Моделирование рассеяния волн на объемных объектах, приложение в физически обоснованной нейронной сети.

Кривцов И.А., Смирнова М.А., Знаменский Н.Е.

Московский физико-технический институт (национальный исследовательский университет)

Содержание

1 Теоретические сведения

1.1 Типы Дифракции

Пусть излучение с длиной волны λ падает на непрозрачный экран с отверстием (щелью) размером b, расстоянием от отверстия до экрана h. Поскольку граница n-й зоны Френеля (Шустера) даётся формулой $r_n = \sqrt{n\lambda h}$, то по отношению к точке наблюдения P в отверстии укладывается число зон

$$n \sim \frac{b^2}{\lambda h} \tag{1}$$

Соответственно говорят, что при

$$n \ll 1 \qquad \text{имеет место дифракция Фраунгофера,} \tag{2}$$

$$n \sim 1 \qquad \text{реализуется дифракция Френеля,} \tag{3}$$

$$n \gg 1 \qquad \text{— геометрическая оптика.} \tag{4}$$

Последний случай связан с тем, что в этом пределе открыта значительная часть волнового фронта, и роль краёв препятствия незначительна — точка наблюдения находится близко к экрану.

Величина $f = \frac{b^2}{\lambda h}$ называется *параметром (или числом) Френеля*. Используют также *волновой параметр*, определяемый формулой

$$\rho = \frac{1}{\sqrt{f}} = \sqrt{\frac{\lambda h}{b}} \tag{5}$$

Условие дифракции Фраунгофера можно записать в виде

$$\frac{b}{h} \ll \frac{\lambda}{b} \quad \Rightarrow \quad \frac{b^2}{h\lambda} \ll 1.$$

Это значит, что из точки наблюдения отверстие видно под малым углом: $b/h \ll 1$, а радиус когерентности значительно превышает размер отверстия: $r_{\text{ког}} = \lambda h/b \gg b$.

Согласно сказанному, условие $f \sim 1$ означает, что в отверстие попадает порядка одной зоны Френеля. Это и есть условие дифракции Френеля.

Наш алгоритм моделирует дифракцию именно Френеля на различных структурах.

1.2 Дифракция Френеля на щели

Пусть на экран Э падает плоская волна с амплитудой A_0. Считая экран $\mathbb{E}$ неограниченным вдоль оси y, обозначим координаты точки P в плоскости наблюдения $Э_1$ как $x_P = x$, $y_P = 0$. Выделим на открытой части волнового фронта участок площадью $d\sigma = dx_1 dy_1$. Тогда расстояние от него до точки P равно

$$r = \sqrt{h^2 + (x - x_1)^2 + y_1^2} \tag{6}$$

Запишем интеграл Френеля:

$$A(P) \equiv A(x) = \int_0^\infty dx_1 \int_{-\infty}^\infty dy_1 \, K(\alpha) \frac{1}{r} e^{ikr} \tag{7}$$

Поскольку размер зон убывает с ростом номера, основной вклад в поле вносит область фронта, видимая под малыми углами к нормали: $K(\alpha) \approx K(0) = \frac{k}{2\pi i}$. Будем также предполагать, что рассматривается область вблизи границы экрана: $|x| \ll h$. В этих предположениях можно произвести разложение в (8) по степеням координат x, x_1, y_1:

$$r \approx h + \frac{(x-x_1)^2 + y_1^2}{2h} \tag{8}$$

Соответственно интеграл Френеля принимает вид:

$$\begin{aligned} A(x) &= \frac{k}{2\pi i h} e^{ikh} \int_0^\infty dx_1 \int_{-\infty}^\infty dy_1 \, \exp\left(ik \frac{(x-x_1)^2 + y_1^2}{2h} \right) \\ &= \frac{k}{2\pi i h} e^{ikh} \left[\int_0^\infty \exp\left(ik \frac{(x-x_1)^2}{2h} \right) dx_1 \right] \cdot \left[\int_{-\infty}^\infty \exp\left(ik \frac{y_1^2}{2h} \right) dy_1 \right] \end{aligned} \tag{9}$$

Сюда входит, во-первых, интеграл

$$\int_{-\infty}^\infty \exp\bigl(i\alpha y_1^2\bigr) dy_1 = \sqrt{\frac{\pi}{\alpha}}, \quad \alpha = \frac{k}{2h}, \tag{10}$$

аналогичный интегралу Пуассона:

$$\int_{-\infty}^\infty e^{-\alpha x^2} dx = \sqrt{\frac{\pi}{\alpha}}.$$

Второй интеграл в (9) имеет вид

$$\int_0^\infty \exp\left(ik \frac{(x-x_1)^2}{2h} \right) dx_1 = \frac{1}{\sqrt{\alpha}} \int_{-\alpha x}^\infty \exp\bigl(i\tau^2\bigr) d\tau \tag{11}$$

Запишем промежуточный результат:

$$A(x) = A_0 e^{ikh} \frac{1}{\sqrt{\pi i}} \int_{-\alpha x}^\infty \exp\bigl(i\tau^2\bigr) d\tau \tag{12}$$

Приведём данный результат к традиционной форме. Введём **интегралы Френеля**:

$$S(u) = \sqrt{\frac{2}{\pi}} \int_0^u \sin\bigl(\tau^2\bigr) d\tau, \quad C(u) = \sqrt{\frac{2}{\pi}} \int_0^u \cos\bigl(\tau^2\bigr) d\tau \tag{13}$$

Эти интегралы обладают следующими свойствами:

$$S(-u) = -S(u), \quad C(-u) = -C(u); \quad S(+\infty) = \frac{1}{2}, \quad C(+\infty) = \frac{1}{2} \tag{14}$$

Поскольку

$$\int_u^\infty e^{i\tau^2} d\tau = \int_0^\infty e^{i\tau^2} d\tau - \int_0^u e^{i\tau^2} d\tau = \frac{1}{2}\sqrt{\pi i} - \sqrt{\frac{\pi}{2}} \left[C(u) + iS(u) \right] \tag{15}$$

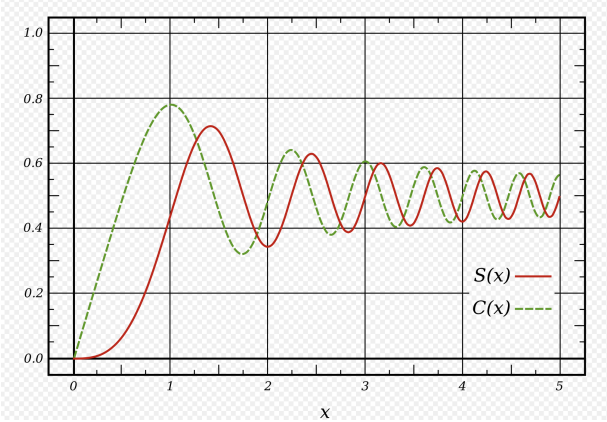

Рис. 1: Интегралы Френеля

учтено, что $\sqrt{i} = \frac{1+i}{\sqrt{2}}$, то выражение (12) можно переписать в виде

$$A(x) = A_0 e^{ikh} \frac{1}{\sqrt{2}} \left[\left(\frac{1}{2} + C\left(x\sqrt{\frac{k}{2h}} \right) \right) + i \left(\frac{1}{2} + S\left(x\sqrt{\frac{k}{2h}} \right) \right) \right] \tag{16}$$

Для интенсивности $I = |A|^2$ получаем

$$I = \frac{1}{2} I_0 \left\{ \left[\frac{1}{2} + C\left(x\sqrt{k/2h} \right) \right]^2 + \left[\frac{1}{2} + S\left(x\sqrt{k/2h} \right) \right]^2 \right\}$$

Найденная с помощью этой формулы зависимость интенсивности излучения от положения точки P показана на рис. 6.3.4. Видно, что в области геометрической тени ($x < 0$) интенсивность монотонно возрастает по мере приближения к границе тени, достигая на самой границе значения $I = I_0/4$. В области $x > 0$ интенсивность совершает колебания с убывающей амплитудой, причём

$$I \to I_0 \quad \text{при} \quad x \to +\infty$$

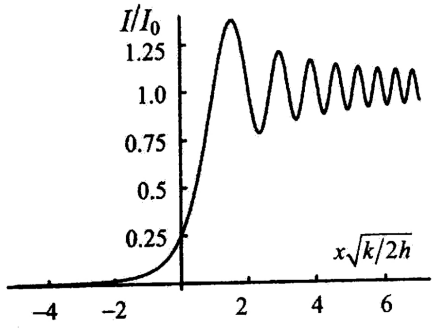

Рис. 2: График зависимости интенсивности излучения от координаты x точки наблюдения относительно края

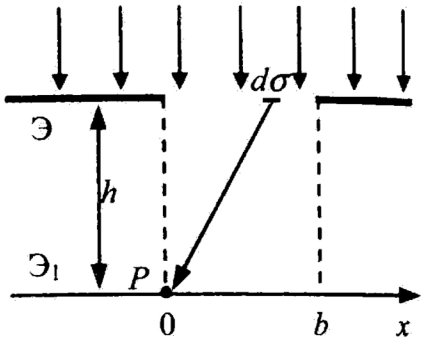

Рис. 3: Схематический рисунок щели и экрана

1.3 Рассеяние Ми

Рассеяние Ми - задача рассеяния плоской электромагнитной волны на сферической частице с заданными радиусом R и диэлектрической проницаемостью ϵ.

Решение задачи находится с помощью разложения электромагнитного поля на векторные сферические гармоники (фундаментальный набор решений векторного уравнения Гельмгольца в сферических координатах).

Основные характеристики рассеяния Ми:

- Применимо для частиц любого размера (от нано- до микро- масштаба)
- Учитывает как поглощение, так и рассеяние излучения

1.3.1 Поля излучения

Полное поле при рассеянии на сфере радиуса a:

$$\mathbf{E}(r,\theta) = \begin{cases} \mathbf{E}_i(r,\theta), & \text{если } r < a \\ \mathbf{E}_f(r,\theta) + \mathbf{E}_s(r,\theta), & \text{иначе} \end{cases}$$

где:

- $\mathbf{E}_f$ – падающее поле (создаётся источником и фокусирующей оптикой)
- $\mathbf{E}_s$ – рассеянное поле
- $\mathbf{E}_i$ – поле внутри сферы
- (r,θ) – сферические координаты

1.3.2 Падающее поле (фокусированное)

Представляется как суперпозиция плоских волн:

$$\mathbf{E}_f(\mathbf{r}) = \mathbf{E}_0 \sum_{j=1}^{J} e^{i\mathbf{k}_j^\top \mathbf{r}}$$

Используя разложение Дебая:

$$e^{i\mathbf{k}^\top \mathbf{r}} = \sum_{l=0}^{\infty} (2l+1) i^l j_l(kr) P_l(\cos\theta)$$

где:

- l - порядок падающей волны
- $j_l(\cdot)$ – сферическая функция Бесселя первого рода порядка l
- $P_l(\cdot)$ – полином Лежандра порядка l
- $k = ||\mathbf{k}|| = \frac{2\pi}{\lambda}$ – волновое число, равное для всех падающих волн, так как считаем падающее излучение когерентным.

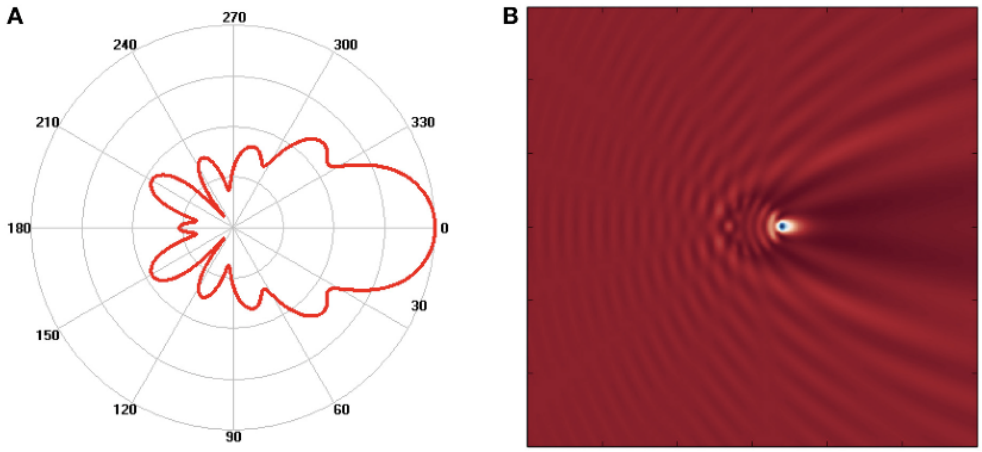

Рис. 4: Пример рассеяния Ми: график зависимости интенсивности излучения I от угла θ, картина рассеяния при R = 1 мкм, λ = 1 мкм

1.3.3 Рассеянное и внутреннее поля

Рассеянное поле для одной плоской волны:

$$\mathbf{E}_s(p) = \sum_{l=0}^{\infty} B_l(\lambda, n, a) h_l^{(1)}(kr) P_l(\cos\theta)$$

где $h_l^{(1)}(x)$ – сферическая функция Ханкеля первого рода. Коэффициенты связи:

$$B_l = (2l+1) i^l \frac{j_l(kn) j_l'(kna) n - j_l(kna) j_l'(k)}{j_l(kna) h_l^{(1)'}(ka) - h_l^{(1)}(ka) j_l(kna) n}$$

Поле внутри сферы:

$$\mathbf{E}_i(p) = \sum_{l=0}^{\infty} A_l(\lambda, n, a) j_l(knr) P_l(\cos\theta)$$

где коэффициенты:

$$A_l = (2l+1) i^l \frac{j_l(kn) h_l^{(1)'}(ka) - j_l'(kn) h_l^{(1)}(ka)}{j_l(kna) h_l^{(1)'}(ka) - h_l^{(1)}(ka) j_l(kna) n}$$

В обоих формулах выше применены следующие обозначения:

- $j_l'(\cdot)$ - первая производная сферической функции Бесселя первого рода
- $h_l^{(1)}(\cdot)$ - сферическая функция Ханкеля первого рода порядка l.
- $h_l^{(1)'}(\cdot)$ - первая производная сферической функции Ханкеля.

Полученные уравнения аналитического решения не имеют, поэтому применяются численные методы.

2 Моделирование с помощью библиотеки `PyMeep`

2.1 Описание работы библиотеки `PyMeep`

`PyMeep` — это высокоуровневая обёртка на языке Python для вычислительного ядра `Meep`, реализованного на C++. Ядро `Meep` основано на методе конечных разностей во временной области (FDTD, *Finite-Difference Time-Domain*) для решения уравнений Максвелла и оптимизировано для высокопроизводительного моделирования распространения электромагнитных волн. `PyMeep` предоставляет возможность задавать геометрию, материалы, источники, параметры вычислений и собирать результаты прямо в привычном синтаксисе Python, при этом фактически все тяжёлые вычисления выполняются в скомпилированной части на C++.

2.2 Архитектура и связь Python $\longleftrightarrow$ C++

- Ядро `Meep` написано на C++ и реализует:
  - Метод FDTD для численного решения уравнений Максвелла.
  - Поддержку различных материалов (диэлектрических, металлов, анизотропных сред).
  - Граничные условия (PML, симметрии).
- Основные классы и структуры:
  - `meep::simulation` — главный класс, отвечающий за организацию FDTD-цикла, хранение массивов полей $\mathbf{E}$ и $\mathbf{H}$, коэффициентов обновления и слоёв PML.
  - `meep::geometric_object` — описание геометрических объектов (блоки, сферы, призмы и т.п.) и их материалов.
  - `meep::medium` — структура, задающая свойства материала в каждой точке пространства (диэлектрическая проницаемость ε, магнитная проницаемость μ, дисперсия и т.д.).
  - `meep::source` — описание источника поля (тип: гауссов импульс, синусоида, временная зависимость, компоненты поля, координаты).
  - `meep::pml_region` — информация о слоях PML и профилях поглощения.

2.3 Создание и запуск симуляции в PyMeep: что происходит

2.3.1 Инициализация среды и объектов

1. Пользователь импортирует библиотеку:

```
import meep as mp
```

   При этом происходит:

   - Загрузка `_meep.so` (или `_meep.pyd`) и динамических библиотек `libmeep.so`, `libfftw3.so`, `libhdf5.so` и т.д.
   - Вызов инициализирующих функций C++ (регистрация типов, установка глобальных флагов, инициализация логирования).

2. Пользователь задаёт параметры симуляции в Python:

```
resolution = 50
cell = mp.Vector3(10, 10, 0)
```

   При этом:

   - `resolution = 50` — это просто Python-целое, которое будет передано далее при создании объекта `Simulation`.

3. Определение геометрии:

```
geometry = [mp.Block(size=mp.Vector3(1, 1, mp.inf),
                     center=mp.Vector3(0, 0, 0),
                     material=mp.Medium(epsilon=12))]
```

   Поймём шаги:

   - `mp.Block(...)`: SWIG-обёртка вызывает C++-конструктор `meep::geometric_object` с параметрами:
     - `size` — физический размер объекта, переводится в число ячеек с учётом `resolution`.
     - `center` — физические координаты центра, тоже переводятся в безразмерные (число ячеек).
     - `material=mp.Medium(epsilon=12)`: создаётся C++-объект `meep::medium`, хранящий диэлектрическую проницаемость $\varepsilon = 12$.
   - Внутри C++-кода создаётся структура, описывающая блок размерами $(\Delta x, \Delta y, \Delta z)$ (в ячейках) и координатами центра. Потом эта структура помещается в список геометрических объектов симуляции.

4. Определение источника:

```
source = [mp.Source(mp.ContinuousSource(frequency=1.0),
                    component=mp.Ez,
                    center=mp.Vector3(-4, 0, 0))]
```

   В подробностях:

- `mp.ContinuousSource(frequency=1.0)`: SWIG создаёт экземпляр C++-класса `meep::continuous_source` с частотой 1.0.
- `component=mp.Ez` — задаёт, что будем возбуждать компоненту E_z. SWIG передаёт в C++ соответствующий enum или константу.
- `center=mp.Vector3(-4,0,0)` — позиция источника (координата в ячейках после пересчёта).
- Итогом получается C++-объект `meep::source`, содержащий:
  - Тип временной зависимости (Continuous).
  - Частоту $\omega = 2\pi f$.
  - Локацию в сетке (индексы ячеек: $i = -4 \cdot \text{resolution}$, $j = 0$, $k = 0$).
  - Компоненту поля, которую нужно возбуждать.

5. Создание объекта симуляции:

```
sim = mp.Simulation(cell_size=cell,
                    geometry=geometry,
                    sources=source,
                    resolution=resolution)
```

Внутри:

- Python-объект `mp.Simulation` вызывает функцию `_meep.new_simulation(...)`:
  - Передаются параметры:
    * `cell_size` (C++-структура `meep::vector3d`).
    * Список указателей на объекты `meep::geometric_object`.
    * Список указателей на объекты `meep::source`.
    * `resolution` (число точек на единицу длины).
    * По умолчанию: граничные условия (PML) обычно задаются отдельно через аргумент `boundary_layers`; если не задано, используется отсутствие PML или стандартный набор.
    * Время моделирования, неявно отсутствующее, задаётся позже при `run()`.
  - В C++ создаётся новый объект `meep::simulation`, выполняется конструктор:
    * На основе `cell_size` и `resolution` рассчитываются размеры сетки:

$$N_x = \text{ceil}(\text{cell\_size}_x \times \text{resolution}),$$
$$N_y = \text{ceil}(\text{cell\_size}_y \times \text{resolution}),$$
$$N_z = \text{ceil}(\text{cell\_size}_z \times \text{resolution}).$$

    * Выделяются массивы для полей $\mathbf{E}$ и $\mathbf{H}$ (массово: размер $N_x \times N_y \times N_z$ для каждого компонента поля).
    * Инициализируются массивы коэффициентов обновления:

$$C_e(i,j,k) = \frac{\Delta t}{\varepsilon(i,j,k)}, \quad C_h(i,j,k) = \frac{\Delta t}{\mu(i,j,k)},$$

где Δt определяется условием Куранта:

$$\Delta t \leq \frac{1}{c\sqrt{\frac{1}{(\Delta x)^2} + \frac{1}{(\Delta y)^2} + \frac{1}{(\Delta z)^2}}}.$$

* Подготавливаются слои PML (если заданы):
  - Рассчитывая профили поглощения $\sigma_x(x)$, $\sigma_y(y)$, $\sigma_z(z)$ в областях PML толщиной d_{PML}, создаются маски для каждого направления.
  - Сохраняются коэффициенты σ_e и σ_h, которые потом добавляются в уравнения обновления.
* Создаются структуры для каждой геометрии:
  - Для каждого `geometric_object` строится маска $\varepsilon(i, j, k)$: каждую ячейку сетки проверяют, попадает ли она внутрь этого объекта, если да — устанавливают $\varepsilon = 12$, иначе по умолчанию $\varepsilon = 1$ (вакуум).
  - Если материал дисперсионный, инициализируются дополнительные переменные состояния (например, поляризационная плотность для Drude или Lorentz-моделей).
* Источники:
  - Для каждого `meep::source` выделяется буфер значений временной функции (например, $\sin(2\pi f t)$ или гауссова импульса).
  - Фиксируются индексы ячеек, в которые нужно добавлять нулевые члены при обновлении полей (например, $E_z(i_s, j_s, k_s)$).

- В результате создаётся объект `meep::simulation*`, привязанный к Python-объекту `sim`, инициализированный всеми параметрами.

2.3.2 Главный цикл FDTD

После инициализации при выполнении команды:

```
sim.run(until=200)
```

происходит следующее:

1. Python-обёртка вызывает C++-метод `meep::simulation::run(double until)`:

   - Аргумент `until=200` интерпретируется как «остановиться, когда $t \geq 200$ (в безразмерных единицах времени)». Если частота единичная, то это соответствует 200 периодам волны.
   - Устанавливается внутренний счётчик времени $t = 0$, шаг $n = 0$.
   - Вычисляется Δt по условию Куранта (если не было заранее рассчитано).

2. Начинается основной цикл (while t $<$ until):

   (а) Обновление магнитного поля $\mathbf{H}$:

   $$\mathbf{H}^{n+\frac{1}{2}}(i,j,k) = \mathbf{H}^{n-\frac{1}{2}}(i,j,k) - \frac{\Delta t}{\mu(i,j,k)} \left(\nabla \times \mathbf{E}^{n}\right)(i,j,k)$$

   с учётом PML: $\mathbf{H} = \mathbf{H} - \sigma_h \mathbf{H}$.

   (б) Применение источников к $\mathbf{H}$ (при необходимости).

   (в) Обновление электрического поля $\mathbf{E}$:

   $$\mathbf{E}^{n+1}(i,j,k) = \mathbf{E}^{n}(i,j,k) + \frac{\Delta t}{\varepsilon(i,j,k)} \left(\nabla \times \mathbf{H}^{n+\frac{1}{2}}(i,j,k) - \mathbf{J}_{\text{источник}}(i,j,k)\right)$$

   с учётом PML: $\mathbf{E} = \mathbf{E} - \sigma_e \mathbf{E}$.

   (г) Обновление $t = t + \Delta t,\ n = n + 1$.

   - **Обновление полей E** и **H** реализовано на основе схемы Yee. Компоненты полей хранятся в ячейках:
     - E_x — в центре грани, перпендикулярной оси X.
     - E_y — в центре грани, перпендикулярной оси Y.
     - E_z — в центре грани, перпендикулярной оси Z.
     - Аналогично для H_x, H_y, H_z — смещены на полшага.
   - Разностные операторы $\nabla\times$ вычисляются через центральные разности. Например при фиксированных i, j, k :

     $$(\nabla\times\mathbf{E})_x = \frac{E_z(i,j+1/2,k) - E_z(i,j-1/2,k)}{\Delta y} - \frac{E_y(i,j,k+1/2) - E_y(i,j,k-1/2)}{\Delta z}.$$

   - **PML (Perfectly Matched Layer)**:
     - В зонах PML коэффициенты σ_e и σ_h растут от нуля к некоторому максимуму внутри слоя толщиной d_{PML}.
     - Уравнения обновления полей получают дополнительные члены вида $-\sigma E$ или $-\sigma H$, моделируя абсорбцию волны на границах, чтобы предотвратить отражение.

3. После достижения $t \geq$ until цикл завершается, и управление возвращается в Python.

2.4 Пример: минимальный код и его разбор

```
import meep as mp

resolution = 50
cell = mp.Vector3(10, 10, 0)

geometry = [mp.Block(size=mp.Vector3(1, 1, mp.inf),
```

```
                        center=mp.Vector3(0, 0, 0),
                        material=mp.Medium(epsilon=12))]

source = [mp.Source(mp.ContinuousSource(frequency=1.0),
                    component=mp.Ez,
                    center=mp.Vector3(-4, 0, 0))]

sim = mp.Simulation(cell_size=cell,
                    geometry=geometry,
                    sources=source,
                    resolution=resolution)

sim.run(until=200)
```

2.4.1 Подробный разбор шагов

1. `import meep as mp`:
   - Python пытается найти `meep` в `site-packages`.
   - Загружается файл `meep/__init__.py`, который импортирует модули, включая `_meep.so`.
   - Внутри `_meep.so` вызывается функция `PyInit__meep()`, инициализируются все объекты C++.
2. `resolution = 50` и `cell = mp.Vector3(10,10,0)`:
   - `resolution` сохраняется в Python-переменную.
   - `mp.Vector3(10,10,0)` создаёт C++-объект `meep::vector3d`, вызывается его конструктор:
     $$\text{vector3d}(x = 10.0,\, y = 10.0,\, z = 0.0).$$
     Python-объект содержит указатель на этот C++-объект.
3. `geometry = [mp.Block(...)]`:
   - `mp.Block(size=..., center=..., material=...)` создаёт C++-объект `meep::geometric_object`:
     - Расчёт числа ячеек по каждой координате:
       $$\Delta x = \frac{1\,\text{unit}}{1} \times \text{resolution} = 50 \text{ ячеек (вдоль X)},$$
       $$\Delta y = 50 \text{ ячеек (вдоль Y)}, \quad \Delta z = \infty \text{ (2D-симуляция)}.$$
     - Материал: `mp.Medium(epsilon=12)` $\rightarrow$ C++:
       $$\varepsilon(i,j,k) = 12 \text{ внутри блока}, \quad = 1 \text{ вне}.$$
   - Ядро создаёт маску $\varepsilon(i,j)$ (двумерный массив) и сохраняет её в `meep::geometry_list`.
4. `source = [mp.Source(...)]`:

- `mp.ContinuousSource(frequency=1.0)`
  C++: `new meep::continuous_source(frequency=1.0)`.
- `component=mp.Ez`, `center=mp.Vector3(-4,0,0)` → C++: `meep::source` с полями:
  type = Continuous,
  $f = 1.0$,
  component = E_z,
  позиция: $i_s = -4 \times 50 = -200, \quad j_s = 0, \quad k_s = 0.$
- Объект `meep::source` добавляется в список источников симуляции.

5. `sim = mp.Simulation(...)`:
   - SWIG вызывает C++-функцию `meep_new_simulation(...)`, передавая:
     - `cell_size` (10×10 в физических единицах).
     - `geometry` (указатель на список `meep::geometric_object`).
     - `sources` (указатель на список `meep::source`).
     - `resolution` = 50.
     - `boundary_layers` = `None` (по умолчанию без PML, если не указано).
   - В конструкторе `meep::simulation`:
     - Рассчитывается дискретизация: $N_x = 10 \cdot 50 = 500$, $N_y = 10 \cdot 50 = 500$, $N_z = 1$ (2D, фактически $k = 0$).
     - Выделяются массивы:
       $E_x[N_x][N_y],\ E_y[N_x][N_y],\ H_z[N_x][N_y],\ (E_z = 0,\ H_x = 0,\ H_y = 0$ — не используются).
     - Вычисляются коэффициенты $\Delta x = 1/50$, $\Delta y = 1/50$, $\Delta z = \infty$ (нет).
     - Вычисляется максимальный Δt по условию Куранта:
       $$\Delta t = \frac{1}{c\sqrt{\left(\frac{1}{\Delta x}\right)^2 + \left(\frac{1}{\Delta y}\right)^2}} = \frac{1}{c\sqrt{(50)^2 + (50)^2}} \approx \frac{1}{c\,70.71}.$$
     - Инициализируются коэффициенты обновления:
       $$C_h(i,j) = \frac{\Delta t}{\mu(i,j)} = \Delta t \quad (\mu = 1), \quad C_e(i,j) = \frac{\Delta t}{\varepsilon(i,j)} = \frac{\Delta t}{1 \text{ или } 12}.$$
     - Слой PML по умолчанию не добавлен (если не передан `boundary_layers`).
     - Готовится список источников: позиция $i_s = -200$, $j_s = 0$.

6. `sim.run(until=200)`:
   - C++-метод `meep::simulation::run(200)` запускает FDTD-цикл:
     - Инициализация $t = 0$, $n = 0$.
     - *Пока* $t < 200$:
       (a) **Обновление H_z (единственная компонента H в 2D-TE)**:
       $$H_z^{n+\frac{1}{2}}(i,j) = H_z^{n-\frac{1}{2}}(i,j) - \frac{\Delta t}{\mu(i,j)}\left(\frac{E_y(i+1,j) - E_y(i,j)}{\Delta x} - \frac{E_x(i,j+1) - E_x(i,j)}{\Delta y}\right).$$

(b) **Добавление источника**:

$$H_z^{n+\frac{1}{2}}(i_s, j_s) \mathrel{+}= \alpha \sin\bigl(2\pi f\, t\bigr),$$

где α — амплитудный коэффициент (по умолчанию $\alpha = 1$), $f = 1.0$.

(c) **Обновление E_x и E_y**:

$$E_x^{n+1}(i,j) = E_x^n(i,j) + \frac{\Delta t}{\varepsilon(i,j)}\Bigl(\frac{H_z^{n+\frac{1}{2}}(i,j) - H_z^{n+\frac{1}{2}}(i,j-1)}{\Delta y}\Bigr),$$

$$E_y^{n+1}(i,j) = E_y^n(i,j) - \frac{\Delta t}{\varepsilon(i,j)}\Bigl(\frac{H_z^{n+\frac{1}{2}}(i,j) - H_z^{n+\frac{1}{2}}(i-1,j)}{\Delta x}\Bigr).$$

(d) Если бы был PML, то к E_x и E_y добавляются члены $-\sigma_e E$.

(e) $n \leftarrow n+1$, $t \leftarrow t + \Delta t$.

– По окончании цикла (когда $t \geq 200$) завершаются все буферы, и метод возвращает управление Python.

2.5 Дополнительные утилиты и взаимодействие с пользователем

2.5.1 Экспорт данных

- `sim.dump('-h5', 'output.h5')` — сразу вызывает C++-функцию, которая:
  - Открывает HDF5-файл `output.h5`.
  - Записывает текущие поля **E**, **H**, карту ε, профили PML в датасеты HDF5 (например, `/Ez`, `/Hx`, `/materials/epsilon` и т.д.).
  - Закрывает файл.
- Пользователь может далее открыть `output.h5` через `h5py` или специализированные программы (ParaView, VisIt) для визуализации.

2.6 Готовые материалы и функции

- Модуль `meep.materials` включает предварительно определённые материалы:
  - `mp.Medium(epsilon=2.25)` — диэлектрик с $\varepsilon = 2.25$ (стёкло).
  - `mp.Medium(index=3.5)` — эквивалентно $\varepsilon = n^2 = 12.25$ (полупроводник).
- При передаче этих объектов в C++ конструктор `meep::medium` создаёт структуры для вычисления дисперсионных поправок при обновлении полей:

$$\mathbf{P}(t) + \frac{\gamma}{\omega_p^2}\frac{d\mathbf{P}}{dt} + \frac{1}{\omega_p^2}\frac{d^2\mathbf{P}}{dt^2} = \varepsilon_0(\varepsilon_\infty - 1)\,\mathbf{E}(t),$$

и т.д.

3 Результаты моделирования

Для моделирования был выбран пакет `PyMeep`, так как он сочетает гибкость Python для постановки задачи и анализа результатов с производительностью и параллельными возможностями C++-ядра, что делает его одним из мощных инструментов для численного моделирования электромагнитных задач методом FDTD. Давайте рассмотрим моделирование на примерах в коде.

4 Физически обоснованная нейросеть

Выше мы представили метод математического моделирования, позволяющий определять значения поля в пространстве. Рассмотрим задачу оптической дифракционной томографии (ODT) — это метод визуализации, предназначенный для восстановления трёхмерного распределения показателя преломления образца (например, биологической клетки) на основе серии двумерных изображений, полученных при различных углах освещения.

Классический метод восстановления 3D-распределения показателя преломления из множества проекций был предложен Вольфом (Wolf) в 1969 году. В этом подходе 3D-область Фурье-представления показателя заполняется 2D-Фурье-преобразованиями измеренных рассеянных полей (двумерное преобразование Фурье рассеянного поля, измеренного в плоскости, перпендикулярной направлению распространения падающей волны, пропорционально значениям трехмерного преобразования Фурье функции $F[n(r)]$ вдоль полукруглой дуги в пространстве Фурье.) Однако из-за ограниченного числа проекций, конечной числовой апертуры оптической системы и использования приближения одиночного рассеяния, в реконструкции возникают искажения, вытягивания и занижения значений показателя преломления из-за отсутствующих пространственных частот.

За последние годы было предложено множество итерационных методов. Основная идея таких подходов заключается в использовании прямой модели распространения, которая предсказывает проекции поля, соответствующие текущей оценке показателя преломления. Однако реализация такой итеративной схемы требует аналитической прямой модели, способной обеспечивать дифференцируемость, то есть возможность обратного распространения ошибки (backpropagation). Поэтому было принято решение попробовать реализовать физически обоснованную нейросеть для применения ее в восстановлении изображений.

4.1 Архитектура физически обоснованной нейросети

Мы будем строить архитектуру, которая в качестве функции потерь использует физические законы, выраженные через дифференциальные уравнения.

В отличие от обычного обучения с учителем, где требуется множество размеченных данных (вход - выход), модель будет минимизировать остаток от соответствующего дифференциального уравнения. Основная идея нашей работы, показанная на рис. 1. Нейронная сеть, которая принимает на вход распределение показателя преломления $n(r)$ и предсказывает рассеянное поле U_s. Ее структура основана на архитектуре U-Net.

Модель принимает на вход дискретное трехмерное распределение показателя преломления размера $N_x \times N_y \times N_z \times 1$ и выдаёт на выходе массив размера $N_x \times N_y \times N_z \times 2$, где два «канала» соответствуют действительной и мнимой частям комплексного поля. Среди различных архитектур выбор падал именно на U-Net, поскольку она эффективно кодирует признаки распределения $n(\mathbf{r})$ посредством последовательных 3D-сверток, а затем декодирует его обратно в пространстве, восстанавливая комплексное электромагнитное поле в тех же координатах. Таким образом архитектура по слоям выглядит следующим образом:

- **InputLayer**: shape $= (64, 64, 64, 1) \rightarrow$ карта показателя преломления
- **Encoder** (сжатие + сохранение признаков):

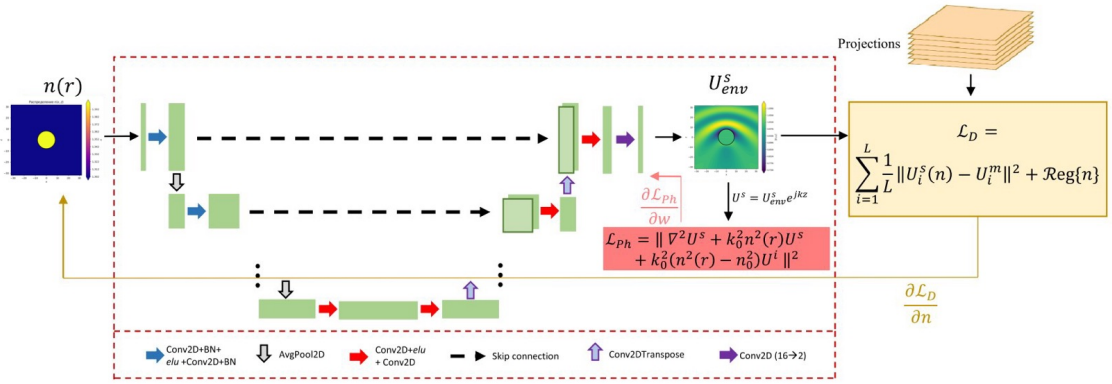

Рис. 5: Архитектура

- Conv3D → BatchNorm → ELU
- Conv3D → BatchNorm → ELU
- AveragePooling3D (2x2x2)

- **Bottleneck**:
  - Conv3D → ELU
  - Conv3D → ELU

- **Decoder**:
  - Conv3DTranspose → Concatenate (skip connection)
  - Conv3D → ELU
  - Conv3D → ELU

4.2 Отличие от классического обучения.

В классическом подходе нам потребовались бы наборы «вход–выход», полученные при помощи численных методов (например, того что мы реализовали в первой части п.3). В том случае, мы бы минимизировали разность между предсказанием сети и «истиной». В нашем подходе не используются заранее готовые вход–выход пары. Вместо этого сеть обучается так, чтобы результирующее поле качественно удовлетворяло уравнению Гельмгольца:

$$\left[\nabla^2 + k_0^2\, n^2(\mathbf{r})\right] U(\mathbf{r}) \;=\; 0, \tag{17}$$

где

$$k_0 = \frac{2\pi}{\lambda}, \quad \lambda = 1.030\ \mu\text{м},$$

$n(\mathbf{r})$ — распределение показателя преломления, а $U(\mathbf{r})$ — комплексное поле. Для ускорения обучения и повышения устойчивости представим рассеянное поле как $U_s^{\text{env}}(\mathbf{r})$, так что

$$U^s(\mathbf{r}) \;=\; U_s^{\text{env}}(\mathbf{r})\, e^{\,i\,k_0\,n_0\,z},$$

где n_0 — показатель преломления фонового слоя. Тогда физически обоснованная функция потерь вычисляется как сумма по всем пикселям $\mathbf{r}$ области:

$$L_{\text{Ph}} \;=\; \sum_{\mathbf{r}} \frac{1}{N} \left\| \left[\nabla^2 + k_0^2\, n^2(\mathbf{r})\right]\, U^s(\mathbf{r}) \;+\; k_0^2\, [\, n^2(\mathbf{r}) - n_0^2\,]\, U^i(\mathbf{r}) \right\|^2, \tag{18}$$

где $U^i(\mathbf{r}) \;=\; e^{\,i\,k_0\,n_0\,z}$ — падающее поле, N — общее число пикселей, а ∇^2 вычисляется методом конечных разностей, реализованной через 3D-свертки.

Чтобы избежать отражений от граничных условий конечного объёма, внутри области расчета используем комплексное «растягивание» координат. Это реализуется через преобразование координат $x \;\to\; x \;+\; i\, f(x)$ при вычислении производных вдоль каждой оси.

Градиенты функции потерь L_{Ph} по весам сети $\mathbf{w}$ автоматически вычисляются с помощью автодифференциации TensorFlow, после чего обновление весов происходит методом Adam: $\mathbf{w} \;\leftarrow\; \mathbf{w} \;-\; \gamma_{\text{Ph}}\, \frac{\partial L_{\text{Ph}}}{\partial \mathbf{w}}$, где γ_{Ph} — скорость обучения.

После того как модель обучена на классе распределений показателя преломления, её можно будет использовать для обратного распространения ошибок в реконструкции томографии. Разработка обратного метода является целью нашей дальнейшей работы.

5 Вывод

В рамках проделанной работы был проведен анализ материала о физике дифракции и рассеяния Мие на предметах различной формы, рассмотрены основные процессы, происходящие при взаимодействии волн с объектами. На основе данного анализа разработан компьютерный метод численного моделирования рассеяния, а также разработана физически обоснованная нейронная сеть на основе U-Net. Они позволяют создавать распределения полей после взаимодействия с объектом. Полученные результаты представлены в коде, а также в Приложении 1.

6 Приложение 1. Результаты работы нейросети