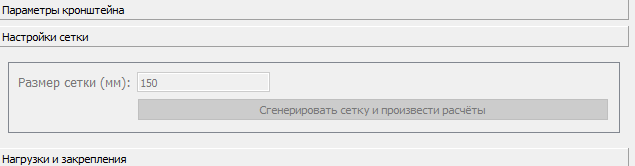


Рисунок 4.3 – Внешний вид полей ввода параметров сетки

***4.2.4*** При первом запуске программы все поля ввода инициализируются начальными значениями, а кнопка «Сгенерировать и произвести расчёты» остается неактивной. Если пользователю необходимо изменить начальные данные, то он может сделать это при помощи ввода своих данных в соответствующие специальные поля ввода.

Для построения сетки пользователю достаточно указать максимальный размер элемента сетки, после чего программа определит количество элементов, на которое будет разбита система. Затем пользователю необходимо нажать соответствующую кнопку, после чего отобразится сетка в виде графика, на котором будут указаны все узлы, в том числе и закрепленные узлы. Результат построения сетки изображен на рисунке 4.4.

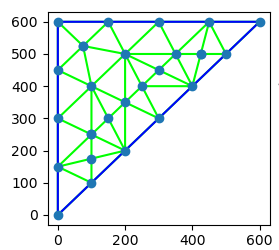


Рисунок 4.4 – Результат построения сетки

Результат вычисления перемещений приведен на рисунке 4.5.

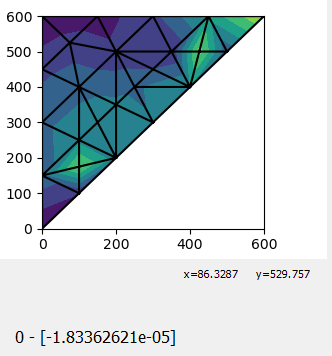


Рисунок 4.5 ­– Результат нахождения деформаций кронштейна

Внизу, под графиком распределения деформаций, указано максимальное значение перемещений.

**4.3 Сравнение полученных результатов**

Для сравнения полученных результатов воссоздадим модель плоского кронштейна в программном комплексе *ANSYS Mechanical APDL* и рассчитаем перемещения. В качестве начальных параметров кронштейна используются параметры, указанные в созданном программе. В результате воссоздания модели получится кронштейн из меди с толщиной 4 мм. Результат воссоздания модели приведен на рисунке 4.6.

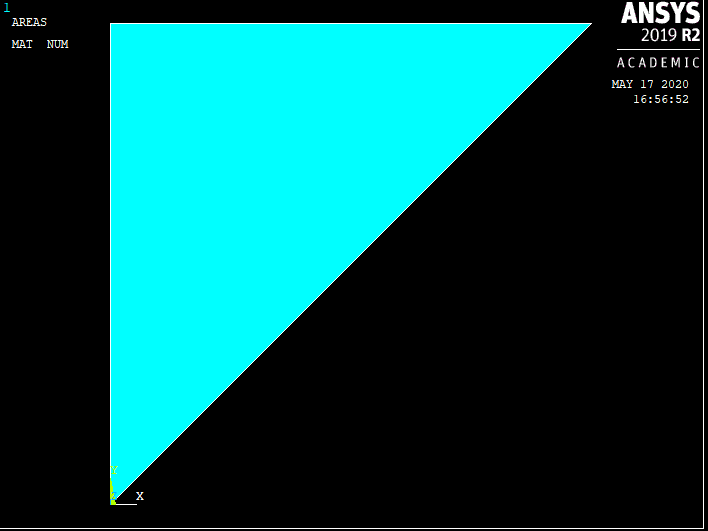


Рисунок 4.6 – Модель кронштейна, созданная в *ANSYS*

Для расчётов в верификационном программном комплексе, необходимо указать информацию о свойствах материала. По условию задачи материал является изотропным. Для задания свойств в среде *ANSYS* необходимо перейти в «*Preprocessor->Material Props->Material Models*» в открывшемся окне необходимо выбрать «*Structural->Linear->Elastic->Isotropic*», в следующем окне необходимо ввести модуль Юнга и коэффициент Пуассона, в соответствующем порядке. Установка данного параметра в *ANSYS Mechanical APDL* приведена на рисунке 4.7.

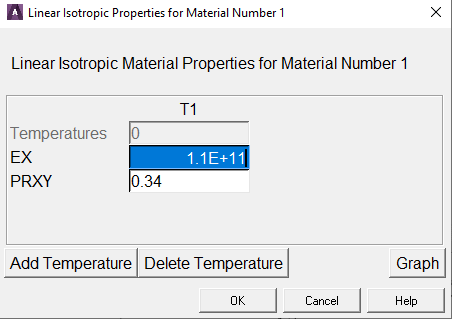


Рисунок 4.7 – Установка свойств материала в *Ansys*

Далее, для проверки верности расчетов необходимо задать значение силы на узел кронштейна. Для этого в проект решения *ANSYS* требуется перейти в «*Loads->Define Loads->Apply->Sctructural->Force*» далее выбирается «*On Nodes*» после чего выделяются узлы и вводится значение силы. Результат установки силы изображен на рисунке 4.8.

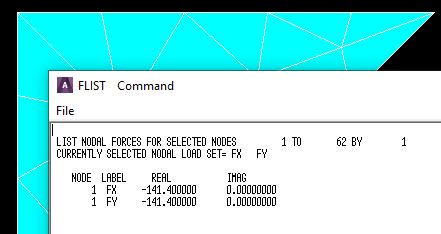


Рисунок 4.8 – Установленное значение силы на узел

После определения значения силы, оказываемой, на кронштейн необходимо установить закрепление. Для этого в проект решения *ANSYS* требуется выбрать «*Displacement*» и указать точки, в которых будут происходить закрепление, затем установить перемещения по осям *Ox* и *Oy* в нуль. Результат установки закрепления представлен на рисунке 4.9.

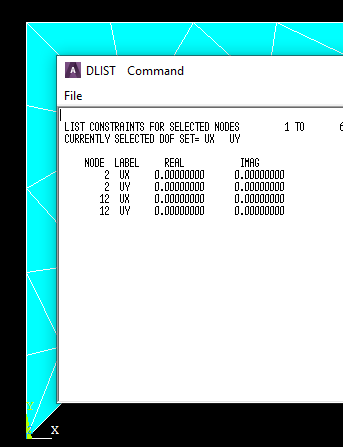


Рисунок 4.9 – Установленное закрепление в точках кронштейна

После всего этого необходимо рассчитать деформацию кронштейна. В результате будет выведен результат со значениями деформации по обеим осям. Результат нахождения максимального значения деформации приведен на рисунке 4.10.

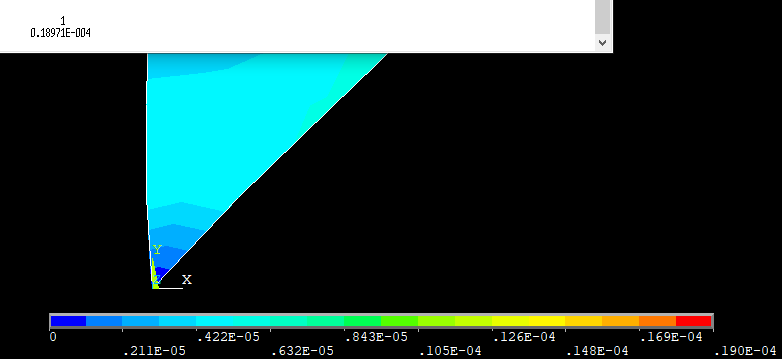


Рисунок 4.10 – Результат нахождения деформаций в программе *ANSYS*

Результат нахождения деформаций в разработанном программном комплексе изображен на рисунке 4.11.

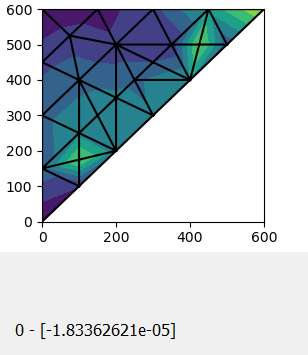


Рисунок 4.11 – Результат анализа деформаций в разработанном

программном комплексе

После нахождения деформаций при помощи программы *ANSYS* необходимо найти погрешность вычислений. Для этого требуется сравнить максимальное значение изгиба из программы *ANSYS* и разработанной программы. В результате получим, что погрешность измерений находится в пределах допустимой нормы погрешности.

**ЗАКЛЮЧЕНИЕ**

Разработан программный комплекс для реализации анализа напряженно-деформированного состояния плоской конструкции кронштейн при помощи метода конечных элементов на языке программирования *Python.* Для работы с матрицами внутри приложения использована библиотека для матричных вычислений *NumPy.* Для разработки графического интерфейса использовался фреймворк *PyQt.*

Данный программный комплекс полностью удовлетворяет поставленным требованиям и выполнят все поставленные задачи. Для верификации программы были произведены аналогичные расчеты в программном комплексе, использующем метод конечных элементов, *ANSYS*.

Полученная программа позволяет пользователю на основе вводимых данных произвести расчет деформаций плоского кронштейна под воздействием силы, поданной на узел.

Полученные результаты вычислений в разработанном приложении и в системе *ANSYS* практически совпадают. Процент погрешности не превышают допустимые нормы.

**Список использованных источников**

1. Finite Difference Method: *Sciencedirect.com*. – Электрон. данные. ­–Режим доступа: <https://www.sciencedirect.com/topics/engineering/>finite-difference-method. – Дата доступа: 10.05.2020.

2. Метод граничных элементов: Файловый архив для студентов. *Studfil-*

*es.* – Электрон. данные. – Режим доступа: https://studfile.net/preview/725119/pa-ge:11/. – Дата доступа: 10.05.2020.

3. Liu G.R. The finite element method. A practical course: учебное пособие / Quek S.S. – Elsevier Science Ltd, 2013. – 348 с.

4. Метод конечных элементов: *Samgtu.ru.* – Электрон. данные. – Режим доступа: http://meh.samgtu.ru/sites/meh.samgtu.ru/files/fokin.pdf. – Дата доступа: 10.05.2020.

5. Гвоздев А.С. Основы метода конечных элементов: методическое пособие / Мелентьев В.С., Уланов А.М. – Самар. гос. аэрокосм. ун-та, 2017. – 20 с.

6. Основы тестирования программного обеспечения: Файловый архив для студентов. Studfiles. – Электрон. данные. – Режим доступа: https://studfile.net/preview/4494386/. – Дата доступа: 10.05.2020.

**ПРИЛОЖЕНИЕ A**

(обязательное)

**Листинг программы**

**Код StiffnessData.py:**

from scipy.spatial import Delaunay  
from math import sqrt, ceil  
from libs.Node import Node  
from libs.TriangleStiffness import clockwise\_angle\_and\_distance, set\_clockwise\_origin  
  
import numpy as np  
  
  
def split\_via\_delaunay(points, max\_length):  
 tri = Delaunay(points)  
 *# get set of edges from the simpleces* edges = set()  
 for simplex in tri.simplices:  
 *# simplex is one triangle: [ 4 5 17]* edges.add((simplex[0], simplex[1]))  
 edges.add((simplex[1], simplex[2]))  
 edges.add((simplex[0], simplex[2]))  
 *# check if all edges are small enough  
 # and add new points if not* is\_finished = True  
 for edge in edges:  
 p1, p2 = edge  
 [x1, y1] = points[p1]  
 [x2, y2] = points[p2]  
 length = sqrt((x2 - x1) \* (x2 - x1) + (y2 - y1) \* (y2 - y1))  
 if length > max\_length:  
 is\_finished = False  
 *# split in how many pieces?* n\_pieces = ceil(length / max\_length)  
 for piece in range(1, int(n\_pieces)):  
 points.append([x1 + piece / float(n\_pieces) \* (x2 - x1), y1 + piece / float(n\_pieces) \* (y2 - y1)])  
 if not is\_finished:  
 split\_via\_delaunay(points, max\_length)  
  
  
class StiffnessData:  
 def \_\_init\_\_(self):  
 self.loads = None  
 self.moving = None  
 self.points = None  
 self.nodes = None  
 self.poisson = None  
 self.thickness = None  
 self.young = None  
 self.tri = None  
 self.\_\_nodes = None  
 self.D = None  
  
 def is\_prepared(self):  
 return self.nodes is not None and self.thickness is not None \  
 and self.young is not None and self.poisson is not None  
  
 def d(self):  
 if self.D is None:  
 if self.is\_prepared():  
 hooke = np.array([  
 [1, self.poisson, 0],  
 [self.poisson, 1, 0],  
 [0, 0, (1 - self.poisson) / 2]  
 ])  
 self.D = (self.young / (1 - self.poisson \*\* 2)) \* hooke  
 return self.D  
  
 def define\_nodes\_mesh(self, \_points, size=None, split=True):  
 if split:  
 split\_via\_delaunay(\_points, size)  
 self.tri = Delaunay(\_points)  
 self.create\_nodes(\_points)  
 self.points = \_points  
 self.loads = np.zeros((2 \* len(self.nodes), 1))  
 self.moving = np.ones((2 \* len(self.nodes), 1))  
  
 def set\_fixing(self, point):  
 index = self.nodes.index(point)  
 self.moving[2 \* index], self.moving[2 \* index + 1] = 0, 0  
 self.set\_undefined\_load(point)  
  
 def set\_load(self, point, value):  
 index = self.nodes.index(point)  
 if not np.isnan(value):  
 value \*= np.cos(np.radians(45))  
 self.loads[2 \* index][0], self.loads[2 \* index + 1][0] = value, value  
  
 def set\_undefined\_load(self, point):  
 self.set\_load(point, np.nan)  
  
 def get\_tri\_x(self):  
 return [n.x for n in self.\_\_nodes] if self.tri is not None else None  
  
 def get\_tri\_y(self):  
 return [n.y for n in self.\_\_nodes] if self.tri is not None else None  
  
 def create\_nodes(self, points):  
 self.\_\_nodes = [Node(p) for p in points]  
 duplicated\_points = self.\_find\_duplicated\_elements(self.\_\_nodes)  
 for k in duplicated\_points: *# change duplicated to one object* for i in duplicated\_points[k]:  
 self.\_\_nodes[i] = self.\_\_nodes[k]  
 *# self.nodes = sorted(list(set(self.\_\_nodes)), key=lambda point: [point.y, point.x])* set\_clockwise\_origin([0, 0]) *# TODO: fake* self.nodes = StiffnessData.mesh\_sort(self.\_\_nodes)  
 for i in range(len(self.nodes)):  
 self.nodes[i].num = i  
  
 def get\_raw\_nodes(self):  
 return self.\_\_nodes  
  
 @staticmethod  
 def \_find\_duplicated\_elements(array):  
 excluded = [] *# for indexes which already appended to dictionary* indexes = {}  
 for i in range(len(array) - 1):  
 if i in excluded:  
 continue  
 for j in range(i + 1, len(array)):  
 if array[i] == array[j]:  
 if i in indexes:  
 indexes[i].append(j)  
 else:  
 indexes[i] = [j]  
 excluded.append(j)  
 return indexes  
  
 @staticmethod  
 def mesh\_sort(nds):  
 nodes = sorted(list(set(nds)), key=clockwise\_angle\_and\_distance)  
 nn = nodes[0]  
 nodes.remove(nn)  
 nodes.append(nn)  
 nodes.reverse()  
 return nodes

**Код MainWindow:**

class MainWindow(QMainWindow):  
 def \_\_init\_\_(self, parent=None):  
 super(MainWindow, self).\_\_init\_\_(parent=parent)  
 self.ui = Ui\_MainWindow()  
 self.ui.setupUi(self)  
 self.figure = Figure()  
 self.figureCanvas = FigureCanvas(self.figure)  
 self.plotToolbar = NavigationToolbar(self.figureCanvas, self)  
 self.data = StiffnessData()  
  
 def setup\_ui(self):  
 self.ui.plotLayout.addWidget(self.figureCanvas)  
 self.ui.plotLayout.addWidget(self.plotToolbar)  
 self.setup\_handlers()  
  
 def setup\_handlers(self):  
 self.ui.setupBraketParametersBtn.clicked.connect(self.bracket\_parameters\_setup)  
 self.ui.meshGenerateBtn.clicked.connect(self.mesh\_parameters\_setup)  
 self.ui.setupLFparametersBtn.clicked.connect(self.load\_fix\_parameters\_setup)  
 self.ui.loadTypeBox.currentIndexChanged.connect(self.load\_fix\_box\_type\_changed)  
 self.ui.fixTypeBox.currentIndexChanged.connect(self.load\_fix\_box\_type\_changed)  
  
 def draw\_plot(self):  
 axis = self.figure.add\_subplot(121) if len(self.figure.get\_axes()) == 0 else self.figure.get\_axes()[0]  
 bracket = default\_bracket.copy()  
 bracket.append(bracket[0])  
 bracket = np.array(bracket)  
 axis.plot(bracket[:, 0], bracket[:, 1], c=**"blue"**)  
  
 def draw\_mesh(self):  
 x = self.data.get\_tri\_x()  
 y = self.data.get\_tri\_y()  
 if x is not None and y is not None:  
 axis = self.figure.get\_axes()[0] if len(self.figure.get\_axes()) > 0 else self.figure.add\_subplot(121)  
 axis.triplot(x, y, self.data.tri.simplices, c=**"lime"**)  
 axis.plot(x, y, **'o'**)  
 self.figure.canvas.draw()  
  
 def calculate(self, p):  
 if self.data.is\_prepared():  
 self.data.set\_fixing(Node(default\_bracket[0]))  
 self.data.set\_fixing(Node(default\_bracket[1]))  
 self.data.set\_load(Node(default\_bracket[2]), p)  
 fg = Finite2DGenerator(self.data)  
 moving = fg.calculate\_moving()  
 self.ui.resultText.setText(**f"0 -** {min(moving)}**"**)  
 fg.draw\_displaced(self.figure.add\_subplot(122)  
 if len(self.figure.get\_axes()) < 2  
 else self.figure.get\_axes()[1])  
 self.figure.canvas.draw()  
 else:  
 self.show\_popup(**"Ошибка"**, **"Не введены необходимые данные"**)  
  
 def bracket\_parameters\_setup(self):  
 try:  
 y = float(self.ui.youngaParameter.text())  
 m = float(self.ui.poissonParameter.text())  
 h = float(self.ui.thicknessParameter.text())  
 self.ui.bracketParametersText.setText(**f"""Параметры кронштейна:  
Коэффициент Юнга:** {y}  
**Коэффициент Пуассона:** {m}  
**Толщина кронштейна:** {h} **мм"""**)  
 self.data.poisson, self.data.young, self.data.thickness = m, y, float(h) / 1000.0  
 self.draw\_plot()  
 self.ui.pageMesh.setEnabled(True)  
 except ValueError:  
 self.show\_popup(**"Ошибка ввода"**, **"Неккоректное число"**)  
  
 def mesh\_parameters\_setup(self):  
 try:  
 size = float(self.ui.meshSizeInput.text())  
 self.data.define\_nodes\_mesh(default\_bracket.copy(), size)  
 self.draw\_mesh()  
 self.setup\_node\_fix\_load\_info()  
 self.set\_checkable\_boxes()  
 val = float(self.ui.powerValue.text())  
 self.calculate(val)  
 except ValueError as e:  
 self.show\_popup(**"Ошибка ввода"**, **"Неккоректное число"**, QMessageBox.Critical, None, **f"**{e}\n{e}**"**)  
  
 def load\_fix\_parameters\_setup(self):  
 fix = self.ui.fixParametersBox.findChildren(QtWidgets.QCheckBox)  
 load = self.ui.loadParametersBox.findChildren(QtWidgets.QCheckBox)  
 fix\_indexes = []  
 load\_indexes = []  
 i = 0  
 for f, l in zip(fix, load):  
 if f.isEnabled() and f.isChecked():  
 fix\_indexes.append(self.define\_node\_edge(f.text(), i))  
 if l.isEnabled() and l.isChecked():  
 load\_indexes.append(self.define\_node\_edge(l.text(), i))  
 i += 1  
 if self.check\_fix\_load\_validate(fix\_indexes, load\_indexes):  
 print(fix\_indexes, load\_indexes)  
  
 @staticmethod  
 def define\_node\_edge(text, default\_value):  
 for i in default\_bracket:  
 if text == **f"**{i}**"**:  
 return i  
 return default\_value  
  
 def check\_fix\_load\_validate(self, fix, load):  
 valid = True  
 if len(fix) == 0 or len(load) == 0:  
 self.show\_popup(**"Неверные данные"**, **"Не заданы закрепления или фиксация"**, QMessageBox.Warning)  
 valid = False  
 elif len(fix) + len(load) > 3:  
 self.show\_popup(**"Неверные данные"**, **"Закрепления и фиксация заданны не корректно"**, QMessageBox.Warning)  
 valid = False  
 elif len([e for e in fix if e in load]) > 0:  
 self.show\_popup(**"Неверные данные"**, **"Закрепления и фиксация совпадают"**, QMessageBox.Warning)  
 valid = False  
 return valid  
  
 def load\_fix\_box\_type\_changed(self):  
 self.set\_checkable\_boxes()  
  
 def set\_checkable\_boxes(self):  
 if self.ui.fixTypeBox.currentIndex() == 0:  
 self.enable\_fix\_node\_checkboxes()  
 else:  
 self.enable\_fix\_node\_checkboxes(False)  
 if self.ui.loadTypeBox.currentIndex() == 0:  
 self.enable\_load\_node\_checkboxes()  
 else:  
 self.enable\_load\_node\_checkboxes(False)  
  
 def setup\_node\_fix\_load\_info(self):  
 fix = self.ui.fixParametersBox.findChildren(QtWidgets.QCheckBox)[0:3]  
 load = self.ui.loadParametersBox.findChildren(QtWidgets.QCheckBox)[0:3]  
 index = 0  
 for f, l in zip(fix, load):  
 f.setText(**f"**{default\_bracket[index]}**"**)  
 l.setText(**f"**{default\_bracket[index]}**"**)  
 index += 1  
 self.ui.bottomFixNodeCheckBox.setChecked(True)  
 self.ui.leftFixNodeCheckBox.setChecked(True)  
 self.ui.rightLoadNodeCheckBox.setChecked(True)  
  
 def enable\_fix\_node\_checkboxes(self, enable=True):  
 self.enable\_checkboxes(self.ui.fixParametersBox, 0, 3, enable)  
 self.enable\_checkboxes(self.ui.fixParametersBox, 3, 6, not enable)  
  
 def enable\_load\_node\_checkboxes(self, enable=True):  
 self.enable\_checkboxes(self.ui.loadParametersBox, 0, 3, enable)  
 self.enable\_checkboxes(self.ui.loadParametersBox, 3, 6, not enable)  
  
 @staticmethod  
 def enable\_checkboxes(widget, i, j, enable):  
 for w in widget.findChildren(QtWidgets.QCheckBox)[i:j]:  
 w.setCheckable(enable)  
 w.setEnabled(enable)  
  
 @staticmethod  
 def show\_popup(title, text, icon=QMessageBox.Critical, info=None, details=None):  
 msg = QMessageBox()  
 msg.setWindowTitle(title)  
 msg.setText(text)  
 msg.setIcon(icon)  
 if info is not None:  
 msg.setInformativeText(info)  
 if details is not None:  
 msg.setDetailedText(details)  
 return msg.exec\_()

**Код Finite2DGenerator:**

from libs.TriangleStiffness import \*  
from libs.StiffnessData import StiffnessData  
import scipy.linalg as sl  
import matplotlib.tri as mtri  
  
  
class Finite2DGenerator:  
 def \_\_init\_\_(self, data: StiffnessData):  
 self.dimension = 2  
 self.data = data  
 self.sorted\_nodes = data.nodes  
 self.stiffness\_elements = self.\_extract\_triangles(self.data.get\_raw\_nodes(), self.data.tri.simplices)  
 self.m\_s = len(self.sorted\_nodes) \* self.dimension  
 self.stiffness\_matrix = None  
 self.calculated\_loads = None  
 self.calculated\_moving = None  
 self.moving\_indexes = None  
 self.\_\_calc\_load = None  
 self.D, self.h = self.data.d(), self.data.thickness  
  
 def get\_stiffness\_matrix(self):  
 col\_size = self.m\_s // len(self.sorted\_nodes)  
 row\_num = 0  
 self.\_\_calc\_load = set()  
 self.stiffness\_matrix = np.zeros((self.m\_s, self.m\_s), dtype=np.float64)  
 for node in self.sorted\_nodes:  
 elements = [e for e in self.stiffness\_elements if node in e.nodes]  
 indexes = [e.nodes.index(node) for e in elements]  
 self.\_append\_to\_matrix(elements, indexes, row\_num, col\_size)  
 row\_num += self.dimension  
 return self.stiffness\_matrix  
  
 def calculate\_moving(self):  
 g\_matrix = self.get\_stiffness\_matrix()  
 indexes = []  
 index = 0  
 for p, m in zip(self.data.loads, self.data.moving):  
 if np.isnan(p[0]) and m[0] == 0:  
 indexes.append(index)  
 elif np.isnan(p[0]) or m[0] == 0:  
 raise ValueError(**"p = "** + str(p) + **" m = "** + str(m))  
 index += 1  
 a = np.delete(np.delete(g\_matrix, indexes, 0), indexes, 1)  
 b = np.delete(self.data.loads, indexes).reshape((-1, 1))  
 self.calculated\_moving = self.data.moving.copy()  
 to\_paste = [i for i in range(len(self.data.moving)) if i not in indexes]  
 x = sl.solve(a, b)  
 for i, v in zip(to\_paste, x):  
 self.calculated\_moving[i] = v  
 self.moving\_indexes = to\_paste  
 return self.calculated\_moving  
  
 def \_append\_to\_matrix(self, elements, indexes, r, col\_size):  
 col = 0  
 col\_nums = [0 for \_ in indexes]  
 rows = [e.get\_stiffness\_matrix(self.D, self.h)[i\*col\_size:i\*col\_size+col\_size] for e, i in zip(elements, indexes)]  
 for node in self.sorted\_nodes:  
 temp = np.zeros((col\_size, col\_size))  
 for i in self.\_get\_node\_in\_element\_indexes(node, elements):  
 temp += rows[i][:, col\_nums[i]:col\_nums[i] + col\_size]  
 col\_nums[i] += col\_size  
 self.stiffness\_matrix[r: r + self.dimension, col: col + col\_size] += temp  
 col += col\_size  
  
 @staticmethod  
 def \_get\_node\_in\_element\_indexes(node, elements):  
 indexes = []  
 i = 0  
 for tr in elements:  
 if node in tr.nodes:  
 indexes.append(i)  
 i += 1  
 return indexes  
  
 @staticmethod  
 def \_extract\_triangles(nodes, simplices):  
 triangles = []  
 for coords in simplices:  
 triangles.append(TriangleStiffness(nodes[coords[0]], nodes[coords[1]], nodes[coords[2]]))  
 return triangles  
  
 def draw(self, axis):  
 axis.triplot(self.data.get\_tri\_x(), self.data.get\_tri\_y(), self.data.tri.simplices)  
 axis.plot(self.data.get\_tri\_x(), self.data.get\_tri\_y(), **'o'**)  
  
 def draw\_displaced(self, axis):  
 cn = self.data.get\_raw\_nodes()  
 sc = []  
 for n in cn:  
 i = self.data.nodes.index(n)  
 sc.append(np.abs(self.calculated\_moving[2 \* i][0]) + np.abs(self.calculated\_moving[2 \* i + 1][0]))  
 triangulation = mtri.Triangulation(self.data.get\_tri\_x(), self.data.get\_tri\_y(), self.data.tri.simplices)  
 axis.triplot(triangulation, **'-k'**)  
 axis.tricontourf(triangulation, sc)

**Код Node:**

from shapely.geometry import Point  
from math import ceil  
  
  
class Node(Point):  
 def \_\_init\_\_(self, \*args):  
 super().\_\_init\_\_(\*args)  
 self.moving = True  
 self.order = None  
 self.num = None  
  
 def set\_fixed(self):  
 self.moving = False  
  
 def is\_moved(self):  
 return self.moving  
  
 def \_\_str\_\_(self):  
 return super().\_\_str\_\_()  
  
 def \_\_repr\_\_(self):  
 return super().\_\_repr\_\_()  
  
 def \_\_hash\_\_(self):  
 return ceil(31 \* self.x + self.y)

**Код TriangleStiffness:**

from abc import ABC  
  
from libs.Node import \*  
from shapely.geometry import Polygon  
import numpy as np  
import math  
import operator  
from functools import reduce  
  
  
origin = []  
refvec = [0, 1]  
  
  
def set\_clockwise\_origin(value\_list):  
 global origin  
 origin = value\_list  
  
  
def clockwise\_angle\_and\_distance(point):  
 vector = [point.x - origin[0], point.y - origin[1]] *# Vector between point and the origin: v = p - o* lenvector = math.hypot(vector[0], vector[1]) *# Length of vector: ||v||* if lenvector == 0: *# If length is zero there is no angle* return -math.pi, 0  
 normalized = [vector[0] / lenvector, vector[1] / lenvector] *# Normalize vector: v/||v||* dotprod = normalized[0] \* refvec[0] + normalized[1] \* refvec[1] *# x1\*x2 + y1\*y2* diffprod = refvec[1] \* normalized[0] - refvec[0]\*normalized[1] *# x1\*y2 - y1\*x2* angle = math.atan2(diffprod, dotprod)  
 *# Negative angles represent counter-clockwise angles so we need to subtract them  
 # from 2\*pi (360 degrees)* if angle < 0:  
 return 2 \* math.pi + angle, lenvector  
 *# I return first the angle because that's the primary sorting criterium  
 # but if two vectors have the same angle then the shorter distance should come first.* return angle, lenvector  
  
  
class TriangleStiffness(Polygon, ABC):  
 def \_\_init\_\_(self, a: Node, b: Node, c: Node):  
 super().\_\_init\_\_([a, b, c])  
 self.a = a  
 self.b = b  
 self.c = c  
 self.\_\_define\_nodes\_order([a, b, c])  
 self.\_stiffness = None  
  
 def draw(self, axis):  
 xy = np.array([self.a.coords[0], self.b.coords[0], self.c.coords[0], self.a.coords[0]])  
 axis.plot(xy[:, 0], xy[:, 1])  
  
 def get\_stiffness\_matrix(self, d, h):  
 if self.\_stiffness is None:  
 i, j, k = 0, 1, 2  
 b = np.array([  
 [self.\_nodes[j].y - self.\_nodes[k].y, 0, self.\_nodes[k].y - self.\_nodes[i].y, 0,  
 self.\_nodes[i].y - self.\_nodes[j].y, 0],  
 [0, self.\_nodes[k].x - self.\_nodes[j].x, 0, self.\_nodes[i].x - self.\_nodes[k].x, 0,  
 self.\_nodes[j].x - self.\_nodes[i].x],  
 [self.\_nodes[k].x - self.\_nodes[j].x, self.\_nodes[j].y - self.\_nodes[k].y,  
 self.\_nodes[i].x - self.\_nodes[k].x, self.\_nodes[k].y - self.\_nodes[i].y,  
 self.\_nodes[j].x - self.\_nodes[i].x, self.\_nodes[i].y - self.\_nodes[j].y],  
 ], dtype=np.float64) / (2 \* self.area)  
 self.\_stiffness = self.area \* h \* b.transpose() @ d @ b  
 return self.\_stiffness  
  
 def \_\_define\_nodes\_order(self, nodes):  
 global origin  
 start\_node = min(nodes, key=lambda n: n.num)  
 coords = [a.coords[0] for a in nodes] *#* center = tuple(map(operator.truediv, reduce(lambda x, y: map(operator.add, x, y), coords), [len(coords)] \* 2))  
 self.\_nodes = [Node(p) for p in sorted(coords, key=lambda coord: (-135 - math.degrees(  
 math.atan2(\*tuple(map(operator.sub, coord, center))[::-1]))) % 360, reverse=True)]  
 temp = self.\_nodes[:self.\_nodes.index(start\_node)]  
 for i in temp:  
 self.\_nodes.remove(i)  
 self.\_nodes.append(i)  
 *# nodes.remove(start\_node)  
 # origin = [start\_node.x, start\_node.y]  
 # self.\_nodes = sorted(nodes, key=clockwiseangle\_and\_distance)  
 # self.\_nodes.append(start\_node)  
 # self.\_nodes.reverse()* @property  
 def nodes(self):  
 return self.\_nodes

**ПРИЛОЖЕНИЕ Б**

(обязательное)

**Чертеж кронштейна**