МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«БАЛТИЙСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ им. И. КАНТА»
ИНСТИТУТ ФИЗИКО-МАТЕМАТИЧЕСКИХ НАУК И ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

Выпускная квалификационная работа
Тема: «Численное моделирование эффекта "blow-up"в обратной задаче для волнового уравнения»
Направление подготовки: 01.03.02: «Прикладная математика и информатика»

Выполнил:
Студент 4 курса
________________ Бока И. В.
Руководитель:
________________ Пестов Л. Н.

Калининград
2020

Содержание

Введение

В работе рассмотрены численное решение задачи Коши для волнового уравнения с использованием метода Finite-Difference Time-Domain и обратной задачи для волнового уравнения.

Метод Finite-Difference Time-Domain был впервые использован Kane Yee для уравнений Максвелла в его работе [1].

1 Численное решение прямой задачи

1.1 Finite-Difference Time-Domain

Рассмотрим задачу Коши для волнового уравнения с однородным граничным условияем Неймана.

$$\begin{aligned}
\frac{\partial^2 u}{\partial t^2} &= c^2 \left(\frac{\partial^2 u}{\partial x_1^2} + \frac{\partial^2 u}{\partial x_2^2} + \dots + \frac{\partial^2 u}{\partial x_n^2} \right), t \in (0, T) \\
u\left(\mathbf{x}, 0\right) &= u_0\left(\mathbf{x}\right) \\
\frac{\partial u}{\partial t}\left(\mathbf{x}, 0\right) &= 0 \\
\frac{\partial u}{\partial \mathbf{n}}\left(\mathbf{x}, t\right) &= 0, x \in \partial\Omega
\end{aligned}$$

Численное решение задачи можно получить с использованием метода Finite-Difference Time-Domain [2].

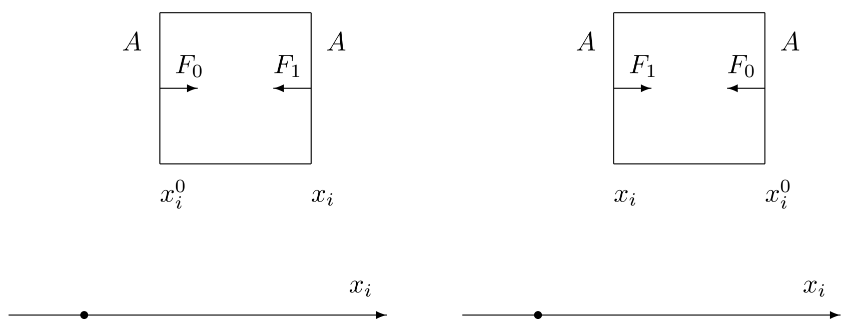
Акустические волны есть изменения давления $u\left(\mathbf{x}, t\right)$.

$c\left(\mathbf{x}\right)$ - скорость звука в точке $\mathbf{x}$ (не зависит от времени t).

Из физических соображений обычно рассматривают случаи $n \leq 3$.
Рассмотрим случай $n = 3$.

$$\begin{aligned}
\frac{\partial^2 u}{\partial t^2} &= c^2 \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} \right), t \in (0, T) \\
u(\mathbf{x}, 0) &= u_0(\mathbf{x}) \\
\frac{\partial u}{\partial t}(\mathbf{x}, 0) &= 0 \\
\frac{\partial u}{\partial \mathbf{n}}(\mathbf{x}, t) &= 0, x \in \partial\Omega
\end{aligned}$$

Пусть $\rho(\mathbf{x}) = \lim \frac{m}{V}$ - плотность в точке $\mathbf{x}$ (не зависит от времени t).
Пусть $\Delta x_i = x_i - x_i^0$.
Рассмотрим куб со стороной $|\Delta x_i|$.
Площадь поперечного сечения $A(\Delta x_i) = (\Delta x_i)^2$.
Объём $V(\Delta x_i) = A|\Delta x_i|$.
Масса $m(\Delta x_i)$.



Проекция сил действующих на куб на ось Ox_i:

$$\begin{aligned}
F_{x_i}(\Delta x_i) &= \operatorname{sign}(\Delta x_i)(F_0 + F_1) \\
&= \operatorname{sign}(\Delta x_i)(u(\mathbf{x})A - u(\mathbf{x} + \Delta x_i \mathbf{e}_i)A) \\
&= \operatorname{sign}(\Delta x_i)(u(\mathbf{x}) - u(\mathbf{x} + \Delta x_i \mathbf{e}_i))A
\end{aligned}$$

где $\mathbf{e}_i$ - базисный вектор на оси Ox_i.
Второй закон Ньютона:

$$
\begin{aligned}
\mathbf{F} &= m\mathbf{a} \\
m\mathbf{a} &= \mathbf{F} \\
ma_{x_i} &= F_{x_i} \\
m\frac{\partial v_{x_i}}{\partial t} &= \operatorname{sign}\left(\Delta x_i\right)\left(u\left(\mathbf{x}\right) - u\left(\mathbf{x} + \Delta x_i \mathbf{e}_i\right)\right) A \\
\frac{m}{V}\frac{\partial v x_i}{\partial t} &= \operatorname{sign}\left(\Delta x_i\right)\left(u\left(\mathbf{x}\right) - u\left(\mathbf{x} + \Delta x_i \mathbf{e}_i\right)\right) \frac{A}{V} \\
\frac{m}{V}\frac{\partial v_{x_i}}{\partial t} &= \operatorname{sign}\left(\Delta x_i\right)\left(u\left(\mathbf{x}\right) - u\left(\mathbf{x} + \Delta x_i \mathbf{e}_i\right)\right) \frac{1}{\left|\Delta x_i\right|} \\
\frac{m}{V}\frac{\partial v_{x_i}}{\partial t} &= \left(u\left(\mathbf{x}\right) - u\left(\mathbf{x} + \Delta x_i \mathbf{e}_i\right)\right) \frac{\operatorname{sign}\left(\Delta x_i\right)}{\left|\Delta x_i\right|} \\
\frac{m}{V}\frac{\partial v_{x_i}}{\partial t} &= \left(u\left(\mathbf{x}\right) - u\left(\mathbf{x} + \Delta x_i \mathbf{e}_i\right)\right) \frac{1}{\operatorname{sign}\left(\Delta x_i\right)\left|\Delta x_i\right|} \\
\frac{m}{V}\frac{\partial v_{x_i}}{\partial t} &= \left(u\left(\mathbf{x}\right) - u\left(\mathbf{x} + \Delta x_i \mathbf{e}_i\right)\right) \frac{1}{\Delta x_i} \\
\frac{m}{V}\frac{\partial v_{x_i}}{\partial t} &= \frac{\left(u\left(\mathbf{x}\right) - u\left(\mathbf{x} + \Delta x_i \mathbf{e}_i\right)\right)}{\Delta x_i} \\
\frac{m}{V}\frac{\partial v_{x_i}}{\partial t} &= -\frac{\left(u\left(\mathbf{x} + \Delta x_i \mathbf{e}_i\right) - u\left(\mathbf{x}\right)\right)}{\Delta x_i}
\end{aligned}
$$

Переходя к пределу получаем $\rho\frac{\partial v_{x_i}}{\partial t} = -\frac{\partial u}{\partial x_i}$ или $\frac{\partial v_{x_i}}{\partial t} = -\frac{1}{\rho}\frac{\partial u}{\partial x_i}$.
Таким образом.

$$
\begin{aligned}
\frac{\partial v_x}{\partial t} &= -\frac{1}{\rho}\frac{\partial u}{\partial x} \\
\frac{\partial v_y}{\partial t} &= -\frac{1}{\rho}\frac{\partial u}{\partial y} \\
\frac{\partial v_z}{\partial t} &= -\frac{1}{\rho}\frac{\partial u}{\partial z}
\end{aligned}
$$

Из уравнения состояния, при использовании нескольких приближений можно получить $\frac{\partial u}{\partial t} = -\rho c^2\left(\frac{\partial v_x}{\partial x} + \frac{\partial v_y}{\partial y} + \frac{\partial v_z}{\partial z}\right)$.

Из системы уравнений

$$\begin{aligned}
\frac{\partial v_x}{\partial t} &= -\frac{1}{\rho}\frac{\partial u}{\partial x} \\
\frac{\partial v_y}{\partial t} &= -\frac{1}{\rho}\frac{\partial u}{\partial y} \\
\frac{\partial v_z}{\partial t} &= -\frac{1}{\rho}\frac{\partial u}{\partial z} \\
\frac{\partial u}{\partial t} &= -\rho c^2 \left(\frac{\partial v_x}{\partial x} + \frac{\partial v_y}{\partial y} + \frac{\partial v_z}{\partial z}\right)
\end{aligned}$$

можно получить волновое уравнение:

$$\begin{aligned}
\frac{\partial^2 u}{\partial t^2} &= \frac{\partial}{\partial t}\frac{\partial u}{\partial t} \\
&= \frac{\partial}{\partial t}\left(-\rho c^2\left(\frac{\partial v_x}{\partial x} + \frac{\partial v_y}{\partial y} + \frac{\partial v_z}{\partial z}\right)\right) \\
&= -\rho c^2 \frac{\partial}{\partial t}\left(\frac{\partial v_x}{\partial x} + \frac{\partial v_y}{\partial y} + \frac{\partial v_z}{\partial z}\right) \\
&= -\rho c^2 \left(\frac{\partial}{\partial t}\frac{\partial v_x}{\partial x} + \frac{\partial}{\partial t}\frac{\partial v_y}{\partial y} + \frac{\partial}{\partial t}\frac{\partial v_z}{\partial z}\right) \\
&= -\rho c^2 \left(\frac{\partial}{\partial x}\frac{\partial v_x}{\partial t} + \frac{\partial}{\partial y}\frac{\partial v_y}{\partial t} + \frac{\partial}{\partial z}\frac{\partial v_z}{\partial t}\right) \\
&= -\rho c^2 \left(\frac{\partial}{\partial x}\left(-\frac{1}{\rho}\frac{\partial u}{\partial x}\right) + \frac{\partial}{\partial y}\left(-\frac{1}{\rho}\frac{\partial u}{\partial y}\right) + \frac{\partial}{\partial z}\left(-\frac{1}{\rho}\frac{\partial u}{\partial z}\right)\right) \\
&= -\rho c^2 \left(-\frac{1}{\rho}\frac{\partial}{\partial x}\frac{\partial u}{\partial x} - \frac{1}{\rho}\frac{\partial}{\partial y}\frac{\partial u}{\partial y} - \frac{1}{\rho}\frac{\partial}{\partial z}\frac{\partial u}{\partial z}\right) \\
&= \left(-\rho c^2\right)\left(-\frac{1}{\rho}\right)\left(\frac{\partial}{\partial x}\frac{\partial u}{\partial x} + \frac{\partial}{\partial y}\frac{\partial u}{\partial y} + \frac{\partial}{\partial z}\frac{\partial u}{\partial z}\right) \\
&= c^2\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}\right)
\end{aligned}$$

Будем решать задачу численно используя метод Finite-Difference Time-Domain. Для этого необходима дискретизация по временным и пространственным переменным.

Узел, в котором вычисляется давление, должен быть окружён узлами, в которых вычисляются компоненты вектора скорости. Расположение узлов отличается от расположения в методе Finite-Difference Time-Domain для законов Ампера и Фарадея.

Кроме того, необходимо смещение по временной переменой - узлы, в которых вычисляется давление, должны быть смещены на полшага от узлов, в которых вычисляются компоненты вектора скорости.

Будем считать пространственный шаг одинаковым $h_x = h_y = h_z = h$.

Заменим производные конечными разностями.

$$
\begin{aligned}
u^l[i,j,k] &\approx u(ih, jh, kh, l\Delta_t) \\
v_x^{l+\frac{1}{2}}[i,j,k] &\approx v_x\left(\left(i+\frac{1}{2}\right)h, jh, kh, \left(l+\frac{1}{2}\right)\Delta_t\right) \\
v_y^{l+\frac{1}{2}}[i,j,k] &\approx v_y\left(ih, \left(j+\frac{1}{2}\right)h, kh, \left(l+\frac{1}{2}\right)\Delta_t\right) \\
v_z^{l+\frac{1}{2}}[i,j,k] &\approx v_z\left(ih, jh, \left(k+\frac{1}{2}\right)h, \left(l+\frac{1}{2}\right)\Delta_t\right) \\
u^l[i,j,k] &= u^{l-1}[i,j,k] - \frac{\Delta_t \rho c^2}{h} (\\
&\qquad v_x^{l-\frac{1}{2}}[i,j,k] - v_x^{l-\frac{1}{2}}[i-1,j,k] \\
&\qquad + v_y^{l-\frac{1}{2}}[i,j,k] - v_y^{l-\frac{1}{2}}[i,j-1,k] \\
&\qquad + v_z^{l-\frac{1}{2}}[i,j,k] - v_z^{l-\frac{1}{2}}[i,j,k-1]) \\
v_x^{l+\frac{1}{2}}[i,j,k] &= v_x^{l-\frac{1}{2}}[i,j,k] - \frac{\Delta_t}{h\rho}\left(u^l[i+1,j,k] - u^l[i,j,k]\right) \\
v_y^{l+\frac{1}{2}}[i,j,k] &= v_y^{l-\frac{1}{2}}[i,j,k] - \frac{\Delta_t}{h\rho}\left(u^l[i,j+1,k] - u^l[i,j,k]\right) \\
v_z^{l+\frac{1}{2}}[i,j,k] &= v_z^{l-\frac{1}{2}}[i,j,k] - \frac{\Delta_t}{h\rho}\left(u^l[i,j,k+1] - u^l[i,j,k]\right)
\end{aligned}
$$

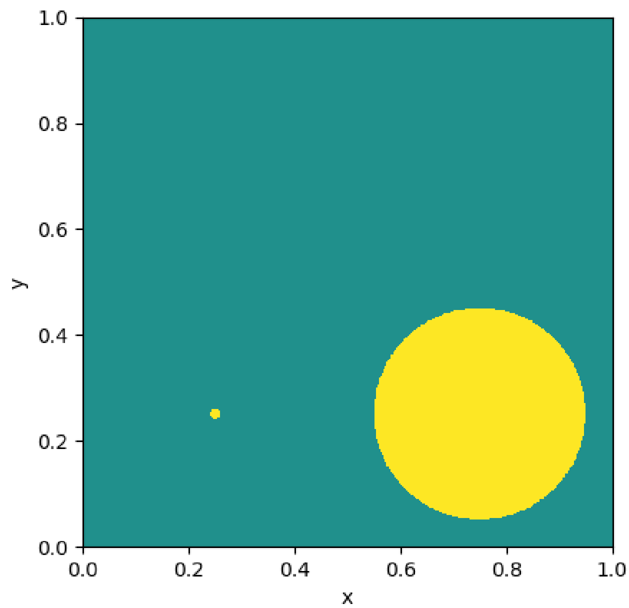
Шаг по времени нужно выбирать так, чтобы число Куранта не превосходило единицу.

$$
\begin{aligned}
C &= \frac{\sup(c)\,\Delta_t}{h} \\
C &\leq 1 \\
\frac{\sup(c)\,\Delta_t}{h} &\leq 1 \\
\Delta_t &\leq \frac{h}{\sup(c)}
\end{aligned}
$$

Для простоты в дальнейшем будем рассматривать случай $n = 2$:

$$
\begin{aligned}
\frac{\partial v_x}{\partial t} &= -\frac{1}{\rho}\frac{\partial u}{\partial x} \\
\frac{\partial v_y}{\partial t} &= -\frac{1}{\rho}\frac{\partial u}{\partial y} \\
\frac{\partial u}{\partial t} &= -\rho c^2\left(\frac{\partial v_x}{\partial x} + \frac{\partial v_y}{\partial y}\right)
\end{aligned}
$$

Рис. 1: Начальные условия



$$u^l\left[i,j\right] \approx u\left(ih, jh, l\Delta_t\right)$$

$$v_x^{l+\frac{1}{2}}\left[i,j\right] \approx v_x\left(\left(i+\frac{1}{2}\right)h, jh, \left(l+\frac{1}{2}\right)\Delta_t\right)$$

$$v_y^{l+\frac{1}{2}}\left[i,j\right] \approx v_y\left(ih, \left(j+\frac{1}{2}\right)h, \left(l+\frac{1}{2}\right)\Delta_t\right)$$

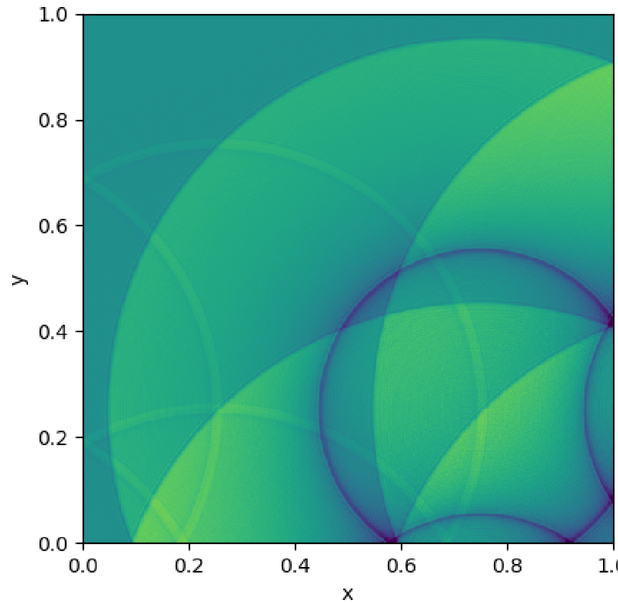
$$u^l\left[i,j\right] = u^{l-1}\left[i,j\right] - \frac{\Delta_t \rho c^2}{h}\Big(v_x^{l-\frac{1}{2}}\left[i,j\right] - v_x^{l-\frac{1}{2}}\left[i-1,j\right] + v_y^{l-\frac{1}{2}}\left[i,j\right] - v_y^{l-\frac{1}{2}}\left[i,j-1\right]\Big)$$

$$v_x^{l+\frac{1}{2}}\left[i,j\right] = v_x^{l-\frac{1}{2}}\left[i,j\right] - \frac{\Delta_t}{h\rho}\left(u^l\left[i+1,j\right] - u^l\left[i,j\right]\right)$$

$$v_y^{l+\frac{1}{2}}\left[i,j\right] = v_y^{l-\frac{1}{2}}\left[i,j\right] - \frac{\Delta_t}{h\rho}\left(u^l\left[i,j+1\right] - u^l\left[i,j\right]\right)$$

Заменим аппроксимацию производной по пространственным переменным на центральную аппроксимацию 12-того порядка. Формулы не приводятся по причине их громоздкости.

Рис. 2: Решение прямой задачи с однородным граничным условием Неймана



1.2 Perfecty Mathed Layer

На практике бывает необходимо решить задачу для неограниченной области. Для этого можно увеличить область Ω настолько, что за время T волны не дойдут до границы $\partial\Omega$. Однако, такой способ неудобен, так как требует больших вычислительных ресурсов. Поэтому на практике используют поглащающие слои на границе, которые позволяют избежать отражения волн.

Техника Perfectly Matched Layer была впервые сформулирована в работе Jean-Pierre Berenger'a [3]. Изначально Perfectly Matched Layer использовался для уравнений Максвелла.

Вместо системы уравнений рассмотренной в предыдущем подразделе рассмотрим следующую систему уравнений:

$$
\begin{aligned}
\frac{\partial v_x}{\partial t} + \sigma_x v_x &= -\frac{1}{\rho}\frac{\partial\left(u_x + u_y\right)}{\partial x} \\
\frac{\partial v_y}{\partial t} + \sigma_y v_y &= -\frac{1}{\rho}\frac{\partial\left(u_x + u_y\right)}{\partial y} \\
\frac{\partial u_x}{\partial t} + \sigma_x^* u_x &= -\rho c^2 \frac{\partial v_x}{\partial x} \\
\frac{\partial u_y}{\partial t} + \sigma_y^* u_y &= -\rho c^2 \frac{\partial v_x}{\partial x}
\end{aligned}
$$

Заметим, что при $\sigma_x = \sigma_x^* = \sigma_y = \sigma_y^* = 0$ получим систему уравнений эквивалентную волновому уравнению.

Во внутренней области положим $\sigma_x = \sigma_x^* = \sigma_y = \sigma_y^* = 0$.

При значениях x близких к граничным $\sigma_x > 0$ и $\sigma_x^* > 0$. При значениях y близких к граничным $\sigma_y > 0$ и $\sigma_y^* > 0$. Причём значение функций σ монотонно возрастает от нуля до максимума по мере приближения к границе.

2 Обратная задача

2.1 Метод граничного управления

Рассмотрим обратную задачу для волнового уравнения.

$$
\begin{aligned}
\frac{\partial^2 u}{\partial t^2} &= c^2\left(\frac{\partial^2 u}{\partial x_1^2} + \frac{\partial^2 u}{\partial x_2^2} + \cdots + \frac{\partial^2 u}{\partial x_n^2}\right), t \in (0, T) \\
u\left(\mathbf{x}, 0\right) &= 0 \\
\frac{\partial u}{\partial t}\left(\mathbf{x}, 0\right) &= 0 \\
u\left(\mathbf{x}, t\right) &= \phi\left(\mathbf{x}, t\right), t\ \in (0, T), x \in \partial\Omega \\
\frac{\partial u}{\partial \mathbf{n}}\left(\mathbf{x}, t\right) &= f\left(\mathbf{x}\right), x \in \partial\Omega
\end{aligned}
$$

Рис. 3: Решение прямой задачи с Perfect Matched Layer на границе

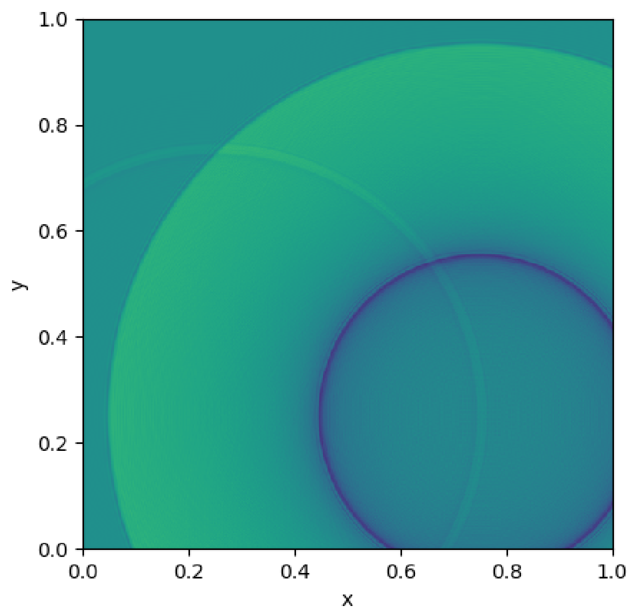
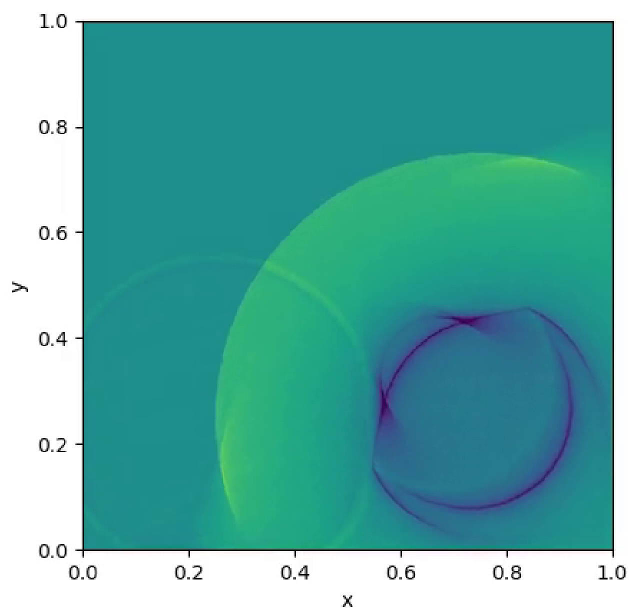


Рис. 4: Решение прямой задачи с Perfect Matched Layer на границе - переменная скорость



Обратная задача граничного управления состоит в отыскании функции $f(\mathbf{x})$, зная функию $\phi(\mathbf{x})$.

2.2 Эффект blow-up

Заключение

Было рассмотрено решение прямой и обратной задачи для волнового уравнения.

Приложения

Программа на языке программирования Python для решения прямой задачи методом Finite-Difference Time-Domain с граничным условием Неймана:

```
from pathlib import Path

import numpy as np
import matplotlib.pyplot as plt

X_LENGTH: int = 1
Y_LENGTH: int = 1
C: int = 1
RHO: int = 1
T: float = 0.5

HALF_MAX_ACCURACY: int = 6
N: int = 512
DELTA_X: float = X_LENGTH / (N - 1)
DELTA_Y: float = Y_LENGTH / (N - 1)
DELTA_T: float = DELTA_X / (4 * C)


def coefficients(half_accuracy: int) -> np.ndarray:
    accuracy: int = 2 * half_accuracy
    return np.delete(
        np.linalg.solve(
            [
                np.concatenate(
                    (
                        np.arange(-half_accuracy + 0.5,
                            0),
                        (0,),
                        np.arange(0.5, half_accuracy),
                    ),
```

```
                )
                **
                i
                for i in range(accuracy + 1)
            ],
            [0, 1] + [0] * (accuracy - 1),
        ),
        half_accuracy,
    )


derivative_approximation_coefficients = [
    coefficients(half_accuracy)
    for half_accuracy in range(1, HALF_MAX_ACCURACY + 1)
]


def derivative_x(function: np.ndarray) -> np.ndarray:
    result: np.ndarray = np.zeros((function.shape[0] - 1,
        function.shape[1]))
    for i in range(function.shape[0] - 1):
        accuracy: int = min(12, 2 * i + 2, (function.
            shape[0] - i - 1) * 2)
        half_accuracy: int = accuracy // 2
        for j in range(accuracy):
            result[i] +=
                derivative_approximation_coefficients[
                 half_accuracy - 1
            ][j] * function[i + j + 1 - half_accuracy]
    return result


def derivative_y(function: np.ndarray) -> np.ndarray:
    result: np.ndarray = np.zeros((function.shape[0],
        function.shape[1] - 1))
    for i in range(function.shape[1] - 1):
        accuracy: int = min(12, 2 * i + 2, (function.
            shape[1] - i - 1) * 2)
        half_accuracy: int = accuracy // 2
        for j in range(accuracy):
            result[:, i] +=
                derivative_approximation_coefficients[
                 half_accuracy - 1
            ][j] * function[:, i + j + 1 - half_accuracy]
    return result
```

```python
def in_circle(x: float, y: float, x0: float, y0: float,
    radius: float) -> bool:
    return (x - x0)**2 + (y - y0)**2 < radius**2


def save_pressure(pressure: np.ndarray, filename: str) ->
     None:
    plt.imshow(pressure, vmin=-1, vmax=1, extent=[0,
        X_LENGTH, 0, Y_LENGTH])
    plt.xlabel('x')
    plt.ylabel('y')
    plt.savefig(Path('..') / 'images' / f'{filename}.png'
        )


def update_neyman(
    pressure: np.ndarray,
    velocity_x: np.ndarray,
    velocity_y: np.ndarray,
    pressure_coefficient: float,
    velocity_coefficient: float,
) -> None:
    velocity_x += velocity_coefficient * derivative_x(
        pressure)
    velocity_y += velocity_coefficient * derivative_y(
        pressure)
    pressure[1:-1] += pressure_coefficient * derivative_x
        (velocity_x)
    pressure[:, 1:-1] += pressure_coefficient *
        derivative_y(velocity_y)

    # Neyman boundary condition
    pressure[0] += 2 * pressure_coefficient * velocity_x
        [0]
    pressure[-1] += 2 * pressure_coefficient * -
        velocity_x[-1]
    pressure[:, 0] += 2 * pressure_coefficient *
        velocity_y[:, 0]
    pressure[:, -1] += 2 * pressure_coefficient * -
        velocity_y[:, -1]


def main() -> None:
    pressure_coefficient: float = -DELTA_T * RHO * C**2 /
         DELTA_X
```

```
    velocity_coefficient: float = -DELTA_T / (DELTA_X *
      RHO)

    pressure: np.ndarray = np.fromfunction(
        lambda i, j: in_circle(
            i * DELTA_X, j * DELTA_Y, 0.75, 0.75, 0.2
        ) + in_circle(
            i * DELTA_X, j * DELTA_Y, 0.75, 0.25, 0.01
        ),
        (N, N)
    ).astype(float)

    save_pressure(pressure, 'initial')

    velocity_x = np.zeros((pressure.shape[0] - 1,
       pressure.shape[1]))
    velocity_y = np.zeros((pressure.shape[0], pressure.
       shape[1] - 1))

    for i in range(round(T / DELTA_T)):
        update_neyman(
            pressure,
            velocity_x,
            velocity_y,
            pressure_coefficient,
            velocity_coefficient,
        )

    save_pressure(pressure, 'forward_neyman_zero_boundary
       ')


if __name__ == '__main__':
    main()
```

Программа на языке программирования Python для решения прямой задачи методом Finite-Difference Time-Domain с использованием техники Perfect Matched Layer:

```
from pathlib import Path

import numpy as np
import matplotlib.pyplot as plt

X_LENGTH: int = 1
Y_LENGTH: int = 1
C: int = 1
```

```
RHO: int = 1
T: float = 0.5
PERFECT_MATCHED_LAYER_WIDTH: float = 0.2

HALF_MAX_ACCURACY: int = 6
N: int = 512
DELTA_X: float = X_LENGTH / (N - 1)
DELTA_Y: float = Y_LENGTH / (N - 1)
DELTA_T: float = DELTA_X / (4 * C)

PERFECT_MATCHED_LAYER_SIZE_X: int = round(
    PERFECT_MATCHED_LAYER_WIDTH / DELTA_X
)
PERFECT_MATCHED_LAYER_SIZE_Y: int = round(
    PERFECT_MATCHED_LAYER_WIDTH / DELTA_Y
)
SIGMA_X_MAX: int = 1000
SIGMA_X_STAR_MAX: int = 1000
SIGMA_Y_MAX: int = 1000
SIGMA_Y_STAR_MAX: int = 1000


def coefficients(half_accuracy: int) -> np.ndarray:
    accuracy: int = 2 * half_accuracy
    return np.delete(
        np.linalg.solve(
            [
                np.concatenate(
                    (
                        np.arange(-half_accuracy + 0.5,
                            0),
                        (0,),
                        np.arange(0.5, half_accuracy),
                    ),
                )
                **
                i
                for i in range(accuracy + 1)
            ],
            [0, 1] + [0] * (accuracy - 1),
        ),
        half_accuracy,
    )


derivative_approximation_coefficients = [
```

```python
    coefficients(half_accuracy)
    for half_accuracy in range(1, HALF_MAX_ACCURACY + 1)
]


def derivative_x(function: np.ndarray) -> np.ndarray:
    result: np.ndarray = np.zeros((function.shape[0] - 1,
        function.shape[1]))
    for i in range(function.shape[0] - 1):
        accuracy: int = min(12, 2 * i + 2, (function.
            shape[0] - i - 1) * 2)
        half_accuracy: int = accuracy // 2
        for j in range(accuracy):
            result[i] +=
                derivative_approximation_coefficients[
                 half_accuracy - 1
            ][j] * function[i + j + 1 - half_accuracy]
    return result


def derivative_y(function: np.ndarray) -> np.ndarray:
    result: np.ndarray = np.zeros((function.shape[0],
        function.shape[1] - 1))
    for i in range(function.shape[1] - 1):
        accuracy: int = min(12, 2 * i + 2, (function.
            shape[1] - i - 1) * 2)
        half_accuracy: int = accuracy // 2
        for j in range(accuracy):
            result[:, i] +=
                derivative_approximation_coefficients[
                 half_accuracy - 1
            ][j] * function[:, i + j + 1 - half_accuracy]
    return result


def in_circle(x: float, y: float, x0: float, y0: float,
    radius: float) -> bool:
    return (x - x0)**2 + (y - y0)**2 < radius**2


def perfect_matched_layer_x(data: np.ndarray, sigma_max:
    int):
    for i in range(PERFECT_MATCHED_LAYER_SIZE_X):
        coefficient = 1 - DELTA_T * sigma_max * (
            1 - i / PERFECT_MATCHED_LAYER_SIZE_X
        )
```

```
        data[i] *= coefficient
        data[-i - 1] *= coefficient


def perfect_matched_layer_y(data: np.ndarray, sigma_max:
    int):
    for i in range(PERFECT_MATCHED_LAYER_SIZE_Y):
        coefficient = 1 - DELTA_T * sigma_max * (
            1 - i / PERFECT_MATCHED_LAYER_SIZE_Y
        )
        data[:, i] *= coefficient
        data[:, -i - 1] *= coefficient


def save_pressure(pressure_x: np.ndarray, pressure_y: np.
    ndarray, filename: str) -> None:
    plt.imshow(
        (
            pressure_x + pressure_y
        )[
            PERFECT_MATCHED_LAYER_SIZE_X:-
                PERFECT_MATCHED_LAYER_SIZE_X,
            PERFECT_MATCHED_LAYER_SIZE_Y:-
                PERFECT_MATCHED_LAYER_SIZE_Y,
        ],
        vmin=-1,
        vmax=1,
        extent=[0, X_LENGTH, 0, Y_LENGTH],
    )
    plt.xlabel('x')
    plt.ylabel('y')
    plt.savefig(Path('..') / 'images' / f'{filename}.png'
        )


def update_perfect_matched_layer(
    pressure_x: np.ndarray,
    pressure_y: np.ndarray,
    velocity_x: np.ndarray,
    velocity_y: np.ndarray,
    pressure_coefficient: float,
    velocity_coefficient: float,
) -> None:
    velocity_x += velocity_coefficient * derivative_x(
        pressure_x + pressure_y)
    velocity_y += velocity_coefficient * derivative_y(
```

```
        pressure_x + pressure_y)

    perfect_matched_layer_x(velocity_x, SIGMA_X_MAX)
    perfect_matched_layer_y(velocity_y, SIGMA_Y_MAX)

    pressure_x[1:-1] += pressure_coefficient *
        derivative_x(velocity_x)
    pressure_y[:, 1:-1] += pressure_coefficient *
        derivative_y(velocity_y)

    perfect_matched_layer_x(pressure_x, SIGMA_X_STAR_MAX)
    perfect_matched_layer_y(pressure_y, SIGMA_Y_STAR_MAX)

    # Neyman boundary condition
    pressure_x[0] += 2 * pressure_coefficient *
        velocity_x[0]
    pressure_x[-1] += 2 * pressure_coefficient * -
        velocity_x[-1]
    pressure_y[:, 0] += 2 * pressure_coefficient *
        velocity_y[:, 0]
    pressure_y[:, -1] += 2 * pressure_coefficient * -
        velocity_y[:, -1]


def main() -> None:
    pressure_coefficient: float = -DELTA_T * RHO * C**2 /
         DELTA_X
    velocity_coefficient: float = -DELTA_T / (DELTA_X *
       RHO)

    pressure_x: np.ndarray = np.fromfunction(
        lambda i, j: in_circle(
            (i - PERFECT_MATCHED_LAYER_SIZE_X) * DELTA_X,
            (j - PERFECT_MATCHED_LAYER_SIZE_Y) * DELTA_Y,
            0.75,
            0.75,
            0.2,
        ) + in_circle(
            (i - PERFECT_MATCHED_LAYER_SIZE_X) * DELTA_X,
            (j - PERFECT_MATCHED_LAYER_SIZE_Y) * DELTA_Y,
            0.75,
            0.25,
            0.01,
        ),
        (
            N + 2 * PERFECT_MATCHED_LAYER_SIZE_X,
```

```
            N + 2 * PERFECT_MATCHED_LAYER_SIZE_Y,
        )
    ).astype(float) / 2
    pressure_y: np.ndarray = pressure_x.copy()

    save_pressure(pressure_x, pressure_y, 'initial')

    velocity_x = np.zeros((pressure_x.shape[0] - 1,
        pressure_x.shape[1]))
    velocity_y = np.zeros((pressure_y.shape[0],
        pressure_y.shape[1] - 1))

    for i in range(round(T / DELTA_T)):
        update_perfect_matched_layer(
            pressure_x,
            pressure_y,
            velocity_x,
            velocity_y,
            pressure_coefficient,
            velocity_coefficient,
        )

    save_pressure(pressure_x, pressure_y, '
        forward_perfect_matched_layer')


if __name__ == '__main__':
    main()
```

Программа на языке программирования Python для решения прямой задачи методом Finite-Difference Time-Domain с использованием техники Perfect Matched Layer - переменная скорость:

```
from pathlib import Path

import numpy as np
from matplotlib.animation import FFMpegWriter
import matplotlib.pyplot as plt

FILE_PATH: Path = Path(
    '..'
) / 'videos' / f'perfect_matched_layer_variable_speed.mp4
    '
FPS: int = 30
DPI: int = 100

X_LENGTH: int = 1
```

```python
Y_LENGTH: int = 1
RHO: int = 1
T: float = 0.5
PERFECT_MATCHED_LAYER_WIDTH: float = 0.2

HALF_MAX_ACCURACY: int = 6
N: int = 512
MAX_SPEED: int = 2
DELTA_X: float = X_LENGTH / (N - 1)
DELTA_Y: float = Y_LENGTH / (N - 1)
DELTA_T: float = DELTA_X / (4 * MAX_SPEED)

PERFECT_MATCHED_LAYER_SIZE_X: int = round(
    PERFECT_MATCHED_LAYER_WIDTH / DELTA_X
)
PERFECT_MATCHED_LAYER_SIZE_Y: int = round(
    PERFECT_MATCHED_LAYER_WIDTH / DELTA_Y
)
SIGMA_X_MAX: int = 1000
SIGMA_X_STAR_MAX: int = 1000
SIGMA_Y_MAX: int = 1000
SIGMA_Y_STAR_MAX: int = 1000

writer = FFMpegWriter(FPS)


def coefficients(half_accuracy: int) -> np.ndarray:
    accuracy: int = 2 * half_accuracy
    return np.delete(
        np.linalg.solve(
            [
                np.concatenate(
                    (
                        np.arange(-half_accuracy + 0.5,
                            0),
                        (0,),
                        np.arange(0.5, half_accuracy),
                    ),
                )
                **
                i
                for i in range(accuracy + 1)
            ],
            [0, 1] + [0] * (accuracy - 1),
        ),
        half_accuracy,
```

```python
    )


derivative_approximation_coefficients = [
    coefficients(half_accuracy)
    for half_accuracy in range(1, HALF_MAX_ACCURACY + 1)
]


def derivative_x(function: np.ndarray) -> np.ndarray:
    result: np.ndarray = np.zeros((function.shape[0] - 1,
        function.shape[1]))
    for i in range(function.shape[0] - 1):
        accuracy: int = min(12, 2 * i + 2, (function.
            shape[0] - i - 1) * 2)
        half_accuracy: int = accuracy // 2
        for j in range(accuracy):
            result[i] +=
                derivative_approximation_coefficients[
                half_accuracy - 1
            ][j] * function[i + j + 1 - half_accuracy]
    return result


def derivative_y(function: np.ndarray) -> np.ndarray:
    result: np.ndarray = np.zeros((function.shape[0],
        function.shape[1] - 1))
    for i in range(function.shape[1] - 1):
        accuracy: int = min(12, 2 * i + 2, (function.
            shape[1] - i - 1) * 2)
        half_accuracy: int = accuracy // 2
        for j in range(accuracy):
            result[:, i] +=
                derivative_approximation_coefficients[
                half_accuracy - 1
            ][j] * function[:, i + j + 1 - half_accuracy]
    return result


def in_circle(x: float, y: float, x0: float, y0: float,
    radius: float) -> bool:
    return (x - x0)**2 + (y - y0)**2 < radius**2


def perfect_matched_layer_x(data: np.ndarray, sigma_max:
    int):
```

```
    for i in range(PERFECT_MATCHED_LAYER_SIZE_X):
        coefficient = 1 - DELTA_T * sigma_max * (
            1 - i / PERFECT_MATCHED_LAYER_SIZE_X
        )
        data[i] *= coefficient
        data[-i - 1] *= coefficient


def perfect_matched_layer_y(data: np.ndarray, sigma_max:
    int):
    for i in range(PERFECT_MATCHED_LAYER_SIZE_Y):
        coefficient = 1 - DELTA_T * sigma_max * (
            1 - i / PERFECT_MATCHED_LAYER_SIZE_Y
        )
        data[:, i] *= coefficient
        data[:, -i - 1] *= coefficient


def save_pressure(pressure_x: np.ndarray, pressure_y: np.
    ndarray) -> None:
    plt.imshow(
        (
            pressure_x + pressure_y
        )[
            PERFECT_MATCHED_LAYER_SIZE_X:-
                PERFECT_MATCHED_LAYER_SIZE_X,
            PERFECT_MATCHED_LAYER_SIZE_Y:-
                PERFECT_MATCHED_LAYER_SIZE_Y,
        ],
        vmin=-1,
        vmax=1,
        extent=[0, X_LENGTH, 0, Y_LENGTH],
    )
    plt.xlabel('x')
    plt.ylabel('y')


def update_perfect_matched_layer(
    pressure_x: np.ndarray,
    pressure_y: np.ndarray,
    velocity_x: np.ndarray,
    velocity_y: np.ndarray,
    speed: np.ndarray,
    pressure_coefficient: float,
    velocity_coefficient: float,
) -> None:
```

```
    velocity_x += velocity_coefficient * derivative_x(
        pressure_x + pressure_y)
    velocity_y += velocity_coefficient * derivative_y(
        pressure_x + pressure_y)

    perfect_matched_layer_x(velocity_x, SIGMA_X_MAX)
    perfect_matched_layer_y(velocity_y, SIGMA_Y_MAX)

    pressure_x[1:-1] += pressure_coefficient * speed
        [1:-1] * derivative_x(
         velocity_x
    )
    pressure_y[:, 1:-1] += pressure_coefficient * speed[
         :, 1:-1
    ] * derivative_y(
         velocity_y
    )

    perfect_matched_layer_x(pressure_x, SIGMA_X_STAR_MAX)
    perfect_matched_layer_y(pressure_y, SIGMA_Y_STAR_MAX)

    # Neyman boundary condition
    pressure_x[0] += 2 * pressure_coefficient *
        velocity_x[0]
    pressure_x[-1] += 2 * pressure_coefficient * -
        velocity_x[-1]
    pressure_y[:, 0] += 2 * pressure_coefficient *
        velocity_y[:, 0]
    pressure_y[:, -1] += 2 * pressure_coefficient * -
        velocity_y[:, -1]


def main() -> None:
    pressure_coefficient: float = -DELTA_T * RHO /
        DELTA_X
    velocity_coefficient: float = -DELTA_T / (DELTA_X *
        RHO)

    speed: np.ndarray = np.fromfunction(
         lambda i, j: 2 - in_circle(
              (i - PERFECT_MATCHED_LAYER_SIZE_X) * DELTA_X,
              (j - PERFECT_MATCHED_LAYER_SIZE_Y) * DELTA_Y,
              0,
              0,
              1,
         )
```

```
    ,
    (
        N + 2 * PERFECT_MATCHED_LAYER_SIZE_X,
        N + 2 * PERFECT_MATCHED_LAYER_SIZE_Y,
    )
).astype(float)

pressure_x: np.ndarray = np.fromfunction(
    lambda i, j: in_circle(
        (i - PERFECT_MATCHED_LAYER_SIZE_X) * DELTA_X,
        (j - PERFECT_MATCHED_LAYER_SIZE_Y) * DELTA_Y,
        0.75,
        0.75,
        0.2,
    ) + in_circle(
        (i - PERFECT_MATCHED_LAYER_SIZE_X) * DELTA_X,
        (j - PERFECT_MATCHED_LAYER_SIZE_Y) * DELTA_Y,
        0.75,
        0.25,
        0.01,
    ),
    (
        N + 2 * PERFECT_MATCHED_LAYER_SIZE_X,
        N + 2 * PERFECT_MATCHED_LAYER_SIZE_Y,
    )
).astype(float) / 2
pressure_y: np.ndarray = pressure_x.copy()

fig = plt.figure()
with writer.saving(fig, FILE_PATH, DPI):
    save_pressure(pressure_x, pressure_y)
    writer.grab_frame()

    velocity_x = np.zeros((pressure_x.shape[0] - 1,
        pressure_x.shape[1]))
    velocity_y = np.zeros((pressure_y.shape[0],
        pressure_y.shape[1] - 1))

    for i in range(round(T / DELTA_T)):
        update_perfect_matched_layer(
            pressure_x,
            pressure_y,
            velocity_x,
            velocity_y,
            speed,
            pressure_coefficient,
```

```
            velocity_coefficient,
        )
        save_pressure(pressure_x, pressure_y)
        if i % 4 == 3:
            writer.grab_frame()


if __name__ == '__main__':
    main()
```

Список литературы

[1] Kane Yee. Numerical solution of initial boundary value problems involving maxwell's equations in isotropic media. *IEEE Transactions on Antennas and Propagation*, 14(3):302–307, May 1966.

[2] John B. Schneider. *Understanding the Finite-Difference Time-Domain Method*. 2010.

[3] Jean-Pierre Berenger. A perfectly matched layer for the absorption of electromagnetic waves. *Journal of Computational Physics*, 114(2):185–200, October 1994.