EMtool

Software package documentation (rus)

Mikhail Osintcev, Ph.D.

Пакет EMtool предназначен для численного моделирования электромагнитных волн в 3D. Программа написана на языке FORTRAN 2007 с использованием принципов объектно-ориентированного программирования. Пакет имеет блочную структуру, что позволяет строить численное решение с различными граничными условиями, добавлять новые граничные условия. Архитектура программы позволяет менять схему расчета, однако, в данный момент имплементирована только стандартная схема Yee второго порядка.

Важной особенностью является возможность проводить расчет отдельной задачи с любым имплементированным граничным условием, вести расчет нескольких задач с разными граничными условиями одновременно, что позволяет применять метода лакун, а также метод лакун со скорректированными вспомогательными токами, имея единую имплементацию граничных условий

В программе интегрирована возможность параллелизации алгоритмов на машинах с разделяемой памятью (shared memory) при помощи стандарта OpenMP.

1. Структура файлов

Файловая структура программы продиктована стандартами ООП, когда имплементация отдельного объекта располагается в отдельном файле. Ниже приведен перечень файлов программы и их базовое назначение. Программа состоит из следующих файлов:

Главный файл и инициализация
main.f90 - главный файл программы, содержащий функцию main.
commonvars.f90 - файл, содержащий глобальные переменные, введение которых продиктовано несовершенством языка FORTRAN и желанием сделать структуру программы проще.
parallel.f90 - файл, содержащий функции для настройки параллелизации программы и оценке эффективности параллелизации.

Модули источников
newdipole.f90 - файл с бездивергентным источником и соответствующим аналитическим решением.
sdipole.f90 - файл с не бездивергентным источником и соответствующим аналитическим решением.
sourceclass.f90 - файл с классом tSource, отвечающим за поставку точного аналитического решения и источников в задачу.

Модули вспомогательных классов
meshclass.f90 - файл с описанием класса tMesh, отвечающего за хранение значений расчетной области.
timersclass.f90 - файл с описанием класса tTimer, отвечающего за учет времени работы вспомогательных задач в методе лакун.

auxmeshclass.f90 - файл с описанием класса tAuxMesh, предназначенного для хранения вспомогательных структур, типа значений вспомогательных функций на границах областях.
edgeauxmeshclass.f90 - файл с описанием класса tEdgeAuxMesh, предназначенного для хранения вспомогательных структур на ребрах расчетной области.

Модули ключевых классов
problemclass.f90 - файл с описанием класса tProblem, отвечающего за расчет одной задачи.
simpleABCs.f90 - файл имплементацией ABC низкого порядка для класса tProblem.
GivoliNetaABCs.f90 - файл имплементацией Givoli-Neta граничного условия для класса tProblem.
HagstromWarburtonBasicABCs.f90 - файл имплементацией Hagstrom-Warburton граничного условия для класса tProblem.
solutionclass.f90 - файл с описанием класса tSolution, отвечающего за комплексный расчет, состоящий из нескольких взаимодействующих задач.

Специальные классы и модули
bufferclass.f90 - файл с описанием класса tBuffer, предназначенного для передачи информации между задачами.
poissonclass.f90 - файл с описанием класса tPoisson, который хранит и обрабатывает данные, связанные с решением задачи Пуассона для вспомогательных токов.
pmlclass.f90 - файл с имплементацией Usplit PML.
fftsg.f, fftsg3d.f - файлы с имплементацией прямого и обратного преобразования Фурье.

2. Компиляция и запуск программы

Программа предназначена для компилирования в командной строке с использованием компилятора Intel Fortran 17.0: https://software.intel.com/en-us/node/677889. Для того, чтобы скомпилировать программу, первоначально необходимо скомпилировать подряд все ее модули в определенном порядке.

В FORTRAN имеется возможность модульной компиляции, однако для удобства использования, все модули явно перечислены в главной программе, что заставляет компилятор проверять и перекомпилировать все модули программы при компиляции главной программы main.f90. Такая реализация увеличивает время компиляции программы, однако позволяет не думать о ручном перекомпилировании всех измененных модулей каждый раз перед запуском программы, а просто компилировать только главную программу.

Тем не менее, до первого использования программы, необходимо провести компиляцию всех модулей. Для этого необходимо запустить командную строку на машине с установленным компилятором от Intel и выполнить следующие команды:

```
ifort -c -qopenmp commonvars.f90
```

```
ifort -c -qopenmp dipole.f90
ifort -c -qopenmp sdipole.f90
ifort -c -qopenmp newdipole.f90
ifort -c -qopenmp timersclass.f90
ifort -c -qopenmp meshclass.f90
ifort -c -qopenmp auxmeshclass.f90
ifort -c -qopenmp edgeauxmeshclass.f90
ifort -c -qopenmp bufferclass.f90
ifort -c -qopenmp parallel.f90
ifort -c -qopenmp poissonclass.f90
ifort -c -qopenmp sourceclass.f90
ifort -c -qopenmp pmlclass.f90
ifort -c -qopenmp problemclass.f90
ifort -c -qopenmp solutionclass.f90
ifort -o main main.f90 fftsg.f fftsg3d.f -qopenmp -mkl
ifort -o main main.f90 fftsg.f fftsg3d.f -qopenmp -mkl
```

Обратите внимание, что главная программа в конце компилируется дважды. Если во время прогона компиляции всех модулей возникают ошибки, то рекомендуется просто повторить компиляцию модулей с самого начала по указанному списку.

После того, как список скомпилирован, последующая компиляция всех модулей не требуется. Достаточно компилировать дважды только главную программу после изменения любых модулей программы:

```
ifort -o main main.f90 fftsg.f fftsg3d.f -qopenmp -mkl
ifort -o main main.f90 fftsg.f fftsg3d.f -qopenmp -mkl
```

Модули `fftsg.f fftsg3d.f` написаны на более ранней версии языка FORTRAN и должны быть использованы при компиляции основного модуля каждый раз, иначе программа их не увидит. Параметры `-qopenmp, -mkl` используются для подключения соотстветствующих библиотек.

Для запуска программы необходимо запустить исполняемый файл ./main (main.exe на Windows). Если запуск прошел успешно, то в командной строке должны появиться

3. Классы

Концептуально программа состоит из нескольких классов, которые взаимодействуют между собой и обеспечивают гибкость структуры программы:

3.1 tMesh (meshclass.f90)

`tMesh` - вспомогательный класс для хранения значений поля и токов на сетке. Так как значения хранятся в динамических массивах, то класс обеспечивает выделение и освобождение памяти, а также позволяет создавать массивы нужного размера и с нужной индексацией.
Этот класс содержит в себе следующие основные элементы:

- Динамические массивы компонент поля: `x, y, z;`
- Тип индексации `shifttype`. Электрические и магнитные поля расположены в разных точках пространства, поэтому для их описания требуются массивы с разной индексацией. Этот параметр отвечает за тип индексации:
  0 - электрическое поле и токи с массивами `x(1:Nx, 0:Ny, 0:Nz), y(0:Nx, 1:Ny, 0:Nz), z(0:Nx, 0:Ny, 1:Nz)`
  1 - магнитное поле с массивами `x(0:Nx, 1:Ny, 1:Nz), y(1:Nx, 0:Ny, 1:Nz), z(1:Nx, 1:Ny, 0:Nz)`
- Дублирующие массивы `chx, chy, chz`, предназначенных для учета обновления всех узлов в разных алгоритмах (можно не использовать).

Данный класс обеспечивает:

- Выделение и освобождение памяти под массивы с данными;
- Очистку массивов;

3.2 tSource (sourceclass.f90)

`tSource` - вспомогательный класс, обеспечивающий единый интерфейс к модулям с аналитическим решениями. В классе храниться тип источника (0 - бездивергентный, 1 - не бездивергентный), а также функции getEpoint, getHpoint и getJpoint, с помощью котороых можно получить значения электрического, магнитного полей и токов в заданной точке и момент времени, которые беруться из соответствующего аналитического решения.

3.3 tProblem (problemclass.f90)

`tProblem` - основной класс, который отвечает за решение задачи с определенным простым граничным условием.
Этот класс содержит в себе следующие основные элементы:

- Описание сетки, на которой производится расчет в виде массивов с координатами `xi, yi, zi, ti, xi05, yi05, zi05, ti05 ;`
- Значения компонент всех полей и токов в расчетной области в объектах типа `tMesh: Ef, Hf, Je, Jh;`
- Значение компонент всех полей на предыдущих временных слоях в объектах типа `tMesh: Efold, Efold2, Hfold, Hfold2 ;`
- Значения компонент всех полей, вычисленных точно, хранящихся в объектах типа `tMesh: Ean, Han;`
- Источник поля `tSource::Source`, который обеспечивает получение точных значений всех полей для вычисления ошибок и значения токов;

- Значения максимальных абсолютных значений относительной ошибки по каждой компоненте поля: `Ex_err, Ey_err, Ez_err, Hx_err, Hy_err, Hz_err` и их положение на сетке.
- Все вспомогательные величины для граничных условий: значения вспомогательных функций на границах, параметры граничных условий.

Данный класс обеспечивает:

- Хранение значений электромагнитного поля внутри расчетной области в соответствующих объектах класса `tMesh`.
- Получение значений токов: `getSourceCurrents` ;
- Продвижение решения на один шаг с использованием заданного источника во внутренней области: `DoIndependentStep` ;
- Замыкание расчетной области с использованием заданного граничного условия и его параметров `fillEboundary, fillHboundary` ;
- Вычисление точного аналитического решения в узлах сетки: `getAnalyticalSolution`
- Вычисление максимумов абсолютных значений относительной ошибки по каждой компоненте поля: `getCurrentError`.

Фактически, при помощи данного объекта можно посчитать задачу с простым граничным условием.

3.4 tSolution (solutionclass.f90)

`tSolution` - главный управляющий класс, который создан для обеспечения взаимодействия нескольких объектов класса `tProblem`. Небходимость в этом возникает, когда необходимо использовать сложное граничное условие, например, метод лакун.

Этот класс содержит в себе следующие основные элементы:

- Настройки расчетной области и настройки для основной и вспомогательных задач;
- Основная задача `tProblem::mainproblem` .
- Набор вспомогательных задач `tProblem::auxproblems(:)` , при помощи которых производится расчет в определенные промежутки времени;

Данный класс обеспечивает:

- Инициализацию и продвижение основной задачи задагнное количество временных шагов: `solution_DoStep`.
- Добавление, удаление и управление вспомогательными задачами: `addAuxProblem, dropAuxProblem, manageAuxProblem;`
- Вывод информации на экран и в файлы в нужный момет времени.

Если в качестве граничного условия используется просто условие, то объект `tSolution` управляет только одним объектом класса `tProblem`. Объект `tSolution` инициализирует объект `tProblem`, вызывает функцию продвижения решения на один шаг заданное количество шагов, выводит результаты работы и в конце концов удаляет объект `tProblem`. Если используется метод лакун, то объек `tSolution` создает и удаляет вспомогательные задачи (объекты `tProblem`) в нужные моменты времени, обеспечивает их продвижение по временным шагам, передает значения между этими объектами (например, значения полей или токов), а

также обеспечивает продвижение основной задачи `tProblem::mainproblem` по времени, в которой значения полей складываются из значений полей вспомогательных задач.

3.5 tBuffer (bufferclass.f90)

`tBuffer` - вспомогательный класс, который обеспечивает передачу данных между объектами класса `tProblem`. Так как в FORTRAN объект не может хранить ссылку на класс, членом которого он является (объект `tProblem` не может хранить ссылку на объект типа `tSolution`), то возникает необходимость создания дополнительного класса, ссылка на который будет храниться во всех объектах `tProblem`. Объекты `tProblem` могут записывать данные и считывать их из объекта `tBuffer`, тем самым, передавая значения токов и полей друг другу (например, из вспомогательных задач в главную).
Этот класс содержит в себе объекты класса `tMesh`, предназначенные для передачи между `tProblem` соответствующих данных.

3.6 tPML (pmlclass.f90)

`tPML``(pmlclass.f90)` - класс, в котором реализован Unsplit uniaxial PML. В программе PML отделен от основной расчетной области для гибкости конструкции, поэтому чтобы задействовать PML, в него сначала нужно передать приграничные точки из расчетной области, затем просчитать все внутри PML и вернуть граничные точки обратно в расчетную область. Здесь не нужен tBuffer, так как объект класса tPML является членом класса tProblem, а значит объект tProblem может напрямую записать в tPML все данные и взять их обратно. В этом классе работа всех функций должна быть ясна из названия методов: обновление разных полей внутри PML, а также заполнение граничных точек самого PML.

3.7 tPoisson (poissonclass.f90)

`tPoisson``(poissonclass.f90)` - класс, предназначенный для решения задачи Ноймана для скалярного потенциала, а также последующий расчет новых значения поля внутри расчетной области. Все эти функции вынесены в этот объект для удобства.

3.8 Дополнительные классы

`tAuxMesh``(auxmeshclass.f90)` - то же самое, что `tMesh`, только предназначен для хранения значения вспомогательных функций на гранях расчетной области. Здесь просто немного иначе устроены массивы данных, чем в `tMesh`.
`tEdgeAuxMesh``(edgeauxmeshclass.f90)` - то же самое, что `tMesh`, только предназначен для хранения значения вспомогательных функций на ребрах расчетной области. Здесь просто немного иначе устроены массивы данных, чем в `tMesh`.
`tTimer``(timerclass.f90)` - вспомогательный класс, предназначенный для учета времени работы вспомогательных задач в методе лакун.

Следующие классы сделаны в общем модуле commonvars.f90 для простоты, чтобы они были видны во всех модулях:

tProblemStarter(commonvars.f90) - вспомогательный тип, предназначенный для инициализации объектов типа tProblem. Просто записываем в этот объект все параметры и передаем его в конструктор объекта tProblem.
tSolutionStarter(commonvars.f90) - аналогичный тип, предназначенный для инициализации объектов типа tSolution.
tFieldError(commonvars.f90) - тип, предназначенный для хранения максимальной абсолютной (absval) и относительной (val) ошибок, ее координат, а также значения поля в этой точке (fval).

4. Особенности имплементации

Концептуально, работа с данными происходит в объекте класс tProblem. Главная задача и вспомогательные задачи являются объектами одного класса, что позволяет добавить любое новое граничное условие в класс tProblem и использовать его отдельно в главной задаче или с лакунами, имея идентичную имплементацию. Объект tSolution отвечает за создание и уничтожение объектов класса tProblem и взаимодействием этих объектов.

4.1 Простое граничное условие

Для того, чтобы построить решение, нужно сначала создать объект tSolution. У этого объекта довольно много параметров, описывающих расчетную область и граничное условие. Чтобы их передать, в начале создаем объект типа tSolutionStarter (он называется `pst` в модуле `main`). Записываем в него значения всех параметров. Кроме этого создаем объект типа tBuffer (`pbuffer` в модуле `main`) на случай, если будут использоваться лакуны. Затем создаем объект tSolution вызовом функции

```
call sol1%solution_init(pst, pbuffer);
```

Внутри этого конструктора инициализируется много всяких параметров расчетной области и самого расчета. В зависимости от выбранного граничного условия, внутри объекта класса tSolution создается объект класса tProblem и инициализируются его параметры: также настройки расчетной области и граничного условия. Если для работы выбранного граничного условия нужно создание каких-то дополнительных объектов или инициализация параметров, то все это происходит в момент создания объекта tProblem (например, создается PML или какие-то вспомогательные массивы для граничного условия Givoli-Neta). Здесь же создается объект класса tSource, обеспечивающий получение значений токов и точного решения.

Итак, у нас есть объект класса tSolution и в нем объект класса tProblem. Далее идет расчет всех временных шагов. В главной программе вызывается метод построения решения:

```
call sol1%solution_build;
```

В случае простого граничного условия, в этом методе просто Nt раз вызывается последовательность шагов:

- Получить значения аналитического решения:
  `call this%mainproblem%getAnalyticSolution(t);`
- Продвижение решения на один шаг в объекте tProblem:
  `call this%solution_DoStep(t);`
  Здесь в зависимости от типа граничного условия, вызываются разные функции. Если граничное условие простое, то здесь просто вызывается функция, обеспечивающая независимое продвижение одной задачи `propagateSingleProblem(t)`
- Вычисление ошибок:
  `call this%mainproblem%getCurrentError(t);`
- Вывод данных в файл:
  `call this%solution_WriteOutput(t);`
- Вывод данных на экран:
  `call this%solution_printstep(t, ttime);`

После того, как решение было продвинуто на `Nt` шагов, все созданные объекты удаляются и память очищается: `call sol1%solution_destroy;`

Схематично расчет показан ниже:

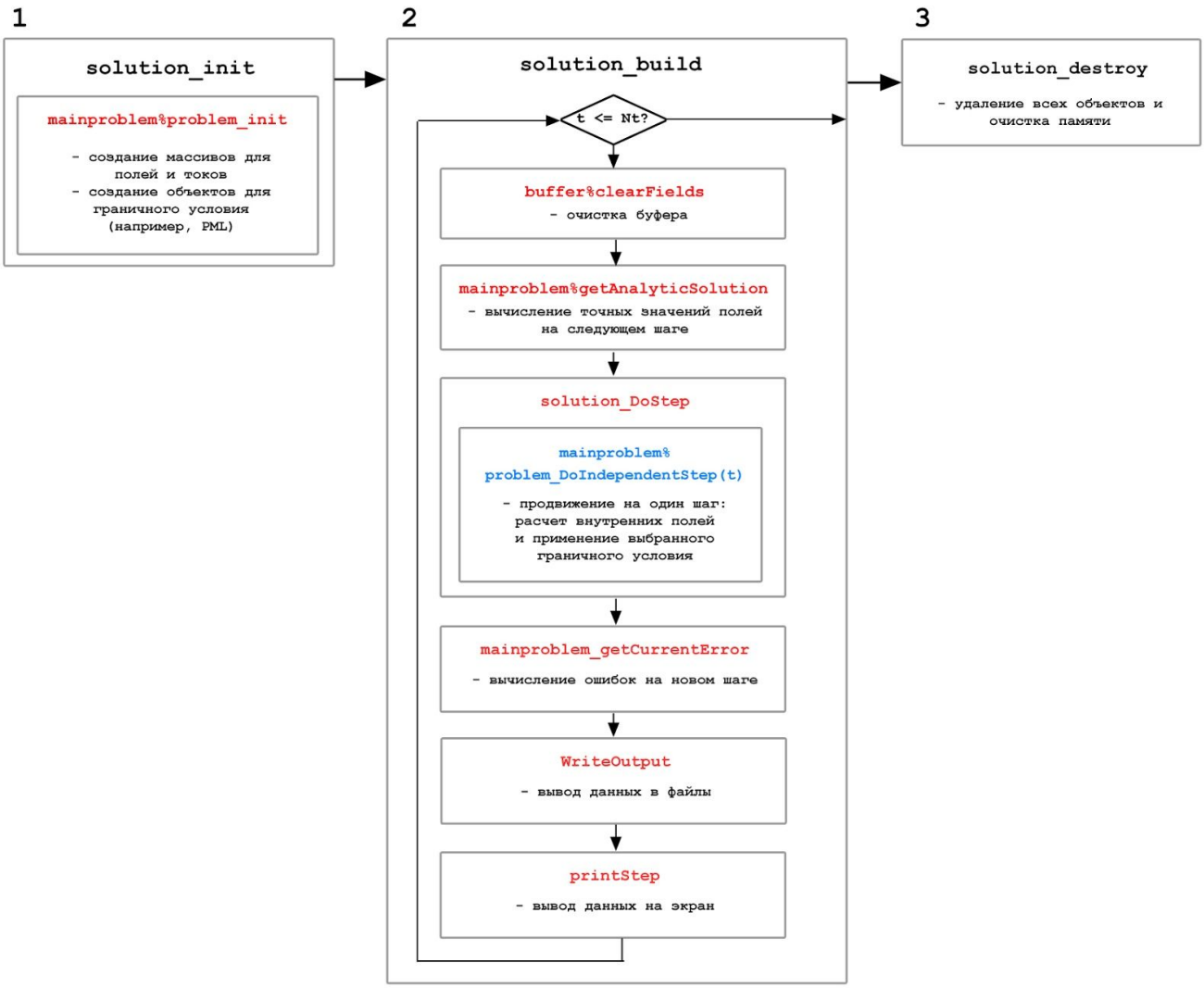

Рисунок 1. Схема простого расчета.

4.2 Сложное граничное условие

Под сложным граничным условием понимается применение метода лакун, когда появляются вспомогательные задачи. В этом случае, алгоритм в целом выглядит также, как предыдущий, однако появляются небольшие модификации:

В начале также создается объект tSolution:

```
call sol1%solution_init(pst, pbuffer);
```

Внутри этого конструктора также инициализируются много параметров расчетной области и самого расчета. Создается главная задача `mainproblem` (как и прежде), а далее инициализируются вспомогательные задачи `auxproblems`. Для оперирования вспомогательными задачами используются методы:

- `AddAuxProblem()` - создание новой вспомогательной задачи в массиве `auxproblems` и ее инициализация.
- `DropAuxProblem(Num)` - удаление вспомогательной задачи под номером `Num` в массиве `auxproblems`.
- `ManageAuxProblems()` - ведение учета вспомогательных задач: добавление новых задач в нужное время и удаление старых, отработавших.

Итак, в этом случае, у нас есть объект класса tSolution и в нем объект `mainproblem` - главная задача, и вспомогательные задачи в массиве `auxproblem`, которые можно создавать и удалять. Далее идет расчет всех временных шагов аналогичным образом. В главной программе вызывается метод построения решения:

```
call sol1%solution_build;
```

В этом методе просто Nt раз вызывается последовательность тех же самых шагов:

1. Получить значения аналитического решения и токов:
   ```
   call this%mainproblem%getAnalyticSolution(t);
   ```
2. Продвижение решения на один шаг в объекте tProblem:
   ```
   call this%solution_DoStep(t);
   ```
   Именно в этом методе заложено различие простого и сложного граничных условий. В случае сложного решения, здесь происходит:
   - Обновление основной задачи во внутренней области;
   - Решение задачи Пуассона;
   - Построение вспомогательных токов;
   - Передача вспомогательных токов в буфер;
   - Продвижение всех вспомогательных задач на один шаг;
   - Сбор значений полей всех вспомогательных задач в виде суммы в буфер;
   - Обновление границы основной задачи;
3. Вычисление ошибок:
   ```
   call this%mainproblem%getCurrentError(t);
   ```
4. Вывод данных в файл:
   ```
   call this%solution_WriteOutput(t);
   ```
5. Вывод данных на экран:
   ```
   call this%solution_printstep(t, ttime);
   ```

После того, как решение было продвинуто на `Nt` шагов, все созданные объекты удаляются и память очищается:

```
call sol1%solution_destroy;
```

Для учета времени работы разных вспомогательных задач, используется класс tTimer. В tSolution есть массив из таймеров `timers`. Во время создания новой вспомогательной задачи, создается один таймер и заводится на время работы этой задачи. В методе `ManageAuxProblems()` - программа обновляет все таймеры и, как только один из таймеров подходит к концу, убивает соответствующую задачу.

4.3 Обновление полей

Для того, чтобы сделать структуру программы гибкой, обновление всех массивов разнесено на следующие функции:

- обновление области;
- обновление в точке.

Например, можно рассмотреть функции обновления внутренних точек расчетной области в классе tProblem. Внутри функциий `updateEInterior()` и `updateHInterior()` мы пробегаем все точки внутри расчетной области (обновление области). На каждом шаге цикла вызывается функция обновления поля в точке - `updateEpoint()`, `updateHpoint()`. Таким образом, можно менять область, по которой бегаем и процедуру обновления точек в этой области раздельно.

4.4 Один временной шаг задачи

Каждая задача может содержит функцию продвижения на один временной шаг `problem_DoIndependentStep()` , которая состоит из следующих шагов:

- `this%getSourceCurrents(t);` - получаем значения токов из источника;
- `call this%saveFields(0);` - сохраняем значения всех полей на предыдущем шаге (если в параметре 1, то сохраняются значения полей на пред-предыдущем шаге).
- `call this%updateEInterior;` - обновление электрического поля во внутренней области.
- `call this%updateHInterior;` - обновление магнитного поля во внутренней области.
- `call this%fillEBoundary(t);` - заполнение граничных точек электрического поля с использованием простого граничного условия;
- `call this%fillHBoundary(t);` - заполнение граничных точек магнитного поля с использованием простого граничного условия;

Это основная структура. В процедуре есть и дополнительные шаги, когда используются специфические граничные условия:

1. Когда мы хотим использовать PML, причем не разделять PML и основную область, то используются следующие функции для получения токов и обновления полей во всей области и PML:

- `call this%getSourceCurrentsPML(t);`
- `call this%PML%updateEInteriorPMLEverywhere;`
- `call this%PML%updateHInteriorPMLEverywhere;`

2. Если используются граничные условия GivoliNeta, Hagstrom-Warburton, то значения вспомогательных функций на предыдущем шаге в них сохраняются при помощи функций:

- `call this%saveGivoliNetaAuxVariables`
- `call this%saveHWAuxVariables`
- `call this%saveHWEvaAuxVariables`

3. Если используется стандартный PML, то сначала нужно загрузить приграничные точки в объект класса tPML, а затем обновить поля внутри этого объекта:

- `call this%uploadFieldsToPML;`

- `call this%PML%pml_DoStep(t);`

4. Если используется какое-то специфическое граничное условие, вместо `fillEboundary()` используется одна из следующих функций:

- `call this%fillEBoundaryABCSommerfeld(t);`
- `call this%fillEBoundaryABCHigdon(t);`
- `call this%fillEBoundaryABCBetzMittra(t);`
- `call this%fillEBoundaryABCMur(t);`
- `call this%fillEBoundaryABCGivoliNeta(t);`
- `call this%fillEBoundaryABCGivoliNetaPML(t);`
- `call this%fillEBoundaryABCGivoliNetaPure(t);`
- `call this%fillEBoundaryABCHW(t);`
- `call this%fillEBoundaryABCHWPML(t);`
- `call this%fillEBoundaryABCHWEva(t);`
- `call this%fillEBoundaryABCHWEvaPML(t);`

В случае использования лакун, для основной задачи нельзя пользоваться методом `DoIndependentStep()` . В этом случае, обновление полей в основной задаче вынесено на уровень объекта tSolution, в методе `PropagateLacunaesPoisson()` .

4.5 PML

В программе реализован Unsplit uniaxial PML. Особенность имплементации заключается в том, что внутренняя расчетная область и PML разделены на разные объекты. Основная расчетная область храниться в объекте tProblem. Внутри этого объекта происходит обновление полей во внутренней области. После этого, приграничные точки должны быть переданы в объект tPML при помощи функции `uploadFieldsToPML()` . Далее объект tPML сам обновляет поля внутри PML слоя при помощи функции `PML%pml_DoStep(t)` .
Важно, что поля в объекте tPML хранятся просто в виде большой коробки размером N+2P, где N - размер внутренней области, а P - толщина PML. В стандартном алгоритме, внутренние точки в PML объекте вообще не обновляются.

Схема работы с PML показана на рисунке 2. Сначала внутри объекта tProblem обновляются точки, показанные красным. Затем, приграничные точки передаются в tPML и там обновляются внутри tPML - желтая область. Граница tPML не обновляется, что показано белыми точками. В конце расчета, значения в граничных точках для основной расчетной области (синие линии) в объекте tProblem беруться из объекта tPML.

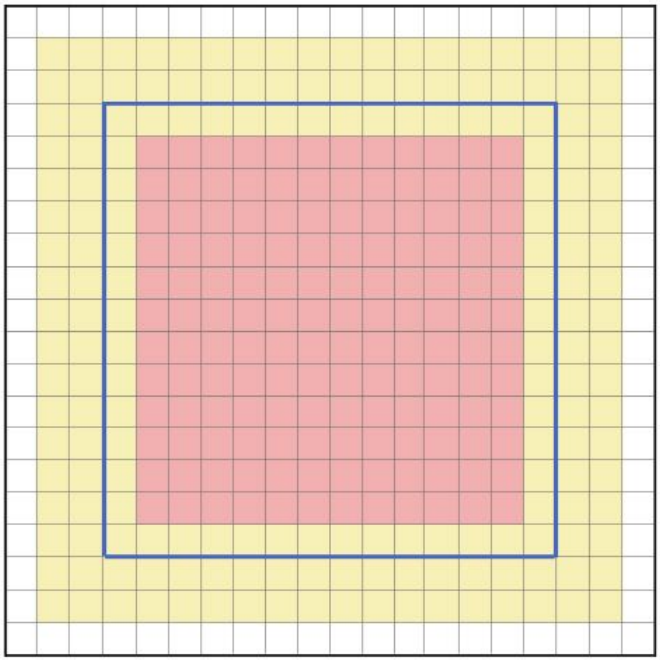

Рисунок 2. Схема расчетной области с PML.

Так как разделение основной расчетной области и PML несколько искусственно, то в программе реализована возможность использовать всю область расчета в объекте tPML (красная И желтая область). То есть, есть основная задача tProblem со своей расчетной областью (красная) и у этого объекта есть объект tPML с большой расчетной областью (красная + желтая). Мы можем обновлять внутреннюю область в основной задаче с помощью токов, но можем делать это параллельно и в объекте tPML. Для этого только необходимо загружать токи в объект tPML при помощи функции `getSourceCurrentsPML()`, а уже в самом PML обновлять поля во всей области при помощи функций `updateEInteriorPMLeverywhere()`, `updateHInteriorPMLeverywhere()`. Такой PML с обновлением всей расчетной области используется в экспериментах с граничными условиями Givoli-Neta и Hagstrom-Warburton, где объект tPML служит поставщиком граничных условий для всех остальных границ кроме той, на которой используются эти граничные условия.

4.6 Граничное условия Givoli-Neta и Hagstrom-Warburton

В программе реализованы граничные условия Givoli-Neta и Hagstrom-Warburton. При этом, данные граничные условия ставятся только на грани Ey_x=0, так как полноценной 3D имплементации у этих граничных условий нет. При этом, в программе есть несколько возможностей для заполнения остальных граней:

- Использовать точное условие;
- Использовать Unsplit PML.

В первом случае, границы просто заполняются значениями из объекта tSource. Во втором случае одна задач по сути разделяется на две внутри одного объекта tProblem. Одна задача

считается в основных массивах объекта, а вторая считается в объекте tPML. Во втором объекте задача полноценно рассчитывается с PML. После этого, в самом объекте tProblem на одной грани ставится Givoli-Neta или Hagstrom-Warburton, а остальные грани заполняются значениями из соответствующих точек в объекте tPML:

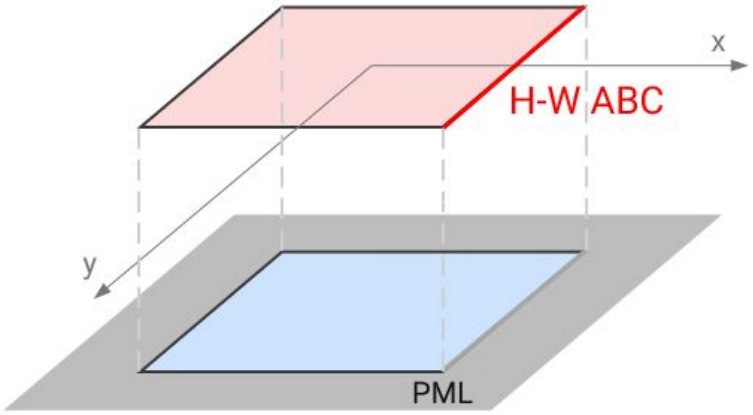

Рисунок 3. Схема работы задачи с H-W ABC на одной грани и PML на остальных.

Если вся эта конструкция используется в методе лакун, то подобным образом разделяются все вспомогательные задачи:

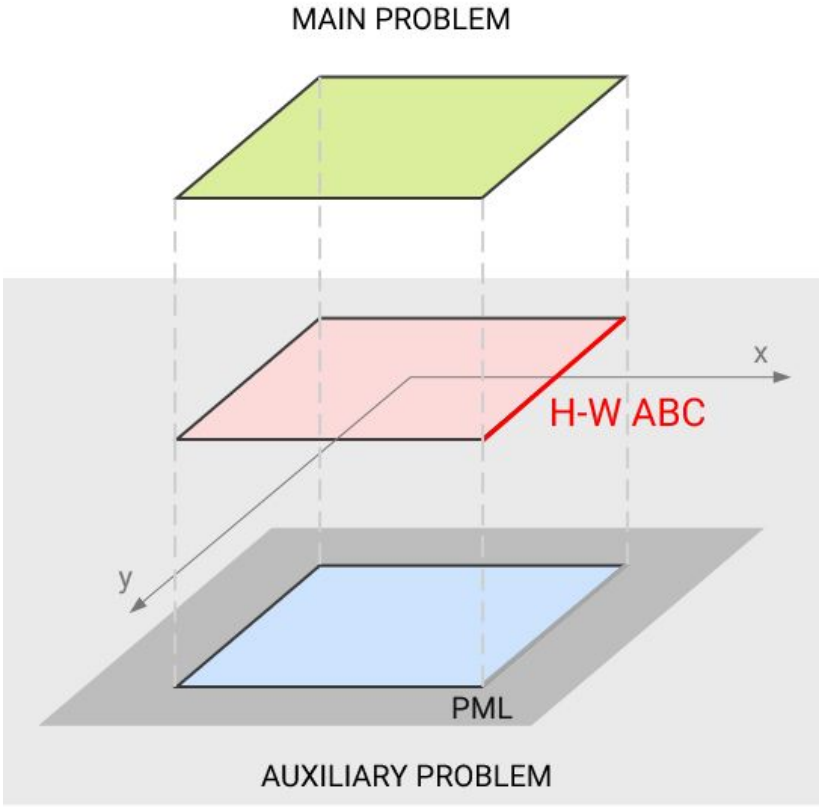

Рисунок 4. Схема работы задачи с H-W ABC на одной грани и PML на остальных гранях в методе лакун.

Схема расчета в этом случае показана на следующем рисунке:

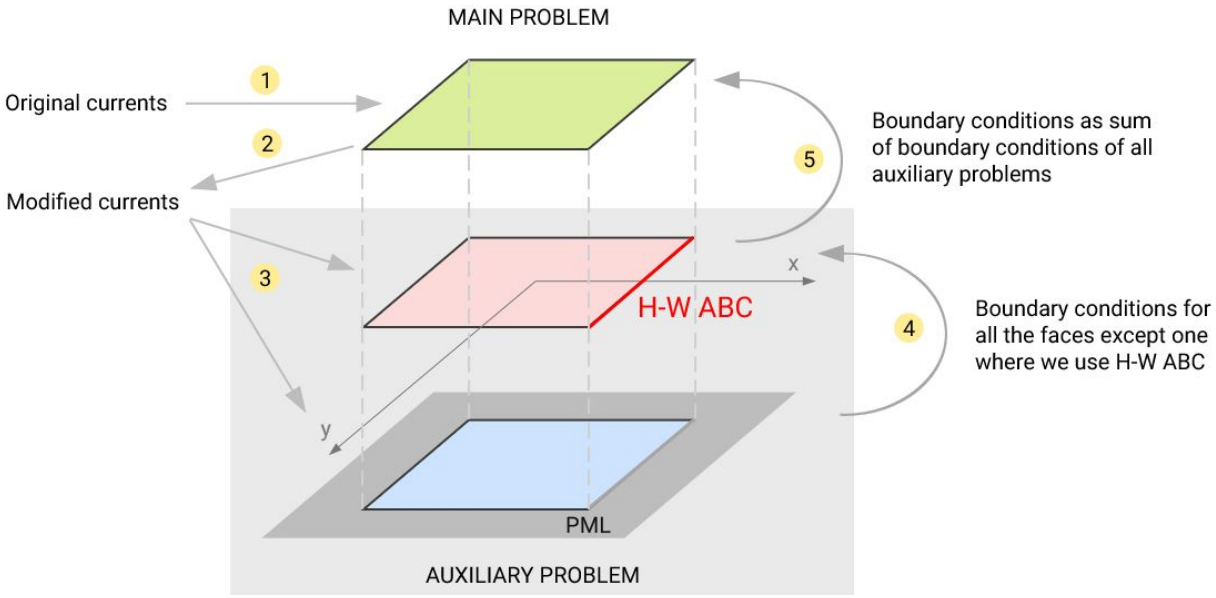

Рисунок 5. Схема расчета задачи с H-W ABC на одной грани и PML на остальных гранях в методе лакун.

5. Работа с программой

5.1 Общие параметры

Основная функциональность заключается в моделировании электромагнитных волн в ограниченной области с использованием различных граничных условий и источников. Основная программа состоит из запуска одной функции:

```
call DoSimpleFieldModelling();
```

Внутри этой функции нужно задать параметры моделирования:

Параметр	**Значение**	**Описание**
Параметры для бездивергентного источника:		
a	0.01	amplification multiplier of dipole
v	1	amplification frequency of the dipole
p	0.5	smoothing factor of dipole field
q	0.1	smoothing factor of amplification sin
Параметры для НЕ бездивергентного источника:		

alphaaa	22.0d0	
om	1.0d0	
Параллельность и номер сетки		
doparallel	0 | 1	В программу внедрена возможность распараллеливания вычислений на машинах с разделяемой памятью, что регулируется переключателем: (0 - без параллельности, 1 - параллельность на всех доступных ядрах)
conv	1 | 2 | 4 | 8	Так как часто приходится исследовать посеточную сходимость, то необходимо установить параметр conv. Если этот параметр равен единице, то решение строится на заданной ниже сетке. Если он равен N, что количество точек в расчетной области увеличивается в N раз. Таким образом, можно исследовать сходимость, устанавливая этот параметр равным 1, 2, 4 и так далее.
Размеры расчетной области и время моделирования		
xsize	5.0d0	размер по оси X
ysize	5.0d0	размер по оси Y
zsize	5.0d0	размер по оси Z
cntx	0.5d0	положение начала координат относительно рассчетной области по оси X, величина от 0 до 1.
cnty	0.5d0	положение начала координат относительно рассчетной области по оси Y, величина от 0 до 1.
cntz	0.5d0	положение начала координат относительно рассчетной области по оси Z, величина от 0 до 1.
ftime	1000	время моделирования в единицах
Настройки вывода информации в файлы и на экран		
screen_erroroutputsteps	10*conv	шаг вывода ошибок на экран (если 10, то ошибка будет выводиться на экран каждый десятый шаг)
file_erroroutputsteps	10*conv	шаг вывода ошибок в файл (если 10, то ошибка будет выводиться на экран каждый десятый шаг)

file_pmlerrors	0 | 1	так как PML реализован как отдельный объект со внутренней областью, то в нем тоже можно считать ошибку во внутренней области и выводить в файлы. Этот параметр - переключатель, выводить ошибку (1) в PML или нет (0).
file_writelocations	0 | 1	В файлах с ошибками по каждой компоненте поля можно выводить координаты максимальной абсолютной ошибки. Если 1, то координаты выводятся, если 0, то нет.
file_auxproblemsoutput	0 - 9	Программа позволяет выводить максимальные абсолютные значение полей во вспомогательных задачах в методе лакун. Здесь указывается количество первых вспомогательных задач, по которым будет выводиться значение полей. Можно указать до 9 вспомогательных задач.
file_auxproblemsoutputstep	1*conv	С каким шагом выводить значения полей во вспомогательных задачах. Если здесь 1, то будут выводиться значения полей на каждом шаге.
file_auxproblemswritelocations	0 | 1	Можно включить или выключить вывод не только максимального значения полей, но и координаты максимума каждой компоненты поля.
file_compositedivergence	0 | 1	Выводить в файл или нет абсолютный максимум дивергенции электрического композитного поля.
file_staticfieldmax	0 | 1	Выводить в файл или нет абсолютный максимум статического поля во вспомогательных задачах в методе лакун.
file_mainfields	0 | 1	Выводить в файл или нет абсолютный максимум по каждой компоненте поля в основной задаче.
file_mainpmlfields	0 | 1	Выводить в файл или нет абсолютный максимум по каждой компоненте поля в PML.

5.2 Настройка объекта Solution

Основным объектом программы является объект Solution, который предназначен для проведения численного моделирования с заданным граничным условием.

В основной программе после задания общих параметров идет секция
`! Solution initialization`
в которой происходит инициализация объекта Solution. Так как для инициализации требуется большое количество параметров, то все эти параметры передаются через объект
`pst (TSolutionStarter)`
в котором и нужно указать все требуемые параметры перед запуском расчетов.

Первый параметр `pst%sid` - это просто ID объекта, который нужен, если мы хотим строить несколько решений одновременно. Так как в простом случае моделирование только одно, то просто присваиваем этому параметру 0.

Далее идут все параметры, определяющие граничные условия. Ниже представлено описания каждого граничного условия, которые доступные в программе и дополнительные параметры, которые нужно задать, чтобы граничное условие работало корректно.

В качестве граничных условий для электрического поля можно использовать следующие:

0. Точное решение в качестве граничного
`pst%soltype = 0`
В этом случае в качестве граничного условия используется точное аналитическое решение задачи. Никаких дополнительных параметров здесь не требуется.

1. Unsplit PML
`pst%soltype = 1`
В этом случае в качестве граничного условия используется Unsplit uniaxial PML (с разделением внутренней задачи и области PML). Никаких дополнительных параметров здесь не требуется.

2. AuxTest 1 - One auxiliary problem with analytical bounds that gives boundary points for the main problem
`pst%soltype = 2`
Это тестовое решение, чтобы проверить, как работает вспомогательная задача. Создается две задачи - главная и одна вспомогательная. Обе задачи обновляются в расчете. В качестве ГУ для вспомогательной задачи используется точное решение, а в качестве ГУ для главной задачи используются точки из вспомогательной задачи. При этом вспомогательная задача по размеру должна быть больше или равна основной задаче. В качестве источника для вспомогательной задачи используется основной источник.

3. AuxTest 2 - One auxiliary problem with PML
`pst%soltype = 3`
Это второе тестовое решение, чтобы проверить, как работает вспомогательная задача. Создается две задачи - главная и одна вспомогательная. Обе задачи обновляются в расчете. В качестве ГУ для вспомогательной задачи используется UnsplitPML, а в качестве ГУ для главной задачи используются точки из вспомогательной задачи. При этом вспомогательная задача по размеру должна быть больше или равна основной задаче. В качестве источника для вспомогательной задачи используется основной источник.

4. AuxTest 4 - One auxiliary problem with PML and effective currents
`pst%soltype = 4`
Это третье тестовое решение, чтобы проверить, как работает вспомогательная задача. Создается две задачи - главная и одна вспомогательная. Обе задачи обновляются в расчете. В качестве ГУ для вспомогательной задачи используется UnsplitPML, а в качестве ГУ для главной задачи используются точки из вспомогательной задачи. При этом вспомогательная задача по размеру должна быть больше или равна основной задаче. В качестве источника для вспомогательной задачи используются сформированные вспомогательные токи.

5. LacunasNoPoisson - lacunaes without Poisson and Unsplit PML (только бездивергентный источник!)
`pst%soltype = 5`
Полноценный алгоритм, в котором есть главная задача, а также последовательность из вспомогательных задач, на границах которых стоит Unsplit PML. ГУ для основной задачи получается путем сложения полей вспомогательных задач. Так как корректировки алгоритма при помощи вспомогательных токов нет, то все вспомогательные задачи добавляют в буффер значения своих статических полей, которые остаются на дне лакуны. Алгоритм так и не заработал.

6. LacunasPoisson - lacunaes with Poisson and Unsplit PML
`pst%soltype = 6`
Это работающий алгоритм с лакунами, решением задачи Ноймана и Unsplit PML во вспомогательных задачах.

7. PMLwithoutSeparation
`pst%soltype = 7`
Как было описано в разделе 4.5, Unsplit PML реализован в виде отдельного объекта, в который из расчетной области передаются значения приграничных точек, затем поля досчитываются в PML и уже значения для граничных точек главной задачи передаются назад в главную задачу из этого объекта. Это седьмое ГУ сделано так, чтобы убрать это разделение. Фактически, задача полностью считается в объекте PML, включая внутреннюю область. Там же происходит расчет ошибок. Это сделано для того, чтобы проверить правильность разделения основной задачи и PML.

8. Sommerfeld ABC (characteristics)
`pst%soltype = 8`
Простейшее граничное условие Зоммерфельда. Алгоритм реализован в функции `tProblem%fillEBoundaryABCSommerfeld()`.

9. Higdon ABC
`pst%soltype = 9`
Простейшее граничное условие Higdon с двумя углами. Алгоритм реализован в функции `tProblem%fillEBoundaryABCHigdon()`.

10. Betz-Mittra ABC
`pst%soltype = 10`
Простейшее граничное условие Betz-Mittra с двумя углами. Алгоритм реализован в функции

`tProblem%fillEBoundaryABCBetzMittra()`.

11. Mur ABC

`pst%soltype = 11`

Простейшее граничное условие Mur, при этом на углах и ребрах используется Sommerfeld. Алгоритм реализован в функции `tProblem%fillEBoundaryABCMur()`.

12. GivoliNeta + Exact Elsewhere - givoli-neta abc ey_X=0, exact elsewhere

`pst%soltype = 12`

Простая имплементация Givoli-Neta ABC. Условие это используется только для компоненты Ey и ставится на грань x=0. На остальных гранях используется точное решение. Алгоритм реализован в функции `tProblem%fillEBoundaryABCGivoliNeta()`.

13. GivoliNeta + PML - givoli-neta abc ey_X=0, Unsplit PML elsewhere

`pst%soltype = 13`

Имплементация Givoli-Neta ABC, когда это условие используется только для компоненты Ey и ставится на грань x=0, а на остальных гранях используется Unsplit PML. В этом случае создается объект PML, в котором расчет ведется независимо (см. раздел 4.6). После чего одна грань расчетной области обновляется при помощи Givoli-Neta, а значения на остальных гранях беруться из объекта PML.

14. GivoliNeta + LargeAuxProblem - givoli-neta abc ey_X=0, one large auxiliary problem without boundaries

`pst%soltype = 14`

В процессе исследования ГУ Givoli-Neta возникла мысль, что возможно неустойчивость этого граничного условия связана с неправильными невязками на ребрах и углах той грани, на которой используется Givoli-Neta. Невязки эти могут приходить как из точного решения, так и из PML в предыдущих двух решениях. В данном решении, есть главная задача, где условие Givoli-Neta используется только для компоненты Ey и ставится на грань x=0. Добавляется одна вспомогательная задача очень большого размера без граничного условия, которая может работать устойчиво определенное время, пока волна не дошла до грани, не отразилась и не вернулась в область, которая по размерам совпадает с главной задачей. При этом, значения на границах главной задачи беруться из этой большой вспомогательной задачи. К сожалению, такая конструкция также довольно быстро разваливается просто из-за неустойчивости Givoli-Neta.

15. GivoliNeta + QuasiLacunas - givoli-neta abc ey_X=0, Unsplit PML elsewhere, lacunaes, no poisson, no effective currents (только бездивергентный источник!)

`pst%soltype = 15`

Попытка оживить GivoliNeta алгоритм при помощи метода лакун с довольно короткими вспомогательными задачами. При этом, каждая вспомогательная задача состоит из двух - см. подробности в разделе 4.6. Из-за малых размеров вспомогательных задач, лакуны получаются не глубокими и все довольно быстро разваливается.

16. One large aux problem

`pst%soltype = 16`

Тестовое решение, когда создается главная задача и одна вспомогательная задача очень большого размера без граничных условий. Это ГУ работает некоторое время, пока отраженная

волна не вернется в область вспомогательной задачи, совпадающей с областью главной задачи.

17. Hagstrom-Warburton + Exact Elsewhere - hagstrom-warburton ABC on ey_X=0 and exact solution elsewhere

`pst%soltype = 17`

Простая имплементация Hagstrom-Warburton ABC. Условие это используется только для компоненты Ey и ставится на грань x=0. На остальных гранях используется точное решение. Алгоритм реализован в функции `tProblem%fillEBoundaryABCHW()`.

18. Hagstrom-Warburton + PML - hagstrom-warburton ABC on ey_X=0, Unsplit PML elsewhere

`pst%soltype = 18`

Имплементация Hagstrom-Warburton ABC, когда это условие используется только для компоненты Ey и ставится на грань x=0, а на остальных гранях используется Unsplit PML. В этом случае создается объект PML, в котором расчет ведется независимо (см. раздел 4.6). После чего одна грань расчетной области обновляется при помощи Hagstrom-Warburton, а значения на остальных гранях беруться из объекта PML.

19. Hagstrom-Warburton + QuasiLacunas - hagstrom-warburton ABC on ey_X=0, lacunaes, no poisson, no effective currents (только бездивергентный источник!)

`pst%soltype = 19`

Попытка оживить Hagstrom-Warburton алгоритм при помощи метода лакун с довольно короткими вспомогательными задачами. При этом, каждая вспомогательная задача состоит из двух - см. подробности в разделе 4.6. Из-за малых размеров вспомогательных задач, лакуны получаются не глубокими и все довольно быстро разваливается.

20. Hagstrom-Warburton (with evanescence waves) + Exact Elsewhere - HWeva ABC on Ey_X=Nx, Exact solution elsewhere

`pst%soltype = 20`

В более поздней статье, авторы Hagstrom и Warburton добавили в граничное условие учет затухающих волн. В программе, имплементация такого граничного условия названа HWeva. Это решение использует простую имплементацию HWeva. Условие это используется только для компоненты Ey и ставится на грань x=0. На остальных гранях используется точное решение. Алгоритм реализован в функции `tProblem%fillEBoundaryABCHWeva()`.

21. Hagstrom-Warburton (with evanescence waves) + PML - HWeva ABC on Ey_X=Nx, Unsplit PML elsewhere

`pst%soltype = 21`

Имплементация HWeva, когда это условие используется только для компоненты Ey и ставится на грань x=0, а на остальных гранях используется Unsplit PML. В этом случае создается объект PML, в котором расчет ведется независимо (см. раздел 4.6). После чего одна грань расчетной области обновляется при помощи HWeva, а значения на остальных гранях беруться из объекта PML.

22. Hagstrom-Warburton (with evanescence waves) + QuasiLacunas - HWeva ABC on Ey_X=Nx, Unsplit PML elsewhere, lacunaes, no poisson, no effective currents, Source 0 only (только бездивергентный источник!)

`pst%soltype = 22`

Попытка оживить HWeva алгоритм при помощи метода лакун с довольно короткими вспомогательными задачами. При этом, каждая вспомогательная задача состоит из двух - см. подробности в разделе 4.6. Из-за малых размеров вспомогательных задач, лакуны получаются не глубокими и все довольно быстро разваливается.

23. Hagstrom-Warburton (with evanescence waves) + Lacunas + Poisson - HWeva ABC on Ey_X=Nx, Unsplit PML elsewhere, lacunaes, Poisson
`pst%soltype = 23`
Полноценная реализация HWeva граничного условия с лакунами и Пуассоном. При этом HWeva ставится только на одну грань, а остальные грани заполняются из дублирующей задачи с PML. Через некоторое время условие все равно разваливается.

25. Sommerfeld ABCe, lacunaes, Poisson
`pst%soltype = 25`
Граничное условие Sommerfeld используется в алгоритме с лакунами и пуассоном. Несмотря на высокий уровень отражений, алгоритм не разваливается.

100. PML+Lacunas, разрезается главный источник - Unsplit PML, lacunaes, no poisson, no effective currents, the main source is cut Source 0 only (только бездивергентный источник!)
`pst%soltype = 100`
Тестовая версия граничного условия с лакунами, только разрезаются не вспомогательные источники, а режется основной источник.

101. Large Auxiliary Problems + Lacunas - Lacunaes, no poisson, no effective currents, Source 0 only (только бездивергентный источник!)
`pst%soltype = 101`
Полноценный метод квази-лакун с огромными вспомогательными задачами без ГУ.

102. Large Auxiliary Problems + Lacunas 2 - Lacunaes, no poisson, no effective currents, Source 0 only (только бездивергентный источник!)
`pst%soltype = 102`
То же самое, что 101, только немного ускорена имплементация.

Таблица 1 - Параметры для каждого алгоритма

pst%soltype	Тип граничного условия	Дополнительные параметры
0	Точное аналитическое решение	-

1	Unsplit PML	Толщина PML в точках и параметр sigma для PML:
		`PML_thickness = 10*conv`
		`PML_param = 0.5d0`

Лакуны		
2	AuxTest 1	Разница в размерах между основной и

		вспомогательной задачами:
		`dNxaux = 0*conv;`
		`dNyaux = 0*conv;`
		`dNzaux = 0*conv;`

3	AuxTest 2	Разница в размерах между основной и вспомогательной задачами:
		`dNxaux = 0*conv;`
		`dNyaux = 0*conv;`
		`dNzaux = 0*conv;`
		Толщина PML в точках и параметр sigma для PML во вспомогательной задаче:
		`PML_thickness = 10*conv`
		`PML_param = 0.5d0`

4	AuxTest 3	Разница в размерах между основной и вспомогательной задачами:
		`dNxaux = 0*conv;`
		`dNyaux = 0*conv;`
		`dNzaux = 0*conv;`
		Толщина PML в точках и параметр sigma для PML во вспомогательной задаче:
		`PML_thickness = 10*conv`
		`PML_param = 0.5d0`
		Размер области, для которой решается задача Ноймана для вспомогательных токов
		`npx = 16*conv-1;`
		`npy = 16*conv-1;`
		`npz = 16*conv-1;`
		Параметры для зоны со вспомогательными токами:
		`Namu = 3*conv;`
		`Ndmu = 4*conv;`

5	LacunasWithoutPoisson (не работает с не бездивергентным источником!)	Разница в размерах между основной и вспомогательной задачами:
		`dNxaux = 0*conv;`
		`dNyaux = 0*conv;`
		`dNzaux = 0*conv;`
		Толщина PML в точках и параметр sigma для PML во вспомогательной задаче:
		`PML_thickness = 10*conv`
		`PML_param = 0.5d0`
		Время работы вспомогательной задачи, sigma для разрезания вспомогательных задачи и дополнительное время работы вспомогательной:
		`auxmaxtime = 40.0d0;`

		sgm_s = 0.8d0; auxgaptime = 10*conv; ! в точках времени

6	LacunasPoisson	Разница в размерах между основной и вспомогательной задачами:
		dNxaux = 0*conv;
		dNyaux = 0*conv;
		dNzaux = 0*conv;
		Толщина PML в точках и параметр sigma для PML во вспомогательной задаче:
		PML_thickness = 10*conv
		PML_param = 0.5d0
		Размер области, для которой решается задача Ноймана для вспомогательных токов
		npx = 16*conv-1;
		npy = 16*conv-1;
		npz = 16*conv-1;
		Параметры для зоны со вспомогательными токами:
		Namu = 3*conv;
		Ndmu = 4*conv;
		Время работы вспомогательной задачи, sigma для разрезания вспомогательных задачи и дополнительное время работы вспомогательной:
		auxmaxtime = 40.0d0;
		sgm_s = 0.8d0;
		auxgaptime = 10*conv; ! в точках времени

7	PMLwithoutDividing	Толщина PML в точках и параметр sigma для PML: PML_thickness = 10*conv PML_param = 0.5d0
8	Sommerfeld ABC	-
9	Higdon ABC	Два угла для алгоритма Higdon: hig1 = 0.0d0 hig2 = 0.0d0
10	Betz-Mittra ABC	Два угла для алгоритма Betz-Mittra: hig1 = 0.0d0 hig2 = 0.0d0
11	Mur ABC	-
12	GivoliNeta + Exact elsewhere	Количество углов для алгоритма Givoli-Neta pst%gnNum = 5

		Сами значения углов задаются равномерно от 0 до 90 градусов в функции `tProblem&initGivoliNetaAuxVariables()`

13	GivoliNeta + PML	Толщина PML в точках и параметр sigma для PML:
		`PML_thickness = 10*conv`
		`PML_param = 0.5d0`
		Количество углов для алгоритма Givoli-Neta
		`pst%gnNum = 5`
		Сами значения углов задаются равномерно от 0 до 90 градусов в функции
		`tProblem&initGivoliNetaAuxVariables()`

14	GivoliNeta + LargeAuxProblem	Разница в размерах между основной и вспомогательной задачами:
		`dNxaux = 0*conv;`
		`dNyaux = 0*conv;`
		`dNzaux = 0*conv;`
		Количество углов для алгоритма Givoli-Neta
		`pst%gnNum = 5`
		Сами значения углов задаются равномерно от 0 до 90 градусов в функции
		`tProblem&initGivoliNetaAuxVariables()`

15	GivoliNeta + QuasiLacunas	Толщина PML в точках и параметр sigma для PML:
		`PML_thickness = 10*conv`
		`PML_param = 0.5d0`
		Разница в размерах между основной и вспомогательной задачами:
		`dNxaux = 0*conv;`
		`dNyaux = 0*conv;`
		`dNzaux = 0*conv;`
		Количество углов для алгоритма Givoli-Neta
		`pst%gnNum = 5`
		Сами значения углов задаются равномерно от 0 до 90 градусов в функции
		`tProblem&initGivoliNetaAuxVariables()`.
		Время работы вспомогательной задачи, sigma для разрезания вспомогательных задачи и дополнительное время работы вспомогательной:
		`auxmaxtime = 40.0d0;`
		`sgm_s = 0.8d0;`
		`auxgaptime = 10*conv; ! в точках времени`

16	One large aux problem	Разница в размерах между основной и вспомогательной задачами:

		dNxaux = 0*conv;
		dNyaux = 0*conv;
		dNzaux = 0*conv;

17	Hagstrom-Warburton + Exact elsewhere	Количество углов для алгоритма Hagstrom-Warburton pst%hwNum = 5 Сами значения углов задаются равномерно от 0 до 90 градусов в функции tProblem&initHWAuxVariables()
18	Hagstrom-Warburton + PML	Количество углов для алгоритма Hagstrom-Warburton pst%hwNum = 5 Сами значения углов задаются равномерно от 0 до 90 градусов в функции tProblem&initHWAuxVariables() Толщина PML в точках и параметр sigma для PML: PML_thickness = 10*conv PML_param = 0.5d0

19	Hagstrom-Warburton + QuasiLacunas	Толщина PML в точках и параметр sigma для PML:
		PML_thickness = 10*conv
		PML_param = 0.5d0
		Разница в размерах между основной и вспомогательной задачами:
		dNxaux = 0*conv;
		dNyaux = 0*conv;
		dNzaux = 0*conv;
		Количество углов для алгоритма Hagstrom-Warburton
		pst%hwNum = 5
		Сами значения углов задаются равномерно от 0 до 90 градусов в функции
		tProblem&initHWAuxVariables().
		Время работы вспомогательной задачи, sigma для разрезания вспомогательных задачи и дополнительное время работы вспомогательной:
		auxmaxtime = 40.0d0;
		sgm_s = 0.8d0;
		auxgaptime = 10*conv; ! в точках времени

20	Hagstrom-Warburton (with evanescence waves) + Exact	Количество углов и затухающих переменных для алгоритма HWeva: pst%hwP = 4 pst%hwA = 0

21	Hagstrom-Warburton (with evanescence waves) + PML	Толщина PML в точках и параметр sigma для PML:
		`PML_thickness = 10*conv`
		`PML_param = 0.5d0`
		Количество углов и затухающих переменных для алгоритма HWeva:
		`pst%hwP = 4`
		`pst%hwA = 0`

22	Hagstrom-Warburton (with evanescence waves) + QuasiLacunas	Толщина PML в точках и параметр sigma для PML:
		`PML_thickness = 10*conv`
		`PML_param = 0.5d0`
		Количество углов и затухающих переменных для алгоритма HWeva:
		`pst%hwP = 4`
		`pst%hwA = 0`
		Количество углов и затухающих переменных для алгоритма HWeva во вспомогательных задачах:
		`pst%hwPa = 4`
		`pst%hwAa = 0`

23	Hagstrom-Warburton (with evanescence waves) + Lacunas + Poisson	Толщина PML в точках и параметр sigma для PML:
		`PML_thickness = 10*conv`
		`PML_param = 0.5d0`
		Разница в размерах между основной и вспомогательной задачами:
		`dNxaux = 0*conv;`
		`dNyaux = 0*conv;`
		`dNzaux = 0*conv;`
		Количество углов и затухающих переменных для алгоритма HWeva:
		`pst%hwP = 4`
		`pst%hwA = 0`
		Количество углов и затухающих переменных для алгоритма HWeva во вспомогательных задачах:
		`pst%hwPa = 4`
		`pst%hwAa = 0`
		Время работы вспомогательной задачи, sigma для разрезания вспомогательных задачи и дополнительное время работы вспомогательной:
		`auxmaxtime = 40.0d0;`
		`sgm_s = 0.8d0;`
		`auxgaptime = 10*conv; ! в точках времени`

25	Sommerfeld ABCe, lacunaes, Poisson	Разница в размерах между основной и вспомогательной задачами: `dNxaux = 0*conv;`

		dNyaux = 0*conv;
		dNzaux = 0*conv;
		Время работы вспомогательной задачи, sigma для разрезания вспомогательных задачи и дополнительное время работы вспомогательной:
		auxmaxtime = 40.0d0;
		sgm_s = 0.8d0;
		auxgaptime = 10*conv; ! в точках времени

100	PML+Lacunas, разрезается главный источник	Толщина PML в точках и параметр sigma для PML:
		PML_thickness = 10*conv
		PML_param = 0.5d0
		Разница в размерах между основной и вспомогательной задачами:
		dNxaux = 0*conv;
		dNyaux = 0*conv;
		dNzaux = 0*conv;
		Время работы вспомогательной задачи, sigma для разрезания вспомогательных задачи и дополнительное время работы вспомогательной:
		auxmaxtime = 40.0d0;
		sgm_s = 0.8d0;
		auxgaptime = 10*conv; ! в точках времени

101	Large Auxiliary Problems + Lacunas	Разница в размерах между основной и вспомогательной задачами:
		dNxaux = 0*conv;
		dNyaux = 0*conv;
		dNzaux = 0*conv;
		Время работы вспомогательной задачи, sigma для разрезания вспомогательных задачи и дополнительное время работы вспомогательной:
		auxmaxtime = 40.0d0;
		sgm_s = 0.8d0;
		auxgaptime = 10*conv; ! в точках времени

102	Large Auxiliary Problems + Lacunas 2	Разница в размерах между основной и вспомогательной задачами:
		dNxaux = 0*conv;
		dNyaux = 0*conv;
		dNzaux = 0*conv;
		Время работы вспомогательной задачи, sigma для разрезания вспомогательных задачи и дополнительное время работы вспомогательной:

		auxmaxtime = 40.0d0;
		sgm_s = 0.8d0;
		auxgaptime = 10*conv; ! в точках времени

5.3 Вывод на экран

При создании любого нового объекта tProblem, на экране появляется описание создаваемого объекта в секции PROBLEM REPORT, из которого можно узнать о том, с какими параметрами был создан объект tProblem. Эта функциональность позволяет проверить правильность работы программы, особенно в методе лакун, где вспомогательные задачи создаются и удаляются периодически. Функция вывода на экран : `problem_report()` . Примерно, эта секция выглядит следующим образом:

```
------PROBLEM REPORT-----
 Problem ID:           0
 Problem Type:           0
 Scheme Type:           0
 Currents Type:           0
 Effective Currents Type:           1
 E boundary:           4
 H boundary:           2
 Nx:          65
 Ny:          65
 Nz:          65
 Nt:       45000
 hhx:  0.100000000000000
 hhy:  0.100000000000000
 hhz:  0.100000000000000
 ht:  3.333333333333333E-002
 Xi(0)  -3.25000000000000
 Xi(Nx)   3.25000000000000
 Yi(0)  -3.25000000000000
 Yi(Nz)   3.25000000000000
 Zi(0)  -3.25000000000000
 Zi(Nz)   3.25000000000000
 Ti(0)  0.000000000000000E+000
 Ti(Nt)   1500.00000000000
 Buffer Offset X:           0
 Buffer Offset Y:           0
 Buffer Offset Z:           0
```

```
 Main dX:           0
 Main dY:           0
 Main dZ:           0
 AuxMaxTime:   40.0000000000000
 SigmaS:  0.800000000000000
 Namu           3
 Ndmu           4
 Nlc           7
 Dlc  0.399999999999999
 Poisson
   Poisson npx:          15
   Poisson npy:          15
   Poisson npz:          15
   Poisson Nin:          25
   Poisson Nx, Ny, Nz:   65,   65,   65
       Poisson   hhx,   hhy,   hhz,   ht:    0.1000000E+00,    0.1000000E+00,
0.1000000E+00,  0.3333333E-01
 ------End Problem Report-----
```

При создании любого нового объекта tSolution, на экране также появляется описание создаваемого объекта. В этот момент вызывается функция `solution_report()`. Примерно такой вывод на экран имеет вид:

```
------SOLUTION REPORT-----
 Solution ID:           0
 Type:           6
 Lacunaes with Poisson correction, aux problems with unsplit PML
 Nx:          65
 Ny:          65
 Nz:          65
 Nt:       45000
 hhx:  0.100000000000000
 hhy:  0.100000000000000
 hhz:  0.100000000000000
 ht:  3.333333333333333E-002
 NXaux:           0
 NYaux:           0
 NZaux:           0
 Scheme: Classic Yee 2-2 scheme
 Source: SDipole
 Domain diameter:   11.2583302491977
 Aux problems standrad life time:        2748
```

```
 Each aux will last more for (gap):          10
 Start and drop times:
           1 Aux task time:           0 :        1548
           2 Aux task time:         960 :        3708
           3 Aux task time:        3120 :        5868
           4 Aux task time:        5280 :        8028
```

После этого программа встает на паузу до нажатия на кнопку Enter. Как только пользователь его нажимает, на экране появляются параметры расчета:

```
--------------COMPUTATION PARAMETERS--------------
Prallelization on threads: 16
Size of Domain: (  6.500,  6.500,  6.500 )
Simulation Time: 45000
```

Начинается сам расчет. На каждом шаге на экран выводится номер текущего шага, количество шагов вообще и время, которое уже было потрачено на расчет:

```
Step    2432 |  45000 |  1m 33s
```

В соответствии с параметром screen_erroroutputsteps, на экране через определенное количество шагов выводятся абсолютные и относительные максимумы ошибок по каждой компоненте поля:

```
 errorHx=  3.227078578376417E-003  1.055536626431111E-002
 errorHy=  3.227078578376417E-003  1.055536626431111E-002
 errorHz=  6.766310458561128E-004  -1.00000000000000
 errorEx=  4.040359450057188E-003  1.237911085243474E-002
 errorEy=  4.040359450057188E-003  1.237911085243474E-002
 errorEz=  5.106679735966142E-003  5.562883537045006E-003
```

Если в программе используется метод лакун, то также выводится список вспомогательных задач, их номера и таймеры до окончания их работы:

```
Slot:  0;    Is Used: 1;    Aux#:  0;    Timer: 1199
Slot:  1;    Is Used: 1;    Aux#:  1;    Timer: 501
```

В конце работы программы, на экран выводятся результаты моделирования: параметры параллелизации, размер расчетной области, общее время моделирования, максимальные относительные ошибки по каждой компоненте поля:

```
 ------ Simulation finished
Prallelization on threads: 16
```

```
Size of Domain: (  6.500,  6.500,  6.500 )
Simulation Time:  20.000
 Maximum errors:
errormaxHX:  0.131319077293120E-01
errormaxHY:  0.131319077293082E-01
errormaxHZ:  0.000000000000000E+00
errormaxEX:  0.155311196592781E-01
errormaxEY:  0.155311196592742E-01
errormaxEZ:  0.164394485291788E-01
```

Кроме этого, выводится общее время, потраченное на расчет в целом и на расчет каждого отдельного шага:

```
Time elapsed:  0.3339508E+02
Timer 0 - MainSource:  0.8082169E+01
Timer 1 - Main E int:  0.1624748E+01
Timer 2 - Main H int:  0.1547163E+01
Timer 3 - Main E bnd:  0.1133170E+02
Timer 4 - Main H bnd:  0.1085720E+01
Timer 5 - Poisson   :  0.0000000E+00
Timer 6 - Manage Aux:  0.0000000E+00
Timer 7 - Output    :  0.3679204E-01
Timer 8 - Analytical:  0.1113381E+01
Timer 9 - GetError  :  0.4762444E+00
Timer10 - AuxSources:  0.0000000E+00
Timer11 - Aux E int :  0.0000000E+00
Timer12 - Aux H int :  0.0000000E+00
Timer13 - Aux E bnd :  0.0000000E+00
Timer14 - Aux H bnd :  0.0000000E+00
Timer15 - PML E int :  0.4782533E+01
Timer16 - PML H int :  0.4471061E+01
Timer17 - BufClear  :  0.3932891E+01
Timer18 - SaveFields:  0.3471878E+01
Timer19 - AuxSaveFie:  0.0000000E+00
```

5.3 Вывод в файлы

Максимальные относительные ошибки и их положение в расчетной области выводятся в файлы:

```
Ex_error.txt
Ey_error.txt
Ez_error.txt
```

`Hx_error.txt`
`Hy_error.txt`
`Hz_error.txt`

Общее время, которое занял расчет в целом и по каждому шагу выводятся в файл `Timing.txt`

Значения полей и токов в нескольких первых вспомогательных задачах выводятся в файлы, которые хранятся в папке `output`.