Описание генератора функций

Содержание

Введение. Возможности, файлы, идея

Сейчас (31.08.15) генератор умеет генерировать функции в таких случаях:

- одномерный случай с соединениями, ни в одном уравнении нет производных порядка выше второго;
- двумерный случай с соединениями, ни в одном уравнении не содержится смешанных производных и производных порядка выше второго;
- двумерный случай, в уравнениях могут содержаться смешанные производные второго порядка, ни в одном уравнении нет производных порядка выше второго — правильно может быть обработана только ситуация, когда нет соединяющихся блоков (т.е. при генерировании функций не приходится приближать смешанные производные в местах соединений).
- трехмерный случай с соединениями, ни в одном уравнении не содержится смешанных производных и производных порядка выше второго.

Для генерирования функций используются файлы, находящиеся в папке domainmodel:

1. customOfficer.py;
2. equationParser.py;
3. funcGenerator.py;
4. abstractGenerator.py;
5. generator1D.py, generator2D.py, generator3D.py;
6. rhsCodeGenerator.py;
7. derivCodeGenerator.py;
8. someFuncs.py;
9. DerivHandler.py.

Файлы 8 и 9 используются для генерации, но функции, лежащие в них, могут использоваться и для других целей. Из файла 8 для генерации используются функции **NewtonBinomCoefficient()**, **generateCodeForMathFunction()**, **determineNameOfBoundary()**, **RectSquare()**, **determineCellIndexOfStartOfConnection2D()**, **getRanges()**. Функции **factorial()** и **getCellCountAlongLine()** из этого же файла используются функциями **NewtonBinomCoefficient()**, **getRanges()**. Файл 9 содержит функцию, которая находит порядок старшей производной.

Основная идея.

Количество и вид рассчетных функций, которые надо сгенерировать, зависит от области, на которой задана система. Центральных функций для блока столько, сколько уравнений задано внутри блока. Сначала генерируются они. Потом генерируются граничные функции.

Сначала все границы каждого блока разбиваются на регионы, каждый из которых обладает уникальными по сравнению с другими регионами свойствами. Свойствами региона являются:

- система уравнений (т.к. в блоке может быть задано несколько систем),
- краевое условие или наличие соединения с другим блоком,
- границы

В зависимости от этих свойств регион может быть либо куском соединения с другим блоком, либо куском границы, на который наложено краевое условие. В первом случае регион будет экземпляром класса Connection, во втором случае – класса BoundCondition. Каждый из этих классов знает систему уравнений, которая задана пользователем на этом регионе, и границы этого региона. Также есть специфические поля. Для каждой границы блока составляется список таких регионов. Из

этих списков составляется список, характерный для всего блока. Получилось, что каждому региону соответствует рассчетная функция, которую предстоит сгенерировать. И никаким двум регионам не соответствуют одинаковые рассчетные функции. Теперь выполняется проход по всем регионам в составленном списке, и для каждого региона генерируется соответствующая функция.

Затем генерирование сначала центральных, потом граничных функций повторяется для следующего блока.

В одномерном, двумерном и трехмерном случаях различаются способы обхода границ блоков, поэтому созданы отдельные классы для обработки 1D, 2D, 3D. А на рисунке 1 нарисован пример сложной области в 2D.

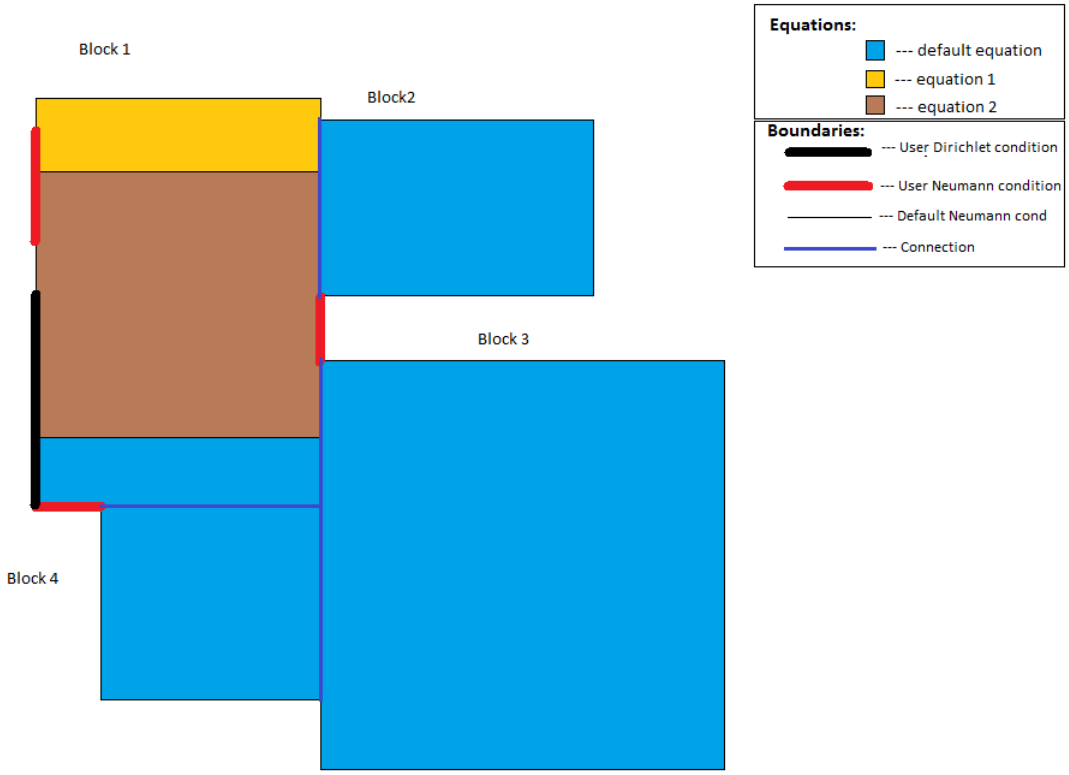

Рис. 1: Пример сложной области в 2D

Для первого блока области, нарисованной на рисунке 1, нужно сгенерировать такие функции: центральную для каждого уравнения (т.е. 3 штуки); на левую границу: два дефолтных Неймана (для 1 и 2 уравнений), два пользовательских Неймана (для 1 и 2 уравнений), 2 условия Дирихле (для 2 и дефолтного уравнений); на верхнюю границу: один дефолтный Нейман; на правую границу: один дефолтный Нейман, 2 функции для соединение со вторым блоком (для 1 и 2 уравнений), один пользовательский Нейман, 2 функции соединения с третьим блоком (для 2 и дефолтного уравнений); на нижнюю границу: один пользовательский Нейман, одно соединение с четвертым блоком. В соответствии с такими требованиями и происходит разбиение границ первого блока на регионы с уникальными свойствами. В 1D все намного проще, а в 3D почти как в 2D.

1D

Тут все просто, потому что каждая из границ каждого блока состоит из единственной клетки. Для каждой границы каждого блока сначала проверяется, есть ли в массиве интерконнектов подходящий интерконнект. Если есть, то сразу генерируется нужная рассчетная функция, иначе генерируется функция с краевым условием (дефолтным или не дефолтным).

2D

Тут обход границ происходит в сторону возрастания координаты по оси, параллельной границе. Все регионы располагаются в массиве в таком же порядке. Для создания рассчетных функций на вершины блока выбирается крайний регион с каждой стороны каждого массива регионов и создаются 4 пары вида (условие на стороне 0, условие на стороне 2), (условие на стороне 2, условие на стороне

1), (условие на стороне 1, условие на стороне 3), (условие на стороне 3, условие на стороне 0). Т.к. они знают информацию о своих соединениях или краевых условиях, то по ним генерируются функции на вершины.

3D

Тут сложность в том, что граница блока – это прямоугольник, поэтому обходить его в каком-то направлении нельзя. В этом случае для каждой границы сначала обходится список систем уравнений (определяется, какие уравнения заданы на этой границе) и вся граница разбивается на меньшие прямоугольники с уникальными системами. Затем для каждого из таких прямоугольников определяется, заданы ли на них (или на отдельных их частях) краевые условия или есть соединения с другими блоками.

1 Описание основных файлов

1.1 Файл customOfficer.py и проверка входных данных

Файл customOfficer.py создан для проверки некоторых введенных в *.json данных на корректность перед запуском генерации функций. Этот файл содержит класс **Reviewer**, который связан с внешним миром методом **ReviewInput()**. В методе **createCPPandGetFunctionMaps()** класса **Model** создается экземпляр класса **Reviewer** и вызывается его метод **ReviewInput()**. Этот метод проверит правильность введения блоков и параметров.

Основные методы класса **Reviewer**:

- **ReviewParameters()**. Проверит, нет ли в списке параметров "Params"повторяющихся имен; совпадает ли множество ключей каждого из словарей в "ParamValues"с множеством элементов списка "Params"; правильно ли установлен "DefaultParamsIndex".
- **ReviewBlocks()**. Проверит каждый из блоков по таким критериям:
  - размеры блока ("Size") заданы неотрицательными числами;
  - индекс уравнения по умолчанию и индекс начального условия по умолчанию заданы корректно;
  - корректно заданы границы регионов уравнений, регионов начальных условий, регионов граничных условий (при этом нет контроля за наложением одного региона на другой, это остается на совести пользователя); этим будет заниматься метод **ReviewEqRegOrInitRegOrBoundReg()**;
  - одинаково количество уравнений в каждой системе, участвующей в задаче (система участвует в задаче, если она задана в списке "Equations"и при этом есть блок или часть блока, на котором задана именно эта система) (метод **ReviewEquations()**);
  - количество компонент для каждого начального условия совпадает с количеством уравнений в каждой системе (метод **ReviewInitials()**). То же самое и для граничных условий (метод **ReviewBounds()**).
- **ReviewInput()**. Вызывает методы **ReviewParameters()** и **ReviewBlocks()**.

Недостатки проверяльщика:

- Не определено до конца, какие данные нужно проверять на корректность, а какие нет. Поэтому некоторые проверки могут быть лишними, некоторые — отсутствовать.
- Корректность ввода интерконнектов пока что нигде не проверяется!

1.2 Файл equationParser.py и формат ввода уравнений, начальных и граничных условий

Формат и правила ввода уравнений и математических функций:

- В левой части уравнения обязательно должно стоять имя компоненты искомой функции, символ ', обозначающий производную по t, затем после любого количества пробелов – знак равно, затем после любого количества пробелов – математическая функция;
- Между отдельными символами внутри математической функции может стоять любое количество пробелов, но названия элементарных функций обязательно должны быть записаны без пробелов (например, нельзая написать '$c\quad os(x)$', но можно написать '$cos(\quad x)$' или '$cos\quad (x)$'!);
- Среди элементарных функций поддерживаются следующие: 'exp', 'sin', 'cos', 'tan', 'sinh', 'tanh', 'sqrt', 'log' ('log' — натуральный логарифм);
- Для возведения какого-нибудь выражения F в **натуральную** степень n надо написать

  `'(F)^n'`

- Аргумент любой функции надо целиком заключать в скобки, даже если он состоит из одного символа; например, $cos\left((x + 25 * y)^{12}\right)$;
- Имена параметров и компонент искомой функции могут быть любыми, а имена независимых переменных всегда такие: t, x, y, z;
- Естественно, при вводе начальных условий нельзя, чтобы функция, играющая роль начального условия, зависела от t; Также (при задании граничных условий) надо внимательно смотреть, на какую границу наложено граничное условие, и соответствующая пространственная переменная должна отсутствовать.

Мозги парсера — библиотека pyparsing. Этот файл (equationParser.py) содержит три класса:

- **CorrectnessController**. Занимается проверкой уравнений и математических функций (например, используемых в качестве начальных или граничных условий) на корректность ввода (например, правильность расстановки скобок, правильность расстановки операторов $+$, $-$, $*$, $/$ и т.д.)
- **ParsePatternCreater**. Чтобы распарсить некое выражение средствами pyparsing, нужно составить подходящую под это выражение грамматику. Нам надо парсить уравнения (выделять их правые части) и отдельно функции (для граничных или начальных условий). Для этого надо создать 2 разных (но очень похожих) грамматики. Еще надо получать из левых частей уравнений список компонент искомой функции. Надо поэтому уравнения парсить уже по-другому (выделять их левые части), т.е. есть необходимость создания третьей грамматики. Этот класс отвечает за создание этих грамматик.
- **MathExpressionParser**. С помощью созданной предыдущим классом грамматики либо парсит уравнение, либо математическую функцию, либо возвращает список компонент искомой функции.

Клиент использует только последний из этих классов. Это происходит в методе **generateAllPointInitials()** класса **abstracGenerator** для того, чтобы парсить начальные условия (после этого сразу генерируются рассчетные функции для начальных условий); в методе **generateCentralFunctionCode()** того же класса для парсинга уравнений, необходимых для генерации центральных функций (после этого генерируются все рассчетные центральные функции); в реализациях матода **generateBoundsAndIcs()** классов **generator1D**, **generator2D**, **generator3D** (после этого генерируются всевозможные граничные условия и соединения).

Недостатки парсера:

- Работает долго.

- Проверка на корректность ввода уравнений и математических функций (расстановка скобок и т.д.) сделана без использования средств pyparsing. Это порождает целый дополнительный класс, который занимается проверкой.

Философия.

Вообще использование pyparsing может быть лишним. Потому что с его помощью сначала строка преобразуется в список строк (в основном все строки там односимвольные, полноценные строки – это только производные, например, "$D[U, \{x, 1\}, \{y, 1\}]$" и степени), а потом только отдельные элементы этого списка (производные; выражения, возведенные в степень; компоненты искомой функции; имена параметров) заменяются на выражения C++. Использование pyparsing удобно тем, что позволяет парсить математические функции любой вложенности, например $[cos(sqrt(x + y) - sin(300 * cos(y)) * y)]$[59], довольно легко (например, pyparsing сам умеет работать с любым количеством пробелов, поэтому во время парсинга об этом думать не приходится). Также pyparsing позволяет удобно парсить выражения, где имена переменных или параметров состоят больше чем из одного символа.

1.3 Файлы funcGenerator.py, abstractGenerator.py, generator1D.py, generator2D.py, generator3D.py

Класс **AbstractGenerator**, находящийся в файле abstractGenerator.py, и его наследники **Generator1D**, **Generator2D** и **Generator3D**, находящиеся в соответствующих файлах, выполняют почти всю работу, связанную с генерацией. В классе **AbstractGenerator** реализованы общие для всех трех случаев методы генерирования:

- сигнатуры вообще любой рассчетной функции;
- всех дефайнов (располагающихся вверху сишного файла);
- функций, инициализимрующих параметры;
- рассчетных функций для начальных условий и функций-заполнителей массивов, содержащих указатели на эти функции;
- центральных функций;
- граничных функций с условием Дирихле (метод **generateDirichlet()** – он не использует других классов и сам генерирует код для одного уравнения) и граничных функций с условием Неймана или с соединением (метод **generateNeumannOrInterconnect()** – использует методы класса **RhsCodeGenerator**);

и еще некоторые общие методы.

Наследники этого класса призваны разделить рассматриваемую область задачи на «регионы уникальности», тем самым определив количество и границы всевозможных рассчетных центральных функций и рассчетных граничных функций. Составляется массив регионов для центральных функций и массив регионов для граничных функций. Каждый из этих классов делает это по-своему. Затем для каждого региона генерирутеся соответствующая функция. Это делается с помощью классов **RhsCodeGenerator**, **PureDerivGenerator** и **MixDerivGenerator** из файлов rhsCodeGenerator.py и derivCodeGenerator.py

Управляет всей этой работой класс **FuncGenerator**, описанный в файле funcGenerator.py. В методе **createCPPandGetFunctionMaps()** класса **Model** создается экземпляр этого класса. В конструкторе этого класса в зависимости от размерности задачи выбирается конкретная реализация генератора (т.е. полем этого класса становится экземпляр одного из трех классов **Generator1D**, **Generator2D**, **Generator3D**). В дальнейшем метод **generateAllFunctions()** класса **FuncGenerator** работает именно с этой конкретной реализацией. Этот метод, оперируя генератором для нужной размерности, использует методы соответствующего генератора для генерирования:

- определений всех констант (дефайнов);
- функций для работы с начальными условиями и параметрами;
- для каждого блока — набора центральных функций, набора граничных, угловых и соединительных (интерконнектов) функций.

Также этот метод формирует список словарей **functionMaps**, по которому формируется матрица пересчета и структура которго описана в файле definition в папке doc.

В файле abstractGenerator.py определены такие классы:

- **AbstractGenerator** (еще раз про него). Содержит общие для всех трех генераторов поля и методы. Например, в нем определены методы генерирования констант (дефайнов) (метод **generateAllDefinitions()**), функций для начальных условий (**generateAllPointInitials()**), параметров (**generateParamFunction()**); методы генерирования функций для граничных условий Дирихле (**generateDirichlet()**) или Неймана, а также интерконнектов (**generateNeumannOrInterconnect()**) (функция, генерирующая интерконнект, совпадает с функцией, генерирующей Неймана, просто ей передаются параметры с другим смыслом); метод генерирования центральных функций **generateCentralFunctionCode()**.

- **InterconnectRegion** и **InterconnectRegion3D**. Созданы для того, чтобы унифицировать обработку границ двумерного и трехмерного блока, т.к. граничные условия и соединения с другими блоками для массива **functionMaps** надо представлять в одном и том же формате. Просто при создании экземпляра генератора для каждого блока создается список его соединительных регионов. Эти классы отличаются тем, что в **InterconnectRegion3D** отсутствуют некоторые поля, которые есть в классе **InterconnectRegion**, т.к. способ обхода блока в 3D другой и эти поля не нужны.
- **BoundCondition** и **Connection**. Метод **getBlockInfo()** в двумерном и трехмерном случаях составляет список элементов, которые есть экземпляры этих классов.

1.3.1 Об изменениях

Изменениям подверглись файлы abstractGenerator.py и generator2D.py.

Из файла abstractGenerator.py в файл boundaryTypes.py вынесено описание классов **BoundCondition** и **Connection**. Также в этот новый файл добавлен класс **BoundaryCondition**, но это сделано зря, и его вообще надо оттуда удалить. Просто он был создан в момент, когда появилась идея объединить классы **BoundCondition** и **Connection** в один. Но потом эта идея оказалась плохой, так что этот класс просто лишний.

В классе **Generator2D** начали делаться такие изменения, которые бы позволили генерировать код для смешанных производных с учетом соединений блоков. Сложность в том, что при генерировании смешанной производной с учетом соединений возникает необходимость генерировать (кроме обычных) несколько рассчетных функций, каждая из которых должна пересчитывать лишь одну клетку блока. И при разном расположении соединений блоков нужно учесть много разных вариантов для генерации таких функций.

Т.о. теперь при разбиении границ блока на «регионы уникальности» среди этих регионов будет много одноклеточных регионов. Но проблема в том, что (т.к. эти новые регионы появляются только в случае наличия смешанной производной хотя бы в одном из уравнений системы) каждый из таких регионов должен знать информацию о соседних с ним регионах (например, граничное условие или соединение, заданное на соседнем регионе).

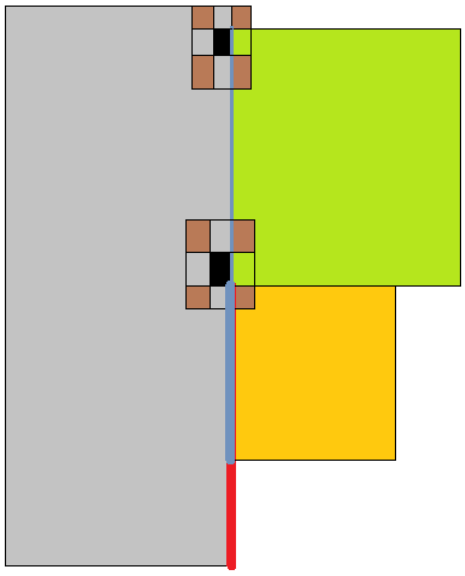

Рис. 2: Для пересчета черных клеток нужна информация из соседних регионов

Идея для решения этого вопроса такая: также как и раньше делать первый проход по границам блока и разбивать их на «регионы уникальности», а потом делать второй проход (уже по этим регионам) и (в зависимости от типа соседних регионов) из одного текущего региона создавать 1, 2 или 3 новых региона, 1 или 2 из которых являются одноклеточными и расположены по краям старого текущего региона. После этого опять получается обычный массив регионов, каждому из которых соответствует уникальная рассчетная функция, которую предстоит сгенерировать. Вот только теперь у каждого региона будет больше свойств, чем раньше (например, некоторые регионы будут знать не только свое условие Неймана, но и условие Неймана соседнего региона). Поэтому класс **RhsCodeGenerator** потребуется научить распознавать новые варианты генерации производных в методе **callDerivGenerator()**. Также класс **MixDerivGenerator** придется научить генерировать конечно-разностные аппроксимации для всевозможных вариантов смешанных производных.

Было сделано только следующее. В файле generator2D создан класс **SideInfoCollector2D**, который и выполняет двойной обход границ блока. Методы, отвечающие за первый обход, сделаны и работают. А вот из методов, отвечающих за второй обход, сделан только **extendBoundCondition()**, который создает необходимые одноклеточные регионы для текущего региона в том случае, если на текущем задано условие Неймана. Изменения в классах **RhsCodeGenerator** и **MixDerivGenerator** не были сделаны.

1.4 Файл rhsCodeGenerator.py и генерирование правых частей уравнений, заданных пользователем

Файл содержит класс **RHSCodeGenerator**, который отвечает за генерацию **правой части одного уарвнения** системы, заданной пользователем, с учетом краевых условий или соединений. Пример возврата такой функции — строка:

```
result[idx + 0] = params[0] * ((2.0 * DXM2 * (
        source[idx - Block0StrideX * Block0CELLSIZE + 0] -
        source[idx + 0] + (0.0) * DX)));
```

Этим занимается метод **generateRightHandSideCode()**. Метод **generateNeumannOrInterconnect()** класса **AbstractGenerator** использует именно этот метод. Метод **generateRightHandSideCode()** получает от вызывателя распарсенное уравнение. Проходит по этому уравнению и заменяет некоторые символы на сишный код, например, при обнаружении производной вызывается метод **callDerivGenerator()**, определяющий способ генерирования производной.

Задачу подготовки к генерированию производной с нужным конечно-разностным приближением решают методы **callDerivGenerator()** и **callSpecialDerivGenerator()**. А саму производную генерируют уже методы одного из двух классов, описанных в следующем пункте.

Метод **callDerivGenerator()** определяет ситуацию, для которой генерируется производная: для вершины двумерного или трехмерного блока, или для центральной функции и т.д. Он принимает массив граничных условий, необходимых для генерирования рассчетной функции нужного типа. В этом массиве может быть от одного до трех элементов, являющихся экземплярами классов **BoundCondition** и **Connection**. Вот как связано количество переданных элементов в этом массиве с методом построения рассчетной функции:

- Если массив пуст, значит надо генерировать строку для центральной функции;
- Если в массиве содержится 1 элемент, то надо сгенерировать строку для условия Неймана на границе или для соединения с другим блоком;
- Если в массиве 2 элемента, то надо сгенерировать строку рассчетной функции для вершины блока в 2D или для ребра блока в 3D;
- Если 3 элемента, то надо сгенерировать строку рассчетной функции для вершины блока в 3D;

Затем этот метод вызывает метод **callSpecialDerivGenerator()**, который определяет, нужно сгенерировать чистую производную или же смешанную, и вызывает соответствующую генерацию с помощью классов, описанных в следующем пункте.

Также в этом классе есть метод **generateCodeForPower()**, который генерирует сишный код для выражения, возведенного в натуральную степень.

1.4.1 Об изменениях

В ветке golubenets1 этот класс изменен в лучшую сторону. Эти изменения можно (и желательно) добавить в текущую версию без добавления остальных (правда надо изменить создание экземпляра этого класса и вызов главного метода везде, где это происходит: в методах **generateCentralFunctionCode** и **generateNeumannOrInterconnect()** класса **AbstractGenerator**). В измененном варианте сделано:

- Создан конструктор;
- Метод **generateCodeForPower()** вынесен в файл someFuncs.py, т.к. вызывается еще в других местах;
- Некрасивый стиль написания метода **callDerivGenerator()** заменен на красивый;

1.5 Файл derivCodeGenerator.py и генерирование производных

Т.к. в уравнении могут встретиться и чистые и смешанные производные, а смешанные производные в случае соединения блоков порождают много разных видов функций, которые необходимо сгенерировать, то удобно поручить генерирование чистых и смешанных производных разным классам. Эти классы:

- **PureDerivGenerator**;
- **MixDerivGenerator**.

1.5.1 PureDerivGenerator

Для центральной функции способен сгенерировать чистую производную любого порядка (потому что конечно-разностные приближения для них в этом случае содержат однотипные элементы, различно только их количество и коэффициенты перед слагаемыми). В зависимости от того, какими значениями был проинициализирован экземпляр этого класса, за генерацию производной отвечает одна из трех функций:

- **commonPureDerivativeAlternative()** — генерировать производную для центральной функции или для граничной в случае, когда граничное условие не влияет на производную (например, первая производная по y вдоль границы $x = 0$);
- **specialPureDerivativeAlternative()** — генерировать производную для граничной функции, когда граничное условие влияет на производную (например, производная по x вдоль границы $x = x_{max}$);
- **interconnectPureDerivAlternative()** — генерировать производную для функции-соединения (для интерконнекта).

1.5.2 MixDerivGenerator

Пока что способен генерировать смешанные производные только второго порядка для несоединенных блоков. Функции-генераторы:

- **commonMixedDerivativeAlternative()** — генерировать производную для центральной функции;
- **specialMixedDerivativeAlternative()** — генерировать производную для граничной функции.

Производные в углах двумерного блока генерируются по частям, а потом объединяются в одну строку в методе **callDerivGenerator()** класса **RHSCodeGenerator**.

2 Конечно-разностные формулы, используемые при генерировании производных

Расчетные функции (которые и генерирует генератор) отличаются друг от друга только выражениями, аппроксимирующими производные по пространственным переменным. Если рассматриваемая система уравнений n-мерна (т.е. искомая функция — это вектор-функция $u = (u_1, \ldots, u_n)$, $u_k = u_k(t, x, y, z)$, $k = \overline{1, n}$) и ни одно из уравнений не содержит смешанных производных и производных порядка выше второго, то i-ое уравнение этой системы ($i = \overline{1, n}$) имеет вид:

$$\frac{\partial u_i}{\partial t} = f\left(u_1, \frac{\partial u_1}{\partial x}, \frac{\partial u_1}{\partial y}, \frac{\partial u_1}{\partial z}, \frac{\partial^2 u_1}{\partial x^2}, \frac{\partial^2 u_1}{\partial y^2}, \frac{\partial^2 u_1}{\partial z^2}, \ldots, u_n, \frac{\partial u_n}{\partial x}, \frac{\partial u_n}{\partial y}, \frac{\partial u_n}{\partial z}, \frac{\partial^2 u_n}{\partial x^2}, \frac{\partial^2 u_n}{\partial y^2}, \frac{\partial^2 u_n}{\partial z^2}\right).$$

Для простоты записи можно считать, что система одномерна, т.е. есть всего одно уравнение:

$$\frac{\partial u}{\partial t} = f\left(u, \frac{\partial u}{\partial x}, \frac{\partial u}{\partial y}, \frac{\partial u}{\partial z}, \frac{\partial^2 u}{\partial x^2}, \frac{\partial^2 u}{\partial y^2}, \frac{\partial^2 u}{\partial z^2}\right). \tag{1}$$

Надо аппроксимировать все производные, стоящие в правой части (1). Нужны аппроксимации для:

1. центральных функций;
2. функций для рассчета границ блока;
3. функций для рассчета вершин блока (в двумерном и трехмерном случаях);
4. функций для рассчета ребер блока (в трехмерном случае).

В 2. - 4. надо учитывать краевые условия (в случае краевых условий Неймана как раз и возникает необходимость аппроксимировать производные, а в случае условий Дирихле вместо правой части уравнения (1) просто подставляется значение производной функции, являющейся этим условием), соединения блоков и то, что на разных частях одного блока могут быть заданы разные уравнения. Для приближения производных во всех функциях используются центральные конечные разности.

Рассмотрим трехмерный блок — параллелепипед

$$\Pi = \{(x, y, z) \in R^3 | x \in [x_0, x_1], y \in [y_0, y_1], z \in [z_0, z_1]\}.$$

Пусть $m, n, p \in N$ и на Π введена сетка с шагами $\Delta x = (x_1 - x_0)/m$, $\Delta y = (y_1 - y_0)/n$, $\Delta z = (z_1 - z_0)/p$ по пространственным переменным x, y, z и введен шаг Δt по времени t. Значения функции u в узлах сетки далее обозначаются обычным образом:

$$u^{\ell}_{ijk} := u(\ell\Delta t, x_0 + i\Delta x, y_0 + j\Delta y, z_0 + k\Delta z),$$

а значения производных в узлах сетки пишутся похожим образом, например:

$$\left(\frac{\partial^2 u}{\partial y^2}\right)^{\ell}_{ijk} := \frac{\partial^2 u}{\partial y^2}(\ell\Delta t, x_0 + i\Delta x, y_0 + j\Delta y, z_0 + k\Delta z).$$

Здесь $\ell = \overline{0, L}$, $i = \overline{0, m}$, $j = \overline{0, n}$, $k = \overline{0, p}$.

Замечание: т.к. генератор все-таки умеет генерировать функции для задачи, когда в уравнении есть смешанные производные и при этом рассматриваются блоки без соединений, то для этой ситуации ниже также описаны формулы, используемые генератором для аппроксимации смешанных производных. Для такой задачи уравнение (1) выглядит по-другому:

$$\frac{\partial u}{\partial t} = f\left(u, \frac{\partial u}{\partial x}, \frac{\partial u}{\partial y}, \frac{\partial u}{\partial z}, \frac{\partial^2 u}{\partial x \partial y}, \frac{\partial^2 u}{\partial x \partial z}, \frac{\partial^2 u}{\partial z \partial y}, \frac{\partial^2 u}{\partial x^2}, \frac{\partial^2 u}{\partial y^2}, \frac{\partial^2 u}{\partial z^2}\right). \tag{2}$$

2.1 Формулы для производных в центральных функциях

В центральных функциях для приближения производных используется самый обычный вид конечно-разностных формул.

Первая производная (например, по z):

$$\left(\frac{\partial u}{\partial z}\right)^{\ell}_{ijk} \approx \frac{u^{\ell}_{ijk+1} - u^{\ell}_{ijk-1}}{2\Delta z} \tag{3}$$

Вторая чистая производная (например, по yy):

$$\left(\frac{\partial^2 u}{\partial y^2}\right)^{\ell}_{ijk} \approx \frac{u^{\ell}_{ij+1k} - 2u^{\ell}_{ijk} + u^{\ell}_{ij-1k}}{(\Delta y)^2} \tag{4}$$

Для центральных функций смешанные производные генерировать очень легко. Смешанная производная (например, по xz):

$$\left(\frac{\partial^2 u}{\partial x \partial z}\right)^{\ell}_{ijk} \approx \frac{u^{\ell}_{i+1jk+1} - u^{\ell}_{i-1jk+1} - u^{\ell}_{i+1jk-1} + u^{\ell}_{i-1jk-1}}{4\Delta x \Delta z} \tag{5}$$

2.2 Формулы для производных в функциях рассчета границ блока

2.2.1 Левая граница

Рассмотрим, например, границу $y = y_0$ блока. Пусть на каком-то куске

$$\Pi_0 = \{(x, y, z) \in \Pi | \quad y = y_0; \quad x_0 \leqslant a \leqslant x \leqslant b \leqslant x_1; \quad z_0 \leqslant c \leqslant z \leqslant d \leqslant z_1\}$$

этой границы пользователь задал краевое условие Неймана:

$$\frac{\partial u}{\partial y}(t, x, y_0, z) = \varphi(t, x, z).$$

И пусть на Π_0 также задано уравнение (2). Тогда при аппроксимации производных в функциях для рассчета куска Π_0 используются такие формулы.

Первая производная по y:

$$\left(\frac{\partial u}{\partial y}\right)^{\ell}_{i0k} \approx \varphi^{\ell}_{ik} = \varphi(\ell\Delta t, x_0 + i\Delta x, z_0 + k\Delta z), \tag{6}$$

а первые производные по x и z будут выглядеть как в (3) (т.е. будут центральными).

Вторая производная по y:

$$\left(\frac{\partial^2 u}{\partial y^2}\right)^{\ell}_{i0k} \approx \frac{2\left(u^{\ell}_{i1k} - u^{\ell}_{i0k} - \Delta y \varphi^{\ell}_{ik}\right)}{(\Delta y)^2},$$

а вторые производные по x и z будут как в (4) (т.е. будут центральными).

Смешанная производная по xy:

$$\left(\frac{\partial^2 u}{\partial x \partial y}\right)^{\ell}_{i0k} \approx \frac{\varphi^{\ell}_{i+1k} - \varphi^{\ell}_{i-1k}}{2\Delta x}; \tag{7}$$

смешанная производная по zy:

$$\left(\frac{\partial^2 u}{\partial z \partial y}\right)^{\ell}_{i0k} \approx \frac{\varphi^{\ell}_{ik+1} - \varphi^{\ell}_{ik-1}}{2\Delta z}; \tag{8}$$

смешанная производная по xz будет как в (5) (т.е. будет центральной).

Предположим теперь, что область Π_0 является местом соединения рассматриваемого блока с каким-то другим. Это значит, что на Π_0 вообще не может быть задано краевых условий. Здесь

генератор бессилен при генерировании смешанных производных, поэтому считаем, что на Π_0 задано уравнение (1). Формулы такие.

Первая производная по y:

$$\left(\frac{\partial u}{\partial y}\right)^{\ell}_{i0k} \approx \frac{u^{\ell}_{i1k} - ic[idx1][idx2]}{2\Delta y},$$

а первые производные по x и z опять будут выглядеть как в (3). Тут ic – массив интерконнектов, передающийся в рассчетную функцию, а $idx1$ и $idx2$ определяются генератором.

Вторая производная по y:

$$\left(\frac{\partial^2 u}{\partial y^2}\right)^{\ell}_{i0k} \approx \frac{u^{\ell}_{i1k} - 2u^{\ell}_{i0k} + ic[idx1][idx2]}{(\Delta y)^2},$$

а вторые производные по x и z опять будут как в (4).

Формулы для границ $x = x_0$, $z = z_0$ строятся по аналогии.

2.2.2 Правая граница

Рассмотрим границу $y = y_1$ блока. Пусть на куске

$$\Pi_1 = \{(x, y, z) \in \Pi | \quad y = y_1; \quad x_0 \leqslant A \leqslant x \leqslant B \leqslant x_1; \quad z_0 \leqslant C \leqslant z \leqslant D \leqslant z_1\}$$

этой границы пользователь задал краевое условие Неймана:

$$\frac{\partial u}{\partial y}(t, x, y_1, z) = \psi(t, x, z).$$

И пусть на Π_1 также задано уравнение (2). Формулы такие.

Для первых производных по x, z формулы будут такие же, как в (3), а по y — как в (6) с заменой функции φ на ψ и среднего индекса 0 на n в левой части. Для смешанной производной по xz формула как в (5), а для производных по xy и zy формулы как в (7), (8) с заменой функции φ на ψ и среднего индекса 0 на n в левой части.

Сильно изменяется вторая производная по y:

$$\left(\frac{\partial^2 u}{\partial y^2}\right)^{\ell}_{ink} \approx \frac{2\left(u^{\ell}_{in-1k} - u^{\ell}_{ink} + \Delta y \psi^{\ell}_{ik}\right)}{(\Delta y)^2},$$

а вторые производные по x и z опять будут как в (4).

Если область Π_1 является местом соединения рассматриваемого блока с каким-то другим, то на Π_0 вообще не может быть задано краевых условий. Считаем, что на Π_1 задано уравнение (1). Формулы такие.

Первая производная по y:

$$\left(\frac{\partial u}{\partial y}\right)^{\ell}_{ink} \approx \frac{ic[idx1][idx2] - u^{\ell}_{in-1k}}{2\Delta y},$$

а первые производные по x и z опять будут выглядеть как в (3).

Вторая производная по y:

$$\left(\frac{\partial^2 u}{\partial y^2}\right)^{\ell}_{ink} \approx \frac{ic[idx1][idx2] - 2u^{\ell}_{ink} + u^{\ell}_{in-1k}}{(\Delta y)^2},$$

а вторые производные по x и z опять будут как в (4).

2.3 Формулы для производных в функциях рассчета вершин блока (в $2D$ и $3D$) и в функциях рассчета ребер (в $3D$)

Здесь для производных формулы те же самые, что и в п. 2.1 и 2.2. Отличие только в том, что для гарничных функций производные только по какому-нибудь одному аргументу будут приближены

нецентральными формулами, а по всем остальным — центральными. Здесь же для вершин в двумерном случае производные по обоим аргументам будут приближены нецентральными формулами, а в трехмерном — по всем трем. Для ребер в $3D$ производные по двум аргументам будут приближены нецентральными формулами, а по третьему аргументу — центральными.

Особенность только в аппроксимации смешанных производных. Пусть рассматривается трехмерный блок П (без соединений!) и в окрестности какого-то угла, например, $x = x_0$, $y = y_0$, $z = z_1$ задано уравнение (2) и заданы такие условия Неймана на соответствующие границы (можно сразу для простоты считать, что они заданы на каждой из границ полностью, а не на отдельных их частях):

$$\frac{\partial u}{\partial x}(t, x_0, y, z) = f(t, y, z), \quad \frac{\partial u}{\partial y}(t, x, y_0, z) = g(t, x, z), \quad \frac{\partial u}{\partial z}(t, x, y, z_1) = h(t, x, y).$$

Тогда производные по xy, xz, yz в рассматриваемой вершине будут аппроксимироваться формулами:

$$\left(\frac{\partial^2 u}{\partial x \partial y}\right)^{\ell}_{00p} \approx \left(\frac{f^{\ell}_{j+1k} - f^{\ell}_{j-1k}}{4\Delta y} + \frac{g^{\ell}_{i+1k} - g^{\ell}_{i-1k}}{4\Delta x}\right)\Bigg|_{i=0,j=0,k=p};$$

$$\left(\frac{\partial^2 u}{\partial x \partial z}\right)^{\ell}_{00p} \approx \left(\frac{f^{\ell}_{jk+1} - f^{\ell}_{jk-1}}{4\Delta z} + \frac{h^{\ell}_{i+1j} - h^{\ell}_{i-1j}}{4\Delta x}\right)\Bigg|_{i=0,j=0,k=p};$$

$$\left(\frac{\partial^2 u}{\partial y \partial z}\right)^{\ell}_{00p} \approx \left(\frac{g^{\ell}_{ik+1} - g^{\ell}_{ik-1}}{4\Delta z} + \frac{h^{\ell}_{ij+1} - h^{\ell}_{ij-1}}{4\Delta y}\right)\Bigg|_{i=0,j=0,k=p}.$$

Проблема тут в том, что приходится считать, что нам известны значения функций f, g и h за пределами блока П.

Формулы для производных на ребрах чуть-чуть отличаются. Ребро — это место пересечения двух границ. Рассмотрим какое-нибудь ребро, например, $x = x_1$, $z = z_0$. Пусть на соответствующих границах заданы условия Неймана:

$$\frac{\partial u}{\partial x}(t, x_1, y, z) = \xi(t, y, z), \quad \frac{\partial u}{\partial z}(t, x, y, z_0) = \eta(t, x, y).$$

Тогда формулы для аппроксимации производных на рассматриваемом ребре такие:

$$\left(\frac{\partial^2 u}{\partial x \partial y}\right)^{\ell}_{mj0} \approx \left(\frac{\xi^{\ell}_{j+1k} - \xi^{\ell}_{j-1k}}{2\Delta y}\right)\Bigg|_{k=0};$$

$$\left(\frac{\partial^2 u}{\partial x \partial z}\right)^{\ell}_{mj0} \approx \left(\frac{\xi^{\ell}_{jk+1} - f^{\ell}_{jk-1}}{4\Delta z} + \frac{\eta^{\ell}_{i+1j} - \eta^{\ell}_{i-1j}}{4\Delta x}\right)\Bigg|_{i=m,k=0};$$

$$\left(\frac{\partial^2 u}{\partial y \partial z}\right)^{\ell}_{mj0} \approx \left(\frac{\eta^{\ell}_{ij+1} - \eta^{\ell}_{ij-1}}{4\Delta y}\right)\Bigg|_{i=m}.$$