1 Постановка задачи

1.1 Цели и задачи модификации

- обеспечить по итогам текущей модернизации кода удобство интегрирования новых реологических моделей МСС, новых численных алгоритмов, выполняющих расчёт этих моделей;
- сделать код более структурированным, более понятным и простым для ознакомления и будующих расширений.
- не потерять при этом значительно быстродействие

1.2 Постановка задачи

На данный момент задача в самой общей постановке представляет собой систему N квазилинейных гиперболических уравнений в частных производных:

$$\frac{\partial \vec{u}}{\partial t} + \mathbf{A}_x \frac{\partial \vec{u}}{\partial x} + \mathbf{A}_y \frac{\partial \vec{u}}{\partial y} + \mathbf{A}_z \frac{\partial \vec{u}}{\partial z} = \vec{f}, \tag{1.1}$$

где $\mathbf{A}_x = \mathbf{A}_x(\vec{u}, \vec{r}, t)$, $\mathbf{A}_y = \mathbf{A}_y(\vec{u}, \vec{r}, t)$, $\mathbf{A}_z = \mathbf{A}_z(\vec{u}, \vec{r}, t)$, $\vec{f} = \vec{f}(\vec{u}, \vec{r}, t)$.

Также движение сетки описывается уравнением:

$$\frac{\partial \vec{r}}{\partial t} = \vec{v}. \tag{1.2}$$

Кинетика разрушений в общем случае описывается системой эволюционных уравнений:

$$\frac{\partial \vec{\chi}}{\partial t} = \vec{F}(\vec{r}, t, \vec{u}, \vec{\chi}), \tag{1.3}$$

где $\vec{\chi}$ – так называемые внутренние параметры, характеризующие внутреннюю структуру материала (пористость, размер пор, повреждённость, параметр упрочнения и пр.).

Уравнение (1.2) подразумевает замену дифференциального оператора разностным нужного порядка.

Уравнения (1.3) могут иметь различный вид. Пока у нас простейшие модели – решение этих(этого) уравнения не составит труда. Дальше будем думать. Вместе с этими уравнениями могут быть различные критерии и корректоры.

2 Общий подход

2.1 Принципиальная архитектура

Код модульный. «Большие» сущности (модель, метод, сетка – см.ниже) являются чисто описательными классами, содержащими соответствующие базовые «кирпичики» (корректоры, сеттеры, интерполяторы и т.д.)

From Sasha: то есть ссылки на них?

From @avasyukov: имхо, наружу - методы для получения. Что внутри - зависит от ситуации. Например, можно хранить экземпляр объекта и возвращать референс на него, а можно не хранить и дёргать правильный Singletone.

2.2 Что задаётся в таске

В таске задаются:

- геометрию задачи (тела, их расположение),
- граничные и контактные условия (**Calculator** + область действия).
- начальные условия
- физическая модель для неразрушенного тела (**RheologyModel** по тексту ниже),
- физическая модель разрушения (**FailureModel** по тексту ниже),
- численный метод (**Solver** по тексту ниже),
- тип сетки (**Mesh** по тексту ниже),
- интерполятор (**Interpolator** по тексту ниже, зависит от метода и сетки, фактически определяет порядок по пространству)

> From Sasha: нужен ли он здесь? Может его в численный метод?
>
> From @avasyukov: меня тоже смущает, что он «мельче» всего остального. Но метод (Solver) – большая штука, а интерполяторов много разных. И если ради прогона «всего того же самого, но с другим лимитером» придётся наплодить копи-пастой кучу Solver-ов – будет нехорошо. М.б. Solver должен нести с собой дефолтный интерполятор, а в таске можно задать кастомный. Обсуждаемо.

3 Модель

3.1 RheologyModel

Для пущей структурированности и удобства имплементации различной физики предлагаю ввести класс **RheologyModel**.

Переменная **RheologyModelName**.

В этот класс предлагаю ввести следующие сущности:

- **Material** material – материал,
- Информация о требуемом типе **Node**'ов. Далее при создании сетки должны использоваться **Node**'ы этого типа. Вероятно, потребует шаблонизации класса сетки. Общая схема, вероятно, по аналогии текущим видом MeshLoader'ов – модель создаёт сетку с нодами нужного типа, после чего Loader грузит геометрию,

- **MatrixSetter** matrixSetter – заполняет матрицы текущей модели,
- **InHomogeneousSetter** inHomogeneousSetter – заполняет правую часть уравнений,
- **Correctors** correctors – набор корректоров, если модель их подразумевает (пример - идеальная пластика на базе корректора.

Внутри всего указанного неявно скрывается размерность вектора $\vec{u}$.
Варианты моделей:

- Elasticity,
- ElasticityFiniteStrains,
- NonLinearElasticity,
- Plasticity,
- ThermoElasticity.

3.2 FailureModel

FailureModel с хорошей точностью совпадает с текущей реализаций. Содержит набор критериев и корректоров. Критерии и корректоры логически включают в себя как уравнения, так и алгоритм их решения (не обращаются в общем случае к дополнительным внешним классам).

Переменная **FailureModelName**. Переменная **FailureModelType** – = **discrete** *or* = **continuum**.

Осозн до конца ли континуальная модель ложится на такую структуру.

Примерные варианты реализаций(дочерние классы):

- FailureDiscrete,
  - MisesCriterion,
  - MohrCoulombCriterion,
  - HashinCriterion,
  - TsaiHillCriterion,
  - TsaiWuCriterion;
- FailureContinuum,
  - Failure1Order,
  - Failure2Order;

4 Метод

4.1 GCMsolver

Родительский класс – **GCMsolver**. Его предлагаю сделать обобщённым.

Содержит переменные **spaceOrder**, **timeOrder**, **bufNodes**. А также **bufCoefficients** – наклоны характеристик, которые уточняются в процессе решения.

> From @avasyukov: оно правда надо? Особенно про наклоны.
> From Sasha: поместить в CrossPointFinder.

Далее, флаг **HomogeneousSystem**. Необходимы отдельно метод для однородных систем и отдельно для неоднородных систем.

Далее, поскольку у нас сеточно-характеристический метод и больше никого (появится FEM будет FEMsolver) разумно определить сюда базовые кирпичи GCM'а.

- **CrossPointFinder** crossPointFinder - ищет точку пересечения характеристики,
- **ModelPtr** – ссылка на текущую модель,
  - **MatrixSetter** matrixSetter – заполняет матрицы,
  - **InHomogeneousSetter** inHomogeneousSetter – заполняет правую часть уравнений,

    > From Sasha: эти две вещи в модель.
    > From @avasyukov: да. Имеется в виду, что есть ссылка на модель, из которой их можно получить.

- **Decomposer** decomposer – разлагает матрицы (солвер несёт с собой «декомпозер по умолчанию», модель может (как именно?) сказать, что нужен другой декомпозер (пример – у меня изотропное тело и линейная упругость, разложение матриц известно в явном виде)),
- **InHomogeneousSolver** inhomogeneousSolver – используется для моделей с правой частью, решает уравнение на перенос инвариантов $\vec{\xi}$:

$$\frac{\partial \vec{\xi}}{\partial t} = \mathbf{\Omega} \vec{F}, \tag{4.1}$$

- **Interpolator** interpolator – интерполирует значения инвариантов в точке, зависит от типа **Mesh**'а (не всё ко всему применимо), полностью инкапсулирует логику получения нового значения (классический гибридный метод – это фактически такой умный интерполятор, который использует то 1-ый, то 2-ой порядок), задаётся в таске («а давайте всё то же самое, но теперь вторым порядком»),
- **MeshMover** meshMover – двигает узлы, решая уравнение (1.2) нужным порядком,

- **FailureSolver** failureSolver – отвечает за решение (1.3), скорее всего, является просто методом, физически вызывая **FailureModel**, в которой критерии и корректоры содержат нужную математику целиком.
- **SplittingMethod** splittingMethod – в каком порядке (за один полный шаг по времени) происходит расчёт отдельных блоков шага по времени (пространственный расчёт по трём направлениям, движение сетки, применение корректоров и моделей разрушения)
  - первый порядок (расщепления по времени)- все по порядку,
  - второй порядок - некая симметричная структура

  Это должен быть класс, описывающий эту последовательность действий. Возможно, он сильно связан с моделью.

Если кто знает TVD, ENO, аппроксимационную вязкость или любую вязкость – милости просим.

Основной метод **doNextTimeStep** – выполняет следующий шаг по времени.

4.2 Calculator'ы

Собственно:

- VolumeCalculator,
- BorderCalculator,
- ContactCalculator.

Примерно в том виде, какой есть. Не все применимы ко всем моделям.

4.3 CrossPointFinder

Родительский класс. Производные классы – реализации конкретного порядка и просто разные реализации. В зависимости от реализации получает набор нодов и реологических параметров

Переменная **spaceOrder**.

Флаг **linearCharateristics** – если характеристики – прямые линии, в соответствии с моделью тут следует использовать первый порядок, дающий точное положение. Такакя же ситуация как и с **HomogeneousSystem**. Тут два варианта:

- отдельно метод второго порядка для прямых характеристик, с указанием нужного метода в модели;
- На этапе инициализации **Model** программа понимает, что нужно использовать линейный **Finder** для метода второго порядка.

Предполагаемые реализации:

В смысле, это будет отдельный новый солвер?

Нужно описать, как задаются и применяются граничные и

- CrossPointFinder1Order;
- CrossPointFinder2Order.

4.4 Interpolator

Зависит от типа **Mesh**'а (не все со всеми могут работать).
Неоходимы отдельные реализации классов:

- Interpolator1Order;
- Interpolator2Order;
  - Interpolator2Order;
  - InterpolatorLimiter,
    - InterpolatorLimiterMinMax,
    - InterpolatorLimiterLinear.

4.5 MatrixSetter

Получает **curNode** и заполняет матрицы $\mathbf{A}_x$, $\mathbf{A}_y$, $\mathbf{A}_z$ в соответствии с моделью.

4.6 InHomogeneousSetter

Получает **curNode** и заполняет правую часть неоднородного уравнения $\vec{F}$ в соответствии с моделью.

> From Sasha: последние две вещи надо перенести в Модель, они к Методу никак не относятся.
>
> From @avasyukov: да. Имеется в виду, что есть ссылка на модель, из которой их можно получить.

4.7 MeshMover

Переменная **timeOrder**. Решает уравнение (1.2) в соответствии с порядком:

- MeshMover1Order;
- MeshMover2Order.

Получает **curNode** и двигает его в соответствии с решением.

4.8 Decomposer

Реализации:

- NumericalDecomposer;
- AnalyticalDecomposer,
  - IsotropicDecomposer,
  - GeneralAnalyticalDecomposer.

4.9 InHomogeneousSolver

Переменная **timeOrder**.
Решаем уравнение на инварианты (4.1).
Предполагаемые реализации:

- InHomogeneousSolver1Order;
- InHomogeneousSolver2Order.

5 Сетка

Более-менее соответствует нынешнему виду. Должна учитывать:

- параллельность;
- потенциальную возможность перестройки в ходе счёта.

6 Вопросы

- Имена всех классов – важно на самом деле. Оно должно восприниматься интуитивно и соответствовать традициям англоязычной литературы.
  - DeformationModel -> RheologyModel;
- Ноды – как в итоге храним реологию? Что в нодах, что в материалах? Класс нода один со всеми возможными параметрами (координата, скорость, напряжение, плотность, повреждаемость, температура и т.д) или таки некая иерархия?
  - Будет куча моделей с документацией. В документации написано какие каждой модели нужны константы и пр. В модели есть: свой Node, свой Material, свои Setter'ы. В Node'е содержатся **лишь те величины, которые изменяются** - остальное в Material (by @mediev). Методы работы с нодами делать по возможности шаблонными (особые случаи - делать спецификации).

  - Не понял мысль. Уточняю, о чём я. Сейчас есть дикая куча мест в интерфейсах примерно всех классов, где фигурирует CalcNode. Это делает код довольно просто читаемым и воспринимаемым, и этот факт хочется сохранить. Если типов нодов станет много - я не понимаю, что будет с интерфейсами. Также отдельно я не понимаю, каким именно механизмом модель скажет всем окружающим, какие ноды ей нужны. Ещё я не понимаю, насколько велик будет ад внутри всех объектов, если они должны будут уметь работать с «тем-не-знаю-чем» (нодом с произвольными полями). Если что - это не ад и ужас, но вопросы, для которых есть разные варианты решения, и надо определяться, так как ответы влияют примерно на всю структуру кода. В общем, я пока склоняюсь к тому, что класс таки один (CalcNode), в котором аккуратно сложены все поля на все случаи жизни. Некоторые модели просто не используют часть полей. Пока оверхед небольшой, имхо, так резко проще жить в плане всего остального кода. (by @avasyukov)

- В тексте бодро упоминаются TVD, ENO и компания. Насколько реально их реализовать на этой структуре? Что конкретно придётся сделать?
  - Насколько я понимаю это всё вопросы к интерполяции, думаю всё будет ок – былобы желание.
  - Я вот совсем не уверен в этом. Я не курил детали, но есть смутное ощущение, что куча хороших разностных схем (а) не сводится к переносу вдоль характеристик, (б) жёстко завязана на структурированность сетки, (в) хз что ещё потребует от окружающих классов. Скорее всего, глубоко кастомный Solver позволит сделать примерно всё. Но хочется таки прикинуть, правда ли мы не наплодили ограничений где-нибудь. Я понимаю, что невнятно формулирую вопрос – да. (by @avasyukov)

- Параллельность – кто из классов должен быть в курсе?

- Параллельность – есть ли тут реально место для GPU, что для этого нужно, кто из классов должен быть в курсе?

- Collision Detector – его место, с кем и как он связан?

- Про наклоны – наклоны нужно будет сохранять в методах типа предиктор-корректор, Кукуджанов утверждает что так делается второй порядок. В отдельной реализации GCMSolver'a - надо.

- Про калькуляторы – возможно ли их сделать шаблонными по моделям? Типа: template <class RheologyModel> class BorderCalculator - указана модель (в которой указан тип нода), в таске выбирается тип граничного(контактного) условия. Далее в методе идёт цикл по нодам и применяется тот или иной калькулятор в зависимости от типа нода. Создавать сущность VolumeCalculator в подобной модификации смысла не вижу, смысл есть – объясните (by @mediev).

- Очень сильно сомневаюсь, что шаблонизация прокатит. Просто потому что тут не простые замены уровня float / double или разных классов с одинаковым набором методов, а принципиальное изменение логики калькулятора в зависимости от модели. Про VolumeCalculator – тоже не уверен. Есть неподтверждённое ощущение, что разные хорошие схемы (см.выше) могут не сводиться к переносам вдоль характеристик. Тогда дефолтный SimpleVolumeCalculator нужно будет заменить. Хочется оставить принципиальную возможность. (by @avasyukov)

Todo list

- Осознать, до конца ли континуальная модель ложится на такую структуру. . . . 3
- В смысле, это будет отдельный новый солвер? 5
- Нужно описать, как задаются и применяются граничные и контактные условия. . 5