Разностные схемы для волнового уравнения

С.В. Лемешевский (sergey.lemeshevsky@gmail.com)

Институт математики НАН Беларуси

Mar 30, 2017

Аннотация

Многие физические волновые процессы приводят описываются уравнениями в частных производных гиперболического типа $\frac{\partial^2 u}{\partial t^2} = \nabla \cdot (c^2 \nabla u) + f$, решение которого с помощью метода конечных разностей мы рассмотрим в данной части.

Содержание

1. Разностная схема для одномерного волнового уравнения

Рассмотрим одномерную математическую модель распространения колебаний на струне. Пусть струна в деформированном состоянии распространяется на интервале $[0, l]$ оси x и $u(x, t)$ — перемещение по времени в направлении y точки, изначально лежащей на оси x. Функция перемещения $u(x, t)$ определяется следующей математической моделью:

$$\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2}, \quad x \in (0, l), \quad t \in (0, T], \tag{1}$$

$$u(x, 0) = I(x), \quad x \in [0, l], \tag{2}$$

$$\frac{\partial u(x, 0)}{\partial t} = 0, \quad x \in [0, l], \tag{3}$$

$$u(0, t) = 0, \quad t \in (0, T], \tag{4}$$

$$u(l, t) = 0, \quad t \in (0, T]. \tag{5}$$

Постоянная c и функция $I(x)$ — заданы.

Уравнение (1) известно как *волновое уравнение* (*уравнение колебаний струны*). Так как это уравнение в частных производных содержит вторую производную по времени, необходимо задать два начальных условия. Условие (2) начальную форму струны, а условие (3) означает, что начальная скорость струны равна нулю. Кроме того уравнение (1) дополняется граничными условиями (4) и (5). Эти два условия означают, что струна закреплена на концах, т.е. перемещения равны нулю.

Перейдем к построению конечно-разностной аппроксимации задачи (1)–(5).

1.1. Расчетная сетка

Для построения разностной схемы надо прежде всего ввести сетку в области изменения независимых переменных и задать шаблон, т.е. множество точек сетки, участвующих в аппроксимации дифференциального выражения. Введем равномерную сетку по переменному x с шагом h

$$\omega_h = \{x_i = ih,\ i = 0, 1, \ldots, N,\ hN = l\},$$

и сетку по переменной t с шагом τ

$$\omega_\tau = \{t_n = n\tau,\ n = 0, 1, \ldots, K,\ K\tau = T\}.$$

Точки (x_i, t_n), $i = 0, 1, \ldots, N$, $n = 0, 1, \ldots, K$, образуют узлы пространственно-временной сетки $\omega_{h\tau} = \omega_h \times \omega_\tau$ (см. рис. 1)

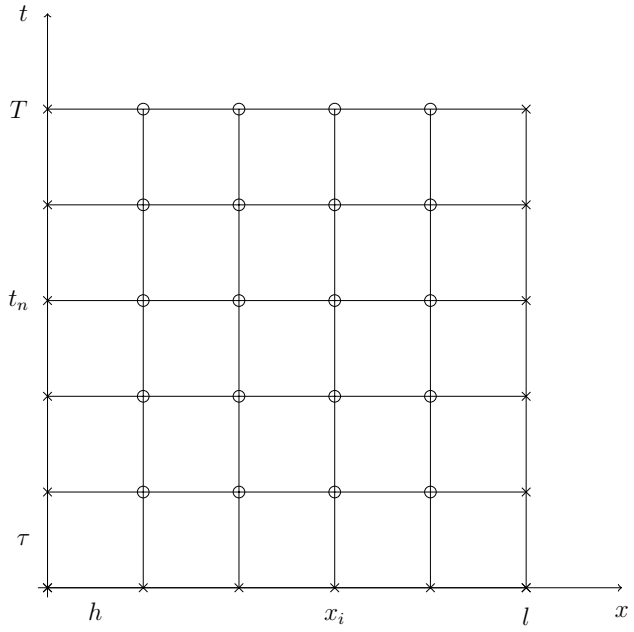

Рис. 1: Пространственно-временная сетка $\omega_{h\tau}$.

Узлы (x_i, t_n), принадлежащие отрезкам $I_0 = \{0 \le x \le l,\ t = 0\}$, $I_l = \{x = 0,\ 0 \le t \le T\}$, $I_r = \{x = l,\ 0 \le t \le T\}$ называются *граничными узлами* сетки $\omega_{h\tau}$, а остальные узлы — *внутренними*. На рис. 1 граничные узлы обозначены крестиками, а внутренние кружочками.

Слоем называется множество всех узлов сетки $\omega_h\tau$, имеющих одну и ту же временную координату. Так, n-м слоем называется множество узлов

$$(x_0, t_n), (x_1, t_n), \ldots, (x_N, t_n).$$

Очевидно, минимальный шаблон, на котором можно аппроксимировать уравнение (1), это пятиточечный шаблон, изображенный на рис. 2. Таким образом, здесь требуется использовать три временных слоя: $n-1$, n, $n+1$. Такие схемы называются *трехслойными*. Их применение предполагает, что при нахождении значений y_i^{n+1} на верхнем слое значения на предыдущих слоях y_i^n, y_i^{n-1}, $i = 0, 1, \ldots, N$ хранятся в памяти.

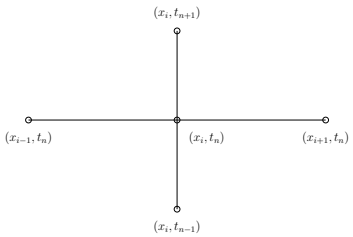

Рис. 2: Минимальный шаблон трехслойной разностной схемы.

1.2. Разностная схема

Простейшей разностной аппроксимацией уравнения (1) и граникных условий (4) и (5) является следующая система уравнений:

$$\frac{y_i^{n+1} - 2y_i^n + y_i^{n-1}}{\tau^2} = \frac{y_{i+1}^n - 2y_i^n + y_{i-1}^n}{h^2}, \tag{6}$$
$$i = 1, 2, \ldots, N-1,\ n = 1, 2, \ldots, K,$$
$$y_0^{n+1} = y_N^{n+1} = 0, \quad n = 0, 1, \ldots, K-1. \tag{7}$$

Разностное уравнение (1) имеет второй порядок погрешности аппроксимации по τ и по h. Решение y_i^{n+1} выражается явным образом через значения на предыдущих слоях:

$$y_i^{n+1} = 2y_i^n - y_i^{n-1} + \gamma^2(y_{i+1}^n - 2y_i^n + y_{i-1}^n), \tag{8}$$
$$i = 1, 2, \ldots, N-1,\ n = 1, 2, \ldots, K-1.$$

Здесь мы ввели параметр

$$\gamma = c\frac{\tau}{h},$$

который называют *числом Куранта*.

Для начала счета по (8) должны быть заданы значения $y_i^0, y_i^1, i = 0, 1, \ldots, N$. Из первого начального условия (2) сразу получаем

$$y_i^0 = I(x_i), \quad i = 0, 1, \ldots, N. \tag{9}$$

1.3. Аппроксимация второго начального условия

Простейшая замена второго начального условия (3) уравнением $(y_i^1 - y_i^0)/\tau = 0$ имеет лишь первый порядок аппроксимации по τ. Поскольку уравнение (6) аппроксимирует уравнение (1) со вторым порядком, желательно, чтобы и разностное начальное условие также имело второй порядок аппроксимации. Построим такую аппроксимацию. Уравнение

$$\frac{y_i^1 - y_i^{-1}}{2\tau} = 0, \tag{10}$$

аппроксимирует уравнение $\frac{\partial u}{\partial t} = 0$ со вторым порядком. Чтобы найти значения y_i^{-1} запишем уравнение (6) при $n = 0$:

$$\frac{y_i^1 - 2y_i^0 + y_i^{-1}}{\tau^2} = y_{\bar{x}x,i}^0,$$

Из (10) имеем $y^{-1} = y^1$. Отсюда получаем

$$y_i^1 = y_i^0 + \frac{\gamma^2}{2}\left(y_{i+1}^0 - 2y_i^0 + y_{i-1}^0\right). \tag{11}$$

Совокупность уравнений (6), (7), (9) и (10) составляет разностную схему, аппроксимирующую исходную задачу (1)–(5).

1.4. Вычислительный алгоритм

Теперь мы можем сформулировать вычислительный алгоритм:

1. Вычисляем y_i^0, используя (9).
2. Вычисляем y_i^1, используя (11) и задаем граничные условия (7) при $n = 0$.
3. Для всех временных слоев $n = 1, 2, \ldots, K - 1$
   (a) находим y_i^{n+1}, используя (8).
   (b) задаем граничные условия (7).

1.5. Эскиз программной реализации

При реализации представленного алгоритма на Python будем использовать массивы `y[i]` для хранения значений y_i^{n+1}, `y_1[i]` для хранения значений y_i^n и `y_2[i]` для хранения y_i^{n-1}. Можно считать, что используется следующее соглашение о названии переменных: `y` используется для вычисляемого пространственного распределения (сеточной функции) на новом временном шаге, `y_1` — решение на временном шаге, отстоящем на один временной слой назад, `y_2` — на два временных слоя назад и т.д.

Алгоритм использует только три временных слоя, таким образом, нам достаточно иметь только три массива для y_i^{n+1}, y_i^n и y_i^{n-1}, $i = 0, 1, \ldots, N$. Хранение всего решения в двумерном массиве размерности $(N+1)\times(K+1)$ возможно в простейшем одномерном случае уравнений в частных производных, но не для двумерных и трехмерных задач. Таким образом, во всех программах для решения уравнений в частных производных мы будем хранить в памяти минимально возможное число временных слоев.

Следующий фрагмент кода реализует вычислительный алгоритм

```
# -*- coding: utf-8 -*-

# --- Start snippet 1 ---
# Заданные сетки как массивы x и t
dx = x[1] - x[0]
dt = t[1] - t[0]
C = c*dt/dx  # Число Куранта
K = len(t) - 1
N = len(x) - 1
C2 = C**2

# Задаем начальное условие
for i in range(N+1) :
    y_1[i] = I(x[i])

# Используем специальную формулу для расчета на первом
# временном шаге с учетом du/dt = 0
for i in range(N):
    y[i] = y_1[i] - 0.5*C2(y_1[i+1] - 2*y_1[i] + y_1[i-1])
y[0] = 0; y[N] = 0 # Применяем граничные условия

# Изменяем переменные перед переходом на следующий
# временной слой
y_2[:], y_1[:] = y_1, y

for n in range(K) :
    # Пересчитываем значения во внутренних узлах сетки на слое n+1
    for i in range(1, N) :
        y[i] = 2*y_1[i] - y_2[i] - C2*(y_1[i+1] - 2*y_1[i] + y_1[i-1])
    # Задаем граничные условия
    y[0] = 0; y[N] = 0
    # Изменяем переменные перед переходом на следующий
    # временной слой
    y_2[:], y_1[:] = y_1, y
# --- End snippet 1 ---
```

```
# --- Start snippet 2 ---
i = 0
ip1 = i+1
im1 = ip1  # i-1 -> i+1
y[i] = y_1[i] + gamma2*(y_1[im1] - 2*y[i] + y_1[ip1])

i = N
im1 = i-1
ip1 = im1  # i+1 -> i-1
y[i] = y_1[i] + gamma2*(y_1[im1] - 2*y[i] + y_1[ip1])
# --- End snippet 2 ---

# --- Start snippet 3 ---
for i in range(0, N+1):
    ip1 = i+1 if i < N+1 else i-1
    im1 = i-1 if i > 0 else i+1
    y[i] = y_1[i] + gamma2*(y_1[im1] - 2*y[i] + y_1[ip1])
# --- End snippet 3 ---

# --- Start snippet 4 ---
# Начальные условия
for i in Ix[1:-1]:
    y[i] = y_1[i] - 0.5*gamma2*(y_1[i+1] - 2*y_1[i] + y_1[i-1])

# Цикл по времени
for i in It[1:-1]:
    # Вычисление значений во внутренних узлах
    for i in Ix[1:-1]:
        y[i] = 2*y_2[i] - y_1[i] + \
               gamma2*(y_1[i+1] - 2*y_1[i] + y_1[i-1])
    # Вычисление граничных условий
    i = Ix[0]; y[i] = 0
    i = Ix[-1]; y[i] = 0
# --- End snippet 4 ---
```

---

2. Верификация программной реализации

Прежде чем реализовывать алгоритм, удобно добавить в уравнение (1) слагаемое, описывающее источник (правую часть), что даст свободу в выборе тестовых задач для верификации алгоритма.

2.1. Неоднородное уравнение

Рассмотрим следующую смешанную задачу для неоднородного волнового уравнения:

$$\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2} + f(x,t), \quad x \in (0,l), \quad t \in (0,T], \tag{12}$$

$$u(x,0) = I(x), \quad x \in [0,l], \tag{13}$$

$$\frac{\partial u(x,0)}{\partial t} = V(x), \quad x \in [0,l], \tag{14}$$

$$u(0,t) = 0, \quad t \in (0,T], \tag{15}$$

$$u(l,t) = 0, \quad t \in (0,T]. \tag{16}$$

Аппроксимируя задачу (12)–(16) (аналогично случаю однородного уравнения) разностной схемой второго порядка аппроксимации на сетке $\omega_{h\tau}$, получим рекуррентное соотношение

$$y_i^{n+1} = 2y_i^n - y_i^{n-1} + \gamma^2(y_{i+1}^n - 2y_i^n + y_{i-1}^n) + \tau^2 f_i^n, \tag{17}$$
$$i = 1, 2, \ldots, N-1, \; n = 1, 2, \ldots, K-1.$$

Кроме того аппроксимируя начальное условие (14) со вторым порядком

$$\frac{y_i^1 - y_i^{-1}}{2\tau} = V(x_i) \Rightarrow y_i^{-1} = y_i^1 - 2\tau V(x_i),$$

для нахождения значений приближенного решения на первом временном слое получим

$$y_i^1 = y_i^0 + \tau V(x_i) + \frac{\gamma^2}{2}\left(y_{i+1}^0 - 2y_i^0 + y_{i-1}^0\right) + \frac{\tau^2}{2} f_i^0. \tag{18}$$

2.2. Использование аналитического решения

Многие волновые задачи описывают синусоидальные по времени и пространству. Например, исходная задача (1)–(5) допускает точное решение

$$u_e(x,t) = A \sin \frac{\pi x}{l} \cos \frac{\pi c t}{l} \tag{19}$$

Это решение удовлетворяет однородному волновому уравнению, однородным граничным условиям, а также начальным условиям $I(x) = A \sin \frac{\pi x}{l}$ и $V = 0$.

Обычной практикой является использование точного решения для тестирования программной реализации. Однако численное решение y_i^n — это только некоторое приближение точного. Мы не знаем величину погрешности этого приближения и, следовательно, мы не можем знать возникает ли разница между y_i^n и $u_e(x_i, t_n)$ из-за математического приближения или из-за ошибок в программе. В частности, когда графики приближенного и точного решений выглядят похоже, возникает соблазн сделать

заключение о том, что программная реализация работает правильно. Однако, даже если графики выглядят похоже и точность кажется хорошей, все равно в программной реализации могут присутствовать существенные ошибки.

Единственный способ использовать точное решение вида (19) при верификации программы заключается в выполнении ряда расчетов, сгущая сетку, вычисляя интегральную погрешность на каждой сетке, и на основе этого оценить скорость сходимости метода.

В нашем случае порядок сходимости метода равен 2 (см. следующий раздел), значит, вычисленная скорость сходимости должна быть близка к 2 на достаточно мелкой сетке.

2.3. Пробные функции

Преимущество использования метода пробных функций заключается в том, что мы можем тестировать все варианты в задаче (12)–(16). Идея метода заключается в том, что мы выбираем некоторую функцию и получаем соответствующие правую часть, граничные и начальные условия, подставив эту функцию в задачу. Кроме того, мы можем выбирать функцию, которая удовлетворяет граничным условиям. Например,

$$u_e(x,t) = x(l-x)\sin t.$$

Подставляя эту функцию в уравнение (12), получаем

$$-x(l-x)\sin t = -c^2 2\sin t + f \Rightarrow f = (2c^2 - x(l-x))\sin t$$

Начальные условия будут следующие

$$u(x,0) = I(x) = 0, \frac{\partial u(x,0)}{\partial t} = V(x) = x(l-x).$$

Для проверки программного кода, также нужно провести серию расчетов на последовательности сгущающихся сеток, чтобы оценить скорость сходимости в предположении, что некоторая мера E погрешности зависит от шагов сетки следующим образом

$$E = C_t\tau^r + C_x h^p,$$

где C_t, C_x, r и p — постоянные. Постоянные r и p характеризуют порядок сходимости по времени и пространству соответственно. Из анализа погрешности аппроксимации разностной схемы, мы ожидаем, что $r = p = 2$.

Используя точное решение дифференциальной задачи, мы можем вычислить меру погрешности E на последовательности сгущающихся сеток и проверить наличие второго порядка точности $r = p = 2$. Мы не будем оценивать константы C_t и C_x.

Удобно ввести один параметр дискретизации $d = \tau = \hat{c}h$ с некоторой константой $\hat{c}$. Так как τ и h связаны числом Куранта $\tau = \gamma h/c$, положим

$d = \tau$, тогда $h = dc/\gamma$. Теперь выражения для меры в случае, когда $p = r$, погрешности упрощается

$$E = C_t\tau^r + C_x h^r = C_t d^r + C_x \left(\frac{c}{\gamma}\right)^r d^r = Dd^r, \quad D = C_t + C_x \left(\frac{c}{\gamma}\right)^r.$$

Выбирая начальный параметр дискретизации d_0, проводим серию расчетов для последовательности уменьшающихся шагов $d_k = 2^{-k}d_0$. Уменьшение шага в два раза необязательно, это обычный выбор. Для каждого расчета следует сохранять E и d. Наиболее часто в качестве меры погрешности используются ℓ^2- или ℓ^∞-нормы сеточной функции погрешности e_i^n:

$$E = \|e_i^n\|_{\ell^2} = \left(\sum_{n=0}^{K} \tau \sum_{i=0}^{K} (e_i^n)^2\right)^{1/2}, \quad e_i^n = u_e(x_i, t_n) - y_i^n, \tag{20}$$

$$E = \|e_i^n\|_{\ell^\infty} = \max_{i,n} |e_i^n|. \tag{21}$$

При программной реализации на языке Python мы можем вычислить на каждом временном шаге $\sum_i (e_i^n)^2$, а затем аккумулировать значение в некоторой переменной, например, `e2_sum`. А на последнем временном шаге выполнить что-то подобное `sqrt(dt*dx*e2_sum)`. Для ℓ^∞-нормы нужно сравнить максимум погрешности на временном слое `e.max()` с глобальной погрешностью, полученной на предыдущих временных слоях, например, так: `e_max = max(e_max, e.max())`.

Альтернативный способ измерения погрешности состоит в использовании только пространственной нормы на временном шаге, например, при значении времени T ($n = K$):

$$E = \|e_i^K\|_{\ell^2} = \left(\sum_{i=0}^{K} (e_i^K)^2\right)^{1/2}, \quad e_i^K = u_e(x_i, t_K) - y_i^K, \tag{22}$$

$$E = \|e_i^K\|_{\ell^\infty} = \max_{0 \le i \le N} |e_i^K|. \tag{23}$$

Главное, что мера погрешности E — это одно число.

Пусть E_k — мера погрешности при расчете с номером k и пусть h_k — соответствующий параметр дискретизации. Учитывая, что $E_k = Dd_k^r$ мы можем оценить r, сравнивая два последовательных расчета

$$E_{k+1} = Dd_{k+1}^r, \quad E_k = Dd_k^r.$$

Отсюда, выражая r, получим

$$r_k = \frac{\ln E_{k+1}/E_k}{\ln d_{k+1}/d_k}.$$

Так как r зависит от k, то добавили индекс к r: r_k, $k = 0, 1, \ldots, m-2$, где m — количество проведенных расчетов: $(d_0, E_0), (d_1, E_1), \ldots, (d_m, E_m)$.

В нашем случае ожидается, что $r = 2$ и, следовательно, последовательность r_k должна стремиться к 2 с ростом k.

2.4. Построение точного решения дискретной задачи

Используя метод пробных функций и точное аналитическое решение дифференциальной задачи, как упоминалось выше, мы можем оценить скорость сходимости и правильное асимптотическое поведение. Опыт показывает, что этот способ верификации достаточно хорош, так как многие ошибки в программной реализации приводят к нарушению скорости сходимости. Однако нам кажется, что для верификации программной реализации, более точный тест тот, который позволяет проверить совпадает ли численное решение с тем, которое точно должно быть. Это требует точного знания численной погрешности, которого мы обычно не можем получить. Однако, можно рассмотреть решение, для которого численная погрешность равна нулю, т.е. решение исходной дифференциальной задачи, которое так же является точным решением разностной схемы. Это часто возникает, когда решением дифференциальной задачи является полином небольшой степени. (Анализ погрешности аппроксимации приводит к оценке погрешности, содержащей производные решения. В нашем случае, погрешность аппроксимации содержит производные четвертого порядка по пространству и времени. Выбирая в качестве точного решения полином степени не выше третьей, мы получим погрешность равную нулю.)

Рассмотрим построение точного решения как дифференциальной так и разностной задачи. Выберем в качестве пробной функции полиномиальную (второго порядка по пространственной переменной и первого по временной переменной):

$$u_e(x,t) = x(l-x)(1+0.5t), \tag{24}$$

которое дает $f(x,t) = 2(1+t)c^2$. Это решение удовлетворяет однородным граничным условиям (15) и (16), а также начальным условиям (13) с $I(x) = x(l-x)$ и (14) с $V(x) = 0.5x(l-x)$.

Чтобы убедиться, что u_e является точным решением разностной схемы выполним вычисления

$$
\begin{aligned}
u^n_{e\bar{t}\bar{t},i} &= x_i(x_i - l)(t)^n_{\bar{t}\bar{t}} \\
&= x_i(x_i - l)\frac{1 + 0.5t_{n+1} - 2 - t_n + 1 + 0.5t_{n-1}}{\tau^2} \\
&= x_i(x_i - l)\tau\frac{0.5(n+1) - n + 0.5(n-1)}{\tau^2} = 0, \\
u^n_{e\bar{x}\bar{x},i} &= (1 + 0.5t_n)(lx - x^2)_{\bar{x}\bar{x},i} \\
&= (1 + 0.5t_n)(l(x)_{\bar{x}\bar{x},i} - (x^2)_{\bar{x}\bar{x},i}) \\
&= -(1 + 0.5t_n)\frac{x^2_{i+1} - 2x^2_i + x^2_{i-1}}{h^2} \\
&= -(1 + 0.5t_n)h^2\frac{(i+1)^2 - 2i^2 + (i-1)^2}{h^2} \\
&= -2(1 + 0.5t_n).
\end{aligned}
$$

Отсюда, $f^n_i = 2(1 + 0.5t_n)c^2$. Кроме того, $u_e(x_i, 0) = I(x_i)$ и $\frac{\partial u(x,0)}{\partial t} = V(x_i)$, а также $u_e(x_i, t_n)$ удовлетворяет разностному уравнению для вычисления приближенного решения на первом временном шаге (18).

Таким образом, точное решение дифференциальной задачи (24) является точным решением разностной схемы. Мы можем использовать его для проверки совпадения вычисленного приближенного решения y^n_i со значением $u_e(x_i, t_n)$ с учетом машинной точности, независимо от значения временных шагов h и τ. Тем не менее, следует учитывать ограничения на шаги из условия устойчивости, т.е. тесты следует выполнять только на сетках удовлетворяющих условию устойчивости, которое в нашем случае имеет вид $\gamma \leq 1$ и будет получено позже.

Замечание.

Произведение квадратичного или линейного выражений от разных независимых переменных, как показано выше, часто является точным решением как дифференциальной так и разностной задач, и может использоваться для верификации программной реализации алгоритма.

Однако, для одномерного волнового уравнения вида $\frac{\partial^2 u}{\partial t^2} = c^2\frac{\partial^2 u}{\partial t^2}$, как мы увидим далее, существует другой способ генерации точных решений, который состоит в только выборе числа Куранта равным единице, $\gamma = 1$!

3. Программная реализация

Представим полный вычислительный алгоритм, его реализация на языке Python, реализация анимации решения, и верификация программной реализации.

Основной вычислительный алгоритм представленный в пунктах 1.4 и 1.5 можно реализовать в виде функции, аргументами которой будут входные данные задачи. Физические параметры: c, $I(x)$, $V(x)$, $f(x,t)$, l и T. Вычислительные параметры — это шаги сетки τ и h.

Вместо шагов τ и h можно задать один из этих шагов и число Куранта γ, так как явный контроль за этим параметром удобен при анализе вычислительного алгоритма. Многие считают естественным задать размер пространственной сетки и установить значение числа узлов пространственной сетки N. В функции-солвере можно тогда вычислить $\tau = \gamma l/(cN)$. Однако для сравнения графиков функций $u(x,t)$ (как функций от x) для разных значений числа Куранта более удобно зафиксировать τ для всех γ и затем изменять h согласно $h = c\tau/\gamma$. При фиксированном временном шаге τ все кадры анимации будут соответствовать одному и тому же моменты времени и такой подход упрощает создание анимации для сравнения результатов моделирования с разным размером пространственной сетки. Построение графиков функций от x при разных размерах сетки тривиально. Таким образом, проще варьировать шаг h при расчетах, чем τ.

3.1. Функция обратного вызова для действий, заданных пользователем

Решение во всех узлах пространственной сетки на новом временном слое хранятся в массиве `y` длины $N+1$. Мы должны решить, что нам делать с полученным решением, например: построить график, проанализировать значения или записать массив в файл для дальнейшего использования. Решение о том, что делать, остается за пользователем и может быть реализовано в виде функции

```
user_action(u, x, t, n)
```

где `u` — решение в узлах пространственной сетки `x` на временном слое `t[n]`. Функцию `user_action` можно вызывать из солвера при нахождении решения на каждом n-ом временном слое.

Если пользователь решит построить график решения или сохранить его на диск на временном слое, он должен реализовать такую функцию и выбрать соответствующее действие внутри нее. Ниже будут приведены примеры таких пользовательский функций.

3.2. Функция-солвер

Первый вариант функции-солвера представлен ниже

```
def solver(I, V, f, c, l, tau, gamma, T, user_action=None):
    K = int(round(T/tau))
    t = np.linspace(0, K*tau, K+1)   # Сетка по времени
    dx = tau*c/float(gamma)
    N = int(round(l/dx))
    x = np.linspace(0, l, N+1)       # Пространственная сетка
    C2 = gamma**2                    # вспомогательная переменная
    if f is None or f == 0 :
        f = lambda x, t: 0
    if V is None or V == 0:
        V = lambda x: 0

    y   = np.zeros(N+1)   # Массив с решением на новом временном слое n+1
    y_1 = np.zeros(N+1)   # Решение на предыдущем слое n
    y_2 = np.zeros(N+1)   # Решение на слое n-1

    import time;  t0 = time.clock()  # для измерения процессорного времени

    # Задаем начальное условие
    for i in range(0,N+1):
        y_1[i] = I(x[i])

    if user_action is not None:
        user_action(y_1, x, t, 0)

    # Используем специальную формулу для расчета на первом
    # временном шаге с учетом du/dt = 0
    n = 0
    for i in range(1, N):
        y[i] = y_1[i] + tau*V(x[i]) + \
               0.5*C2*(y_1[i-1] - 2*y_1[i] + y_1[i+1]) + \
               0.5*tau**2*f(x[i], t[n])
    y[0] = 0;  y[N] = 0

    if user_action is not None:
        user_action(y, x, t, 1)

    # Изменяем переменные перед переходом на следующий
    # временной слой
    y_2[:] = y_1;  y_1[:] = y

    for n in range(1, K):
        # Пересчитываем значения во внутренних узлах сетки на слое n+1
        for i in range(1, N):
            y[i] = - y_2[i] + 2*y_1[i] + C2*(y_1[i-1] - 2*y_1[i] + y_1[i+1]) + tau**2*f(x[i], t[n])

        y[0] = 0; y[N] = 0 # Задаем граничные условия
        if user_action is not None:
            if user_action(y, x, t, n+1):
                break
        # Изменяем переменные перед переходом на следующий
        # временной слой
        y_2[:] = y_1;  y_1[:] = y
```

```
    cpu_time = t0 - time.clock()
    return y, x, t, cpu_time
```

3.3. Верификация: точное решение — полином второй степени

Для верификации программной реализации будем использовать тестовую задачу из пункта 2.1. Ниже представлен юнит-тест основанный на этой задаче и реализованный в соответствующей тестовой функции (совместимой с фреймворками для юнит-тестирования `nose` или `py.test`).

```
def test_quadratic():
    """
    Проверяет воспроизводится ли точно решение u(x,t)=x(l-x)(1+t/2).
    """

    def u_exact(x, t):
        return x*(l-x)*(1 + 0.5*t)

    def I(x):
        return u_exact(x, 0)

    def V(x):
        return 0.5*u_exact(x, 0)

    def f(x, t):
        return 2*(1 + 0.5*t)*c**2

    l = 2.5
    c = 1.5
    gamma = 0.75
    N = 6  # Используем грубую сетку
    tau = gamma*(l/N)/c
    T = 18

    def assert_no_error(u, x, t, n):
        u_e = u_exact(x, t[n])
        diff = np.abs(u - u_e).max()
        tol = 1E-13
        assert diff < tol

    solver(I, V, f, c, l, tau, gamma, T,
           user_action=assert_no_error)
```

Если эти функции поместить в файл wave1d_1.py то запустить юнит-тест можно используя `py.test` или `nosetests`:

Terminal

```
Terminal > py.test -s -v wave1d_1.py
Terminal > nosetests -s -v wave1d_1.py
```

Будут выполнены все функции с именами `test_*()`.

3.4. Визуализация: анимация решения

После верификации программной реализации солвера можно приступить к выполнению расчетов, а также к визуализации результатов (распространение волн) на экране. Так как функция `solver` ничего не знает о способе визуализации (в солевере вызывается функция обратного вызова `user_action(u, x, t, n)`), мы должны реализовать соответствующую функцию обратного вызова.

Функция для управления расчетом. Следующая функция `viz`

1. определяет функцию обратного вызова `user_action` для построения графика решения на каждом временном слое;
2. вызывает функцию “solver“;
3. объединяет все графики в видео файлы разных форматов.

```
def viz(
    I, V, f, c, l, tau, gamma, T,  # Параметры задачи
    umin, umax,                # Интервал для отображения и
    animate=True,              # Расчет с анимацией?
    tool='matplotlib',         # 'matplotlib' или 'scitools'
    solver_function=solver,    # Функция, реализующая алгоритм расчета
    ):
    """Запуск солвера и визуализации и на каждом временном слое."""

    def plot_u_st(u, x, t, n):
        """Функция user_action для солвера."""
        plt.plot(x, u, 'r-',
                 xlabel='x', ylabel='u',
                 axis=[0, l, umin, umax],
                 title='t=%f' % t[n], show=True)
        # Начальные данные отображаем на экране в течение 2 сек.
        # Далее меду временными слоями пауза 0.2 сек.
        time.sleep(2) if t[n] == 0 else time.sleep(0.2)
        plt.savefig('frame_%04d.png' % n)  # для генерации видео

    class PlotMatplotlib:
        def __call__(self, u, x, t, n):
            """Функция user_action для солвера."""
            if n == 0:
                plt.ion()
                self.lines = plt.plot(x, u, 'r-')
                plt.xlabel('x');  plt.ylabel('u')
                plt.axis([0, l, umin, umax])
                plt.legend(['t=%f' % t[n]], loc='lower left')
            else:
                self.lines[0].set_ydata(u)
                plt.legend(['t=%f' % t[n]], loc='lower left')
```

```
            plt.draw()
        time.sleep(2) if t[n] == 0 else time.sleep(0.2)
        plt.savefig('tmp_%04d.png' % n)  # для генерации видео

if tool == 'matplotlib':
    import matplotlib.pyplot as plt
    plot_u = PlotMatplotlib()
elif tool == 'scitools':
    import scitools.std as plt  # scitools.easyviz
    plot_u = plot_u_st
import time, glob, os

# Удаляем старые кадры
for filename in glob.glob('tmp_*.png'):
    os.remove(filename)

# Вызываем солвер и выполняем расчет
user_action = plot_u if animate else None
u, x, t, cpu = solver_function(
    I, V, f, c, l, tau, gamma, T, user_action)

# Генерируем видео файлы
fps = 4  # Количество кадров в секунду
codec2ext = dict(flv='flv', libx264='mp4', libvpx='webm',
                 libtheora='ogg')  # Видео форматы
filespec = 'tmp_%04d.png'
movie_program = 'ffmpeg'  # или 'avconv'
for codec in codec2ext:
    ext = codec2ext[codec]
    cmd = '%(movie_program)s -r %(fps)d -i %(filespec)s '\
          '-vcodec %(codec)s movie.%(ext)s' % vars()
    os.system(cmd)

if tool == 'scitools':
    # Создаем HTML для показа анимации в браузере
    plt.movie('tmp_*.png', encoder='html', fps=fps,
              output_file='movie.html')
return cpu
```

Анализ кода. Функция `viz` может использовать либо `scitools`, либо `matplotlib` для визуализации решения. Функция действий пользователя, основанная на `scitools` называется `plot_u_st`, тогда как функция, использующая `matplotlib`, чуть более сложная и реализована как класс и должна использовать выражения отличные от построения статических графиков. Библиотека `scitools` может использовать как `matplotlib` так и `gnuplot` (и много других графических программ) для построения графиков, но `gnuplot` более подходящая программа для больших значений N или для двумерных задач, так как `gnuplot` работает существенно быстрее при построении анимации на экране.

Функция внутри другой функции, такая как `plot_u_st` в представленном выше фрагменте кода, имеет доступ ко всем локальным переменным функции `viz`. Такой подход называется *включением* и является очень удоб-

ным. Например, модули `plt` и `time` определенные вне `plot_u_st` являются доступными для `plot_u_st`, когда эта функция вызывается (как `user_action`) в функции `solver`. Возможно использование классов вместо включений более понятно для понимания кода при реализации функции действий пользователя.

Функция `plot_u_st` просто вызывает стандартную команду `plot` модуля `scitools` для построения графика зависимости `u` от `x` в каждый момент времени `t[n]`. Для того, чтобы добиться гладкой анимации, команда `plot` должна принимать параметры вместо того, чтобы прерываться вызовом `xlabel`, `ylabel`, `axis`, `time` и `show`. Несколько вызовов функции `plot` будет автоматически вызывать анимацию на экране. Кроме того, мы сохраняем каждый кадр в файл с именами, где номер кадра дополнен нулями: `tmp_0000.png`, `tmp_0001.png` и т.д. Для этого используется соответствующий формат вывода `tmp_%04d.png`.

Солвер вызывается с аргументом `user_action = plot_u`. Если пользователь использует `scitools`, то `plot_u` — это функция `plot_u_st`, а для `matplotlib` параметр `plot_u` является экземпляром класса `PlotMatplotlib`. Также этот класс использует переменные, определенные в функции `viz`: `plt` и `time`. В случае использования `matplotlib` нужно первый график строить стандартным образом, а затем обновлять значения по оси :math:‘y‘ на графике для каждого временного слоя. Обновление требует активного использования значения, возвращаемого функцией `plt.plot` при первом построении графика. Это значение нужно было бы сохранять в локальной переменной, если бы мы использовали включение для функции действий пользователя при построении анимации на основе `matplotlib`. Проще сохранять эту переменную как свойство класса `self.lines`. Так как по существу данный класс является функцией, мы реализуем функцию как специальный метод `__call__` так, что экземпляр класса `plot_u(u, x, t, n)` может быть вызван как функция обратного вызова из `solver`.

Создание видео файлов. Из файлов `tmp_*.png`, содержащих кадры анимации, мы можем сгененрировать видео файлы. Мы используем программу `ffmpeg` (или `avconv`) для объединения отдельных графиков в видео файл в следующих форматах: Flash, MP4, WebM и Ogg. Обычная команда вызова `ffmpeg` (или `avconv`) для генерации видео файла в формате Ogg с частотой 4 кадра в секунду из набора файлов вида `tmp_%04.png`, выглядит следующим образом

Terminal

```
Terminal > ffmpeg -r 4 -i tmp_%04d.png -c:v libtheora movie.ogg
```

Для разных форматов должны быть указан соответствующий кодировщик: `flv` для Flash, `libvpx` для WebM и `libx264` для MP4:

Terminal

```
Terminal > ffmpeg -r 4 -i tmp_%04d.png -c:v flv movie.flv
Terminal > ffmpeg -r 4 -i tmp_%04d.png -c:v libvpx movie.webm
Terminal > ffmpeg -r 4 -i tmp_%04d.png -c:v libx264 movie.mp4
```

Для просмотра полученных видео файлов можно использовать медиа проигрыватели такие как `vlc`, `mplayer` и т.п.

Функция `viz` генерирует команду вызова `ffmpeg` или `avconv` с соответствующими аргументами для каждого формата. Задача существенно упрощается, если воспользоваться словарем `codec2ext` соответствия имени кодека расширению файла. Для того, чтобы быть уверенным, что любой браузер отобразит видео файл достаточно только два формата: MP4 и WebM.

При создании видео файлов, содержащих большое число графических файлов, с помощью команд `ffmpeg` или `avconv` могут возникать проблемы. Метод, который всегда будет работать заключается в проигрывании PNG файлов в браузере с использованием JavaScript в HTML файле. Пакет модулей `scitools` содержит функцию `movie` (или автономную команду `scitools movie`) для создания HTML страниц, содержащих такие проигрыватели. Вызов `plt.movie` в функции `viz` демонстрирует использование этой функции. Файл `movie.html` можно загрузить в браузере.

Пропуск кадров для ускорения анимации. Иногда большие значения T и малые значения τ приводят большому количеству кадров и медленному воспроизведению анимации на экране. Решение этой проблемы заключается в выборе общего числа кадров в анимации, `num_frames`, и построении графиков решения только для каждых `skip_frame` кадров. Например, задание `skip_frame = 5` приводит к построению каждого 5 кадра. Значение по умолчанию `skip_frame = 1` дает построение каждого кадра. Общее количество временных слоев (т.е. максимально возможное количество кадров) — это длина массива `t`, `t.size` (или `len(t)`), тогда если мы зададим количество кадров `num_frames` в анимации, мы должны строить каждый `t.size/num_frames` кадр:

```
skip_frame = int(t.size/float(num_frames))
if n % skip_frame == 0 or n == t.size - 1:
    st.plot(x, u, 'r-', ...)
```

Простой выбор количества кадров можно проиллюстрировать следующим образом: пусть всего у нас есть 801 кадр и мы хотим, чтобы только 60 кадров было построено. Значит мы должны строить каждый 801/60 кадр, т.е. каждый (`every`) 13 кадр. Операция `n % every` будет принимать значение ноль каждый раз, когда `n` делится на 13 без остатка.

3.5. Запуск варианта расчета

Первый пример использования солвера одномерного волнового уравнения будет связан с колебанием струны, имеющей начальное положение в виде

треугольника:

$$I(x) = \begin{cases} ax/x_0, & x < x_0, \\ a(l-x)/(l-x_0), & x \geq x_0. \end{cases} \tag{25}$$

Пусть $l = 75$ см, $x_0 = 0.8l$, $a = 5$ мм, и частота колебаний $\nu = 440$ Гц. Соотношение между скоростью волны c и частотой ν имеет вид $c = \nu\lambda$, где λ — длина волны, взятая равной $2l$. Отсутствуют внешние силы, поэтому $f = 0$, и в начальный момент времени струна находится в состоянии покоя, поэтому $V = 0$. Также мы должны задать τ.

Функция, устанавливающая физические и численные параметры и вызываемая из `viz` может иметь вид:

```
def guitar(gamma):
    """Треугольная волна."""
    l = 0.75
    x0 = 0.8*l
    a = 0.005
    freq = 440
    wavelength = 2*l
    c = freq*wavelength
    omega = 2*np.pi*freq
    num_periods = 1
    T = 2*np.pi/omega*num_periods
    # Выбираем tau таким же, как при условии устойчивости для N=50
    tau = l/50./c

    def I(x):
        return a*x/x0 if x < x0 else a/(l-x0)*(l-x)

    umin = -1.2*a;  umax = -umin
    cpu = viz(I, 0, 0, c, l, tau, gamma, T, umin, umax,
              animate=True, tool='scitools')
```

Соответствующий код представлен в файле wave1d_1.py.

3.6. Безразмерная модель

В зависимости от изучаемой модели, может понадобиться получить согласующиеся и обоснованные значения физических параметров. Пример моделирования гитарной струны иллюстрирует эту ситуацию. Однако, масштабировав (обезразмерив) математическую задачу, часто можно уйти от проблемы оценки физических параметров. Метод обезразмеривания состоит во введении новых независимых и зависимых переменных, благодаря чему из абсолютные значения не будут очень большими или малыми, а желательно близкими к единице. Введем безразмерные переменные

$$\bar{x} = \frac{x}{l}, \quad \bar{t} = \frac{c}{l}t, \quad \bar{u} = \frac{u}{a}.$$

Здесь l — характерный масштаб длины, например, размер области, a — характерный размер u, например, полученный из начальных данных $a = \max_x |I(x)|$. Подставив новые переменные, получим

$$\frac{\partial u}{\partial t} = \frac{al}{c}\frac{\partial \bar{u}}{\partial \bar{t}},$$

откуда, в случае $f = 0$ имеем

$$\frac{a^2 l^2}{c^2}\frac{\partial^2 \bar{u}}{\partial \bar{t}^2} = \frac{a^2 l^2}{c^2}\frac{\partial^2 \bar{u}}{\partial \bar{x}^2}.$$

Отбрасывая черту сверху у переменных, приходим к безразмерному волновому уравнению

$$\frac{\partial^2 u}{\partial t^2} = \frac{\partial^2 u}{\partial x^2}, \tag{26}$$

в котором отсутствует коэффициент c^2. Начальные условия масштабируются следующим образом

$$a\bar{u}(\bar{x}, 0) = I(l\bar{x})$$

и

$$\frac{a}{l/c}\frac{\partial \bar{u}(\bar{x}, 0)}{\partial \bar{t}} = V(l\bar{x}).$$

Отсюда

$$\bar{u}(\bar{x}, 0) = \frac{I(l\bar{x})}{\max_x |I(x)|}, \quad \frac{\partial \bar{u}(\bar{x}, 0)}{\partial \bar{t}} = \frac{l}{ac}V(l\bar{x}).$$

В случае, когда $V(x) = 0$, видим, что в математической модели отсутствуют физические параметры.

Если у нас есть реализована программа для математической модели, учитывающей физические параметры и размерности, мы можем получить безразмерную версию, выбрав $c = 1$. Начальное условия для моделирования гитарной струны (25) может быть обезразмерено с помощью выбора следующих параметров $a = 1$, $l = 1$ и $x_0 \in [0, 1]$. Это означает, что мы должны выбирать только значение x_0 как долю единицы, так как значения остальных параметров равны единице. В коде мы должны только задать `a = c = l = 1`, `x_0 = 0.8` и больше не нужно никаких вычислений длины волны и частоты для оценки коэффициента c.

Осталось оценить в обезразмеренной задаче конечный момент времени, или более точно, оценить как этот момент связан с количеством периодом колебаний, так как часто возникает потребность задавать конечный момент времени как некоторое количество периодов. В безразмерной модели период колебаний равен 2, таким образом, конечный момент времени может задаваться как желаемое количество периодов, умноженное на 2.

Почему безразмерный период равен 2? Предположим, что u ведет себя как $\cos(\omega t)$ в зависимости от временной переменной. Соответствующий

период тогда равен $P = 2\pi/\omega$, но мы должны оценить ω. Естественное решение волнового уравнения имеет вид $u(x,t) = A\cos(kx)\sin(\omega t)$, где A — амплитуда, а k связано с длиной волны λ в пространстве: $\lambda = 2\pi/k$. Как λ, так и A будут заданы начальным условием $I(x)$. Подставляя $u(x,t)$ в волновое уравнение получим $-\omega^2 = -c^2k^2$, т.е. $\omega = ck$. Следовательно, период равен $P = 2\pi/(kc)$. Если для граничных условий выполнено $u(0,t) = u(l,t)$, будем иметь $kl = n\pi$, $n \in \mathbb{Z}$. Тогда $P = 2l/(nc)$. Максимальный период $P = 2l/c$. Безразмерный период $\tilde{P}$ получаем делением P на временной масштаб l/c, что дает $\tilde{P} = 2$. Кратчайшие волны в начальных условиях будут иметь безразмерный период $\tilde{P} = 2/n$ ($n > 1$).

4. Векторизация

Вычислительный алгоритм решения волнового уравнения в каждом узле сетки выполняет по заданной формуле вычисление нового значения y_i^{n+1}. Программно это реализовано посредством цикла по элементам массива. Такие циклы могут выполнятся медленно в Python (и аналогичных интерпретируемых языках таких как R и MATLAB). Один из методов ускорения циклов заключается в выполнении операций с целых массивах вместо работы с одним элементом массива в текущий момент времени. Это называют *векторизацией* или *векторными вычислениями*. Операции над целыми массивами возможны, если вычисления, затрагивающие каждый элемент, не зависят от других элементов. Векторизация не только ускоряет работу программы на последовательных компьютерах, но также делают программу проще для использования параллельных вычислений.

4.1. Операции на срезах массивов

Эффективное применение `numpy` требует, чтобы мы избегали использования циклов, а проводили вычисления с целыми массивами за один раз (или как минимум с большими частями массивов). Рассмотрим такое вычисление разностей $d_i = u_{i+1} - u_i$:

```
n = u.size
for i in range(0, n-1):
    d[i] = u[i+1] - u[i]
```

Все разности в этом случае не зависят друг от друга. Вычисление массива `d` может, таким образом, быть получено вычитанием массива `[u[0], u[1], ..., u[n-1]]` из массива, в котором элементы сдвинуты на один индекс вперед (см. рис. 3). Первое подмножество массива можно выразить следующим образом `u[0:n-1]`, `u[0:-1]` или просто `u[:-1]`, т.е. элементы с индексами от `0` до `n-2`. Второе подмножество можно получить так `u[1:n]` или `u[1:]`, т.е. элементы с индексами от `1` до `n-1`. Вычисление `d` теперь можно выполнить без явных циклов на Python:

```
d = u[1:] - u[:-1]
```

или с явным указанием границ:

```
d = u[1:n] - u[0:n-1]
```

Индексы с двоеточием, идущие от одного до (но не включая его) другого индекса называются *срезами*. При использовании массивов `numpy` вычисления выполняются все еще с использованием циклов, но посредством эффективного компилированного оптимизированного C или Fortran кода. Такие циклы иногда называются *векторизованными циклами*. Такие циклы могут также легко быть распределены между многими процессорами на параллельных компьютерах. Будем говорить, что *скалярный код*, работающий с одним элементов в конкретный момент времени, заменен на эквивалентный *векторизованный код*. Процесс получения векторизованного кода называется *векторизацией*.

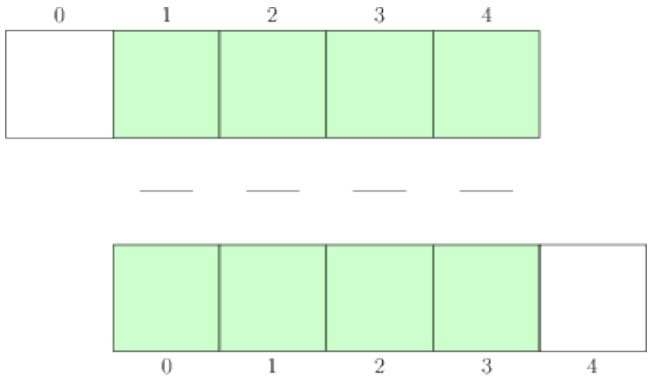

Рис. 3: Иллюстрация вычитания срезов двух массивов

Преимущество векторизованных вычислений.

Для понимания преимущества векторизованных вычислений задайте любой небольшой массив u, например, из пяти элементов, и попробуйте смоделировать на бумаге как циклическую, так и векторизованную версии рассмотренной выше операции.

Конечно-разностные схемы в своей основе содержат разности между элементами массивов со сдвинутыми индексами. Например, рассмотрим формулу вычисления значений на новом временном слое

```
for i in range(1, n-1):
    u2[i] = u[i-1] - 2*u[i] + u[i+1]
```

Векторизация состоит в замене цикла на арифметику срезов массивов размера `n-2`:

```
u2 = u[:-2] - 2*u[1:-1] + u[2:]
```

или

```
u2 = u[0:n-2] - 2*u[1:n-1] + u[2:n]
```

Отметим, что длина массива `u2` становится равной `n-2`. Если массив `u2` — массив длины `n` и нам нужно использовать формулы пересчета значений во "внутренних" элементах массива `u2`, мы можем написать

```
u2[1:-1] = u[:-2] - 2*u[1:-1] + u[2:]
```

или

```
u2[1:n-1] = u[0:n-2] - 2*u[1:n-1] + u[2:n]
```

Правая часть первого выражения осуществляется следующими шагами, привлекающими временные массивы с промежуточными результатами, так как каждый операция над массивами может использовать один или два массива. Пакет `numpy` осуществляет первое выражение за четыре шага:

```
temp1 = 2*u[1:-1]
temp2 = u[:2] - temp1
temp3 = temp2 + u[2:]
u2[1:-1] = temp3
```

Нам требуется три временных массива, но пользователь не должен беспокоится о таких временных массивах.

Распространенные ошибки при использовании срезов.

Выражения со срезами массивов требуют, чтобы срезы имели одну и ту же форму (shape). Легко сделать ошибку, например, в

```
u2[1:n-1] = u[0:n-2] - 2*u[1:n-1] + u[2:n]
```

и написать

```
u2[1:n-1] = u[0:n-2] - 2*u[1:n-1] + u[1:n]
```

Теперь `u[1:n]` имеет длину `n-1` в отличие от других срезов массива, что приводит к ошибке `ValueError` и появлению сообщения `could not broadcast input array from shape 103 into shape 104` (если `n` равно `105`). Когда возникает такая ошибка нужно тщательно проверить все срезы. Обычно, проще получить правильно верхнюю границу среза используя `-1` или `-2` или пустую границу, в отличие от использования в выражении длины массива.

Еще одна распространенная ошибка заключается в том, что пользователь забывает указать срез в левой части выражения

```
u2 = u[0:n-2] - 2*u[1:n-1] + u[2:n]
```

Это на самом деле критично: теперь `u2` становится *новым* массивом неправильного размера `n-2`, так как в нем будут отсутствовать граничные значения.

Векторизация также хорошо работает при использовании функций. Для того, чтобы проиллюстрировать это, мы можем расширить предыдущий пример следующим образом:

```
def f(x):
    return x**2 + 1

for i in range(1, n-1):
    u2[i] = u[i-1] - 2*u[i] + u[i+1] + f(x[i])
```

Векторизованный вариант может быть записан следующим образом:

```
u2[1:-1] = u[:-2] - 2*u[1:-1] + u[1:] + f(x[1:-1])
```

Очевидно, что `f` должна иметь возможность принимать в качестве аргумента массив, чтобы выражение `f(x[1:-1])` имело смысл.

4.2. Разностные схемы, выраженные в срезах

Перейдем к векторизации вычислительного алгоритма, математическое описание которого дано в разделе 1.4, а программная реализация описана в разделе 3.2. Алгоритм содержит три цикла: один для задания начальных

данных, один для расчета значений на первом временном слое, и, наконец, цикл, который повторяется для последовательных временных слоев. Рассмотрим векторизацию последнего цикла:

```
for i in range(1, N):
    u[i] = 2*u_1[i] - u_2[i] + \
    C2*(u_1[i-1] - 2*u_1[i] + u_1[i+1])
```

Его векторизованная версия может быть записана следующим образом:

```
u[1:-1] = - u_2[1:-1] + 2*u_1[1:-1] + \
          C2*(u_1[:-2] - 2*u_1[1:-1] + u_1[2:])
```

или

```
u[1:N] = - u_2[1:N] + 2*u_1[1:N] + \
         C2*(u_1[:N-1] - 2*u_1[1:N] + u_1[2:N+1])
```

Программа wave1d_v.py содержит новую версию функции `solver`, в которой используются как скалярные, так и векторизованные циклы (аргумент `version` может принимать значения `scalar` или `vectorized`, соответственно).

4.3. Верификация

Мы можем повторно использовать квадратичное решение $u_e(x,t) = x(l-x)(1+0.5t)$ для верификации векторизованного кода. Тестовая функция может проверять как скалярную, так и векторизованную версии. Кроме того, мы можем использовать функцию `user_action`, которая сравнивает точное и расчитанное решения на каждом временном слое и выполнять тест:

```
def test_quadratic():
    """
    Проверяет воспроизводят ли скалярная и векторизованная версии
    решение u(x,t)=x(l-x)(1+t/2) точно.
    """
    # Следующие функции должны работать при x заданном как массив или скаляр
    u_exact = lambda x, t: x*(l - x)*(1 + 0.5*t)
    I = lambda x: u_exact(x, 0)
    V = lambda x: 0.5*u_exact(x, 0)
    # f --- скаляр (zeros_like(x) тоже работает для скалярного x)
    f = lambda x, t: np.zeros_like(x) + 2*c**2*(1 + 0.5*t)

    l = 2.5
    c = 1.5
```

```
gamma = 0.75
N = 3  # Очень грубая сетка для теста
tau = gamma*(l/N)/c
T = 18

def assert_no_error(y, x, t, n):
    u_e = u_exact(x, t[n])
    tol = 1E-13
    diff = np.abs(y - u_e).max()
    assert diff < tol

solver(I, V, f, c, l, tau, gamma, T,
       user_action=assert_no_error, version='scalar')
solver(I, V, f, c, l, tau, gamma, T,
       user_action=assert_no_error, version='vectorized')
```

Лямбда-функции.

Представленный выше фрагмент кода иллюстрирует, как получить компактный код без потери читабельности, используя лямбда-функции для разных входных параметров-функций. По существу, код

```
f = lambda x, t: l*(x-t)**2
```

эквивалентен следующему

```
def f(x, t):
    return l*(x-t)**2
```

Отметим, что лямбда-функции могут содержать только одно выражение, а не операторы.

Одним из преимуществ лямбда-функций является то, что они могут быть использованы непосредственно в вызовах:

```
solver(I=lambda x: sin(pi*x/L), V=0, f=0, ...)
```

4.4. Измерение эффективности

В сценарии wave1d_v.py содержится новая функция `solver`, в которой реализованы как скалярные так и векторизованные вычисления. Для оценки эффективности векторизованного варианта по сравнению со скалярным нам потребуется функция `viz` рассмотренная в разделе 3.4. Ее можно использовать всю за исключением вызова функции-солвера. В этом вызове

отсутствует параметр `version`, который нам понадобиться для измерения эффективности.

Одно из решений этого вопроса — скопировать функцию `viz` из сценария wave1d_1.py в сценарий wave1d_v.py и добавить аргумент `version` в вызов `solver_function`. Однако, в этом случае мы будем дублировать большой фрагмент сложного кода, реализующего анимацию, поэтому такой подход — не очень хорошая идея. Добавление параметра 'version'в функции `wave1d_1.py.viz` тоже плохое решение, так как этот параметр не имеет смысла в сценарии wave1d_1.py.

Решение 1. Вызов функции `viz` из wave1d_1.py с параметром `solver_function` заданным, как наш новый `solver` из wave1d_v.py — приемлемый вариант, так как параметр `version` по умолчанию установлен в значение `'vectorized'`. Новую функцию `viz` в wave1d_v.py которая имеет параметр `version` и вызывает только `wave1d_1.viz`, можно реализовать следующим образом:

```
def viz(
    I, V, f, c, l, tau, gamma, T,  # Параметры задачи
    umin, umax,                # Интервал для отображения и
    animate=True,              # Расчет с анимацией?
    tool='matplotlib',         # 'matplotlib' или 'scitools'
    solver_function=solver,    # Функция, реализующая алгоритм
    version='vectorized',      # 'scalar' или 'vectorized'
    ):
    import wave1d_1
    if version == 'vectorized':
        # Повторно использует viz из wave1d_1, но с новой
        # векторизованной функцией solver из данного модуля
        # (где version='vectorized' задан по умолчанию;
        # wave1d_1.viz не имеет этого аргумента)
        cpu = wave1d_1.viz(
            I, V, f, c, l, tau, gamma, T, umin, umax,
            animate, tool, solver_function=solver)
    elif version == 'scalar':
        # Вызваем wave1d_1.viz со скалярным солвером
        # и используем wave1d_1.solver.
        cpu = wave1d_1.viz(
            I, V, f, c, l, tau, gamma, T, umin, umax,
            animate, tool,
            solver_function=wave1d_1.solver)
```

Решение 2. Существует более продвинутое решение, использующее очень полезный "трюк": мы можем объявить новую функцию, которая будет всегда вызывать `wave1d_v.solver` с параметром `version='scalar'`. Функция Python `functools.partial` принимает функцию `func` в качестве аргумента и ряд других параметров и возвращает новую функцию, которая вызывает `func` с заданными аргументами. Рассмотрим простейший пример:

```
def f(a, b, c=2):
    return a + b + c
```

Мы хотим, чтобы функция `f` всегда вызывалась с `c=3`, т.е. чтобы `f` имела только два варьируемых параметра `a` и `b`. Это можно получить следующим образом:

```
import functools
f2 = functools.partial(f, c=3)

print f2(1, 2) # результат: 1 + 2 + 3 = 6
```

Теперь функция `f2` вызывает `f` с любыми заданными параметрами `a` и `b`, но `c` всегда будет иметь значение `3`.

В функции `viz` можно сделать следующее:

```
import functools

scalar_solver = functools.partial(solver, version='scalar')

cpu = wave1d_1.viz(
      I, V, f, c, l, tau, gamma, T, umin, umax,
      animate, tool,
      solver_function=scalar_solver)
```

Новая функция `scalar_solver` принимает те же аргументы, что и `wave1d_1.solver`, а вызывает `wave1d_v.solver`, но всегда задает параметр `version='scalar'`.

Эксперименты по проверке эффективности. Теперь у нас есть функция `viz`, которая может вызывать солвер в режиме как скалярных, так и векторизованных вычислений. Функция `run_efficiency_experiments` в wave1d_v.py выполняет серию экспериментов и сообщает процессорное время, затраченное скалярным и векторизованным солверами для задачи о колебании струны с количеством узлов пространственной сетки $N = 50, 100, 200, 400, 800$. Запуск этой функции показывает, что векторизованные вычисления существенно быстрее: векторизованный код работает примерно в $N/10$ раз быстрее, чем скалярный.

4.5. Замечание об обновлении массивов

В конце расчета на каждом временном слое мы должны обновить массивы `y_2` и `y_1` так, чтобы они содержали правильные значения для расчета на следующем временном слое:

```
y_2[:] = y_1
y_1[:] = y
```

Здесь важен порядок! Если сначала обновить `y_1`, то массив `y_2` будет равен `y`, что неправильно.

Присваивание `y_1[:] = y` копирует содержимое массива `y` в элементы массива `y_1`. Такое копирование занимает время, но это время незначительно по сравнению со временем необходимым для вычисления "y" по разностной схеме, даже если эти вычисления векторизованы. Однако, эффективность программного кода — это ключевой момент при численном решении задач для уравнений в частных производных (в частности, двумерных и трехмерных задач), поэтому стоит отметить, что существует более эффективный способ обновления массивов `y_2` и `y_1` для расчета на новом временном слое. Идея основана на переключающихся ссылках.

Переменные в Python — это, на самом деле, ссылки на некоторые объекты. Вместо копирования данных, мы можем указать, что `y_2` ссылается на объект `y_1`, а `y_1` ссылается на объект `y`. Это очень быстрая операция. Простая реализация вида

```
y_2 = y_1
y_1 = y
```

будет ошибочной, потому что теперь `y_2` ссылается на объект `y_1`, но теперь `y_1` — это ссылка на объект `y`, так что теперь объект `y` имеет две ссылки, при этом наш третий массив, на который изначально ссылалась переменная `y_2`, больше не имеет ссылки и поэтому потерян. Это означает, что переменные `y_2`, `y_1` и `y` ссылаются на массива, а не на три. Следовательно, вычисления на следующем временном слое будут перемешаны, так как изменение элементов `y` будет приводить также к изменению элементов `y_1`. Поэтому решение на предыдущем временном слое нарушается.

В то время как выражение `y_2 = y_1` сработает хорошо, выражение `y_1 = y` вызовет проблемы. Чтобы избежать этой проблемы нужно быть уверенным, что `y` будет ссылаться на массив `y_2`. Математически это неправильно, но новые корректные значения будут записаны в `y` при расчете на следующем временном слое.

Корректное переключение ссылок имеет вид:

```
tmp = y_2
y_2 = y_1
y_1 = y
y = tmp
```

Можно избавиться от временной ссылки `tmp`, используя следующую запись:

```
y_2, y_1, y = y_1, y, y_2
```

Такое переключение ссылок будет использоваться нами в дальнейших программных реализациях.

Предупреждение.

Обновление `y_2, y_1, y = y_1, y, y_2` оставляет неправильные значения на последнем временном слое. Это значит, что если мы будем возвращать `y`, как делалось в примерах кода, мы, на самом деле, вернем `y_2`, что неправильно. Поэтому важно скорректировать содержимое `y` перед его возвращением следующим образом: `y = y_1`.

5. Упражнения

Упражнение 1: Моделирование стоячей волны

Цель данного упражнения — провести моделирование стоячей волны на отрезке $[0, l]$ и проиллюстрировать расчет погрешности. Стоячие волны порождаются начальным условием

$$u(x, 0) = A \sin \frac{\pi m x}{l}$$

где m — целое число, A — заданная амплитуда. Соответствующее точное решение может быть получено:

$$u_e(x, t) = A \sin \frac{\pi m x}{l} \cos \frac{\pi m c t}{l}.$$

1. Показать, что для функции $\sin kx \cos \omega t$ длина волны в пространстве $\lambda = 2\pi/k$ и период по времени $P = 2\pi/\omega$. Используйте эти выражения для нахождения соответствующих пространственной длины волны и периода по времени функции u_e.

2. Импортировать функцию `solver` из сценария wave1d_1.py в новый файл, где функция `viz` будет реализована так, что будут строится графики содержащие или численное и точное решение, или погрешность.

3. Создать анимацию, где будет иллюстрироваться как погрешность $e_i^n = u(x, t_n) - y_i^n$ будет расти и уменьшаться со временем. Также создать анимацию совместно изменяющихся y и u_e.

Подсказка 1. Необходим достаточно длинный расчет, чтобы увидеть существенное несоответствие между точным и приближенным решениями.

Подсказка 2. Возможный набор параметров: $l = 12$, $m = 9$, $c = 2$, $A = 1$, $N_x = 80$, $\gamma = 0.8$. Сеточная функция погрешности e^n можно вычислить для 10 периодов, в то время как для того, чтобы увидеть существенную разницу между точным и приближенным решениями, нужно 20–30 периодов. Имя файла: `wave_standing.py`.

Замечания. Важными параметрами, влияющими на качество численного решения, являются число Куранта $\gamma = c\tau/h$ и соотношение kh, пропорциональное числу узлов сетки на длину волны.

Упражнение 2: Добавить сохранение решения в функции действий пользователя

Расширить функцию `plot_u` в файле wave1d_1.py, чтобы также сохранялось решение `y` в список. С этой целью объявите в функции `viz` вне `plot_u` переменную `all_u` как пустой список выполните операцию добавления элемента к списку внутри функции `plot_u`. Заметьте, что функция, как `plot_u`, внутри другой функции, как `viz`, помнит все локальные переменные функции `viz`, включая `all_u`, даже когда `plot_u` вызывается (как `user_action`) из функции `solver`. Протестируйте как `all_u.append(y)` так и `all_u.append(y.copy)`. Почему одна из этих конструкций не выполняет сохранение решения корректно? Пусть функция `viz` возвращает список `all_u`, преобразованный в двумерный массив `numpy`. Имя файла: `wave1d_1_store.py`.

Упражнение 3: Использование класса для функции действий пользователя

Переделать 2, используя класс для функции `user_action`, т.е. определить класс `Action`, у которого список `all_u` является свойством, и реализовать функцию действий пользователя как метод этого класса (специальный метод `__call__` — естественный выбор). Такая версия исключает ситуацию, когда функция действий пользователя зависит от параметров, определенных вне функции (такую как в 2). Имя файла: `wave1d_1_s2c.py`.

Упражнение 4: Сравнение нескольких чисел Куранта на одном видео

Цель упражнения — сделать видео, на котором отображается несколько кривых, соответствующих разным числам Куранта. Импортировать `solver` из wave1d_1.py в новый файл `wave_compare.py`. Заново реализовать функцию `viz` так, чтобы она получала в качестве аргумента список значений `gamma` и создавала анимацию с решениями, соответствующими заданным

значениям `gamma`. Функция `plot_u` должна быть изменена для того, чтобы сохранять значения в массив (см. 2 или 3), `solver` должен выполняться для каждого значения числа Куранта, и, наконец, на всех временных слоях должны быть построены графики решений и решения должны быть сохранены в файл.

Главная проблема такой визуализации заключается в том, что мы должны быть уверены, что графики решений на одном кадре соответствуют одному и тому же моменту времени. Простейший способ устранения проблемы — это оставить шаги по времени и пространству постоянными, а менять скорость волны c для того, чтобы изменялось число Куранта. Имя файла: `wave_compare.py`.

Проект 5: Исчисления с одномерными сеточными функциями

Этот проект — изучение интегрирования и дифференцирования сеточных функций, как в скалярной так и в векторизованной реализации. Пусть задана сеточная функция f_i на одномерной пространственной сетке

$$\omega_h = \{x_i = a + ih,\ i = 0, 1, \ldots N,\ h = (b-a)/N\},$$

заданной на отрезке $[a, b]$.

1) Определить разностную производную от f_i, используя центральную производную во внутренних узлах секи и направленные производные на концах отрезков. Реализовать программно скалярную версию вычислений в виде функции Python и написать соответствующий юнит-тест для линейной функции $f(x) = 4x - 2.5$, для которой разностные производные должны совпадать с непрерывными.

2) Векторизовать реализацию вычисления разностных производных. Расширить юнит-тест для проверки адекватности такой реализации.

3) Для вычисления дискретного интеграла F_i от f_i, предположим, что сеточная функция f_i изменяется линейно между узлами сетки (линейна или кусочно-линейна). Пусть $f(x)$ — линейная интерполяция f_i. Тогда имеем

$$F_i = \int_{x_0}^{x_i} f(x)dx$$

Точный интеграл кусочно-линейной от функции $f(x)$ дается формулой трапеций. Показать, что если вычислено значение F_i, то F_{i+1} можно получить следующим образом

$$F_{i+1} = F_i + \frac{1}{2}\left(f_i + f_{i+1}\right)h$$

Создать функцию скалярной реализации дискретного интеграла как сеточной функции. Это значит, что функция должна возвращать F_i для

$i = 0, 1, \ldots, N$. Для юнит-тестирования используйте тот факт, что описанный выше дискретный интеграл от линейной функции (например, $f(x) = 4x - 2.5$) дает точное значение интеграла.

4) Векторизовать программную реализацию дискретного интеграла. Расширить юнит-тест для проверки адекватности такой реализации.

5) Создать класс `MeshCalculus`, который позволит интегрировать и дифференцировать сеточные функции. В классе должны быть определены методы, которые вызывают созданные раньше функции. Ниже приведен пример использования класса:

```
import numpy as np
calc = MeshCalculus(vectorized=True)
x = np.linspace(a, b, 11)     # сетка
f = np.exp(x)                 # сеточная функция
df = calc.differentiate(f, x) # разностная производная
F = calc.integrate(f, x)      # дискретный интеграл
```

Подсказка. Представьте рекуррентную формулу для F_{i+1} в виде суммы. Создайте массив, содержащий элементы суммы, и используйте "cumsum" (`numpy.cumsum`) операцию для вычисления накапливаемой суммы: `numpy.cumsum([1,3, 5])` даст `[1, 4, 9]`.
Имя файла: `mesh_calculus_1d.py`.

6. Обобщения: отражающие границы

Граничные условия $u = 0$ для волнового уравнения означают отражение волны, но при этом u меняет знак на границе, условие же $\frac{\partial u}{\partial x} = 0$ на границе означает отражение волны с сохранением знака решения.

Следующая задача, которую мы рассмотрим заключается в реализации граничного условия второго рода (*условие Неймана*) $\frac{\partial u}{\partial x} = 0$, которое является более сложным для численной реализации, чем *условие Дирихле*, т.е. при заданном значении u на границе. Ниже мы приведем два способа разностной аппроксимации условий Неймана: один из них основан на построении модифицированного шаблона вблизи границы, а второй основан на расширении сетки мнимыми ячейками и узлами.

6.1. Граничные условия Неймана

Для описания процесса, когда волна ударяется в границу и отражается назад, используется условие

$$\frac{\partial u}{\partial \mathbf{n}} \equiv \mathbf{n} \cdot \nabla u = 0, \tag{27}$$

где $\partial/\partial\mathbf{n}$ — производная вдоль нормали, внешней к границе. В одномерном случае (отрезок $[0, l]$), имеем

$$\left.\frac{\partial}{\partial\mathbf{n}}\right|_{x=l} = \frac{\partial}{\partial x}, \quad \left.\frac{\partial}{\partial\mathbf{n}}\right|_{x=0} = -\frac{\partial}{\partial x}$$

6.2. Аппроксимация производной на границе

Построим аппроксимацию граничного условия (27) со вторым порядком аппроксимации по пространственной переменной. Для этого воспользуемся центральной разностной производной:

$$y^n_{\hat{x},0} = \frac{y^n_1 - y^n_{-1}}{2h} = 0. \tag{28}$$

Проблема заключается в том, что y^n_{-1} не является расчетным значением, так как задано в нерасчетном узле не принадлежащем сетке. Однако, если мы объединим (28) с разностным уравнением (8), записанным в узле $i = 0$:

$$y^{n+1}_0 = 2y^n_0 - y^{n-1}_0 + \gamma^2(y^n_1 - 2y^n_0 + y^n_{-1}), \tag{29}$$

мы можем исключить фиктивное значение y^n_{-1}. Учитывая (28), имеем $y^n_{-1} = y^n_1$. Подставив последнее в (29), получим модифицированное уравнение в граничной точке y^{n+1}_0:

$$y^{n+1}_0 = 2y^n_0 - y^{n-1}_0 + 2\gamma^2(y^n_1 - 2y^n_0). \tag{30}$$

На рис. 4 представлен шаблон схемы на левой границе области с учетом аппроксимации условия Неймана.

Аналогично, получаем апрокссимацию условия (27) на правой границе $x = l$:

$$y_{\hat{x},N} = \frac{y^n_{N+1} - y^n_{N-1}}{2h} = 0$$

Объединяя последнее с разностным уравнением (8) при $i = N$ получим модифицированное уравнение на правой границе:

$$y^{n+1}_N = 2y^n_N - y^{n-1}_N + 2\gamma^2(y^n_{N-1} - y^n_N).$$

Кроме того, на границах нужно построить модификацию разностного уравнения (11) для вычисления значений на первом временном шаге.

6.3. Программная реализация условий Неймана

В предыдущем пункте мы вывели специальные формулы для расчета вблизи границ. При этом, учитывая аппроксимацию условий Неймана центральной разностной производной, мы заменили значения y^n_{-1} на y^n_1 на левой границе и y^n_{N+1} на y^n_{N-1} на правой границе. Эти наблюдения могут легко

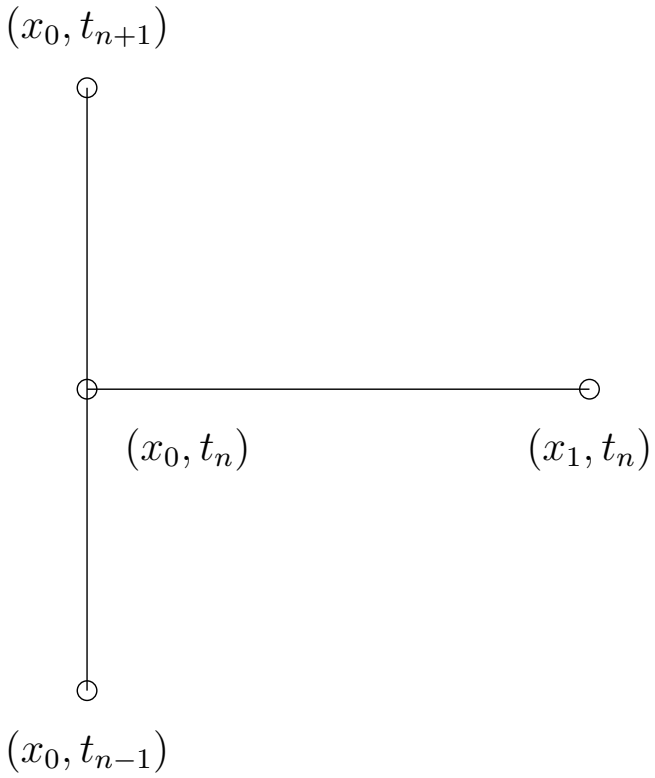

Рис. 4: Модифицированный шаблон на левой границе для аппроксимации условия Неймана.

использоваться при программной реализации: мы можем просто использовать общий шаблон во всех узлах сетки, но написать код так, чтобы можно было легко заменить `y[i-1]` на `y[i+1]` и наоборот. Этого можно добиться задавая индексы `i+1` и `i-1` как переменные `ip1` (`i` plus `1`) и `im1` (`i` minus `1`), соответственно. Следовательно на левой границе мы можем определить `im1 = i+1`, в то время как во внутренних узлах сетки `im1 = i-1`. Ниже представлена программная реализация такого подхода:

```
i = 0
ip1 = i+1
im1 = ip1  # i-1 -> i+1
y[i] = y_1[i] + gamma2*(y_1[im1] - 2*y[i] + y_1[ip1])

i = N
im1 = i-1
ip1 = im1  # i+1 -> i-1
y[i] = y_1[i] + gamma2*(y_1[im1] - 2*y[i] + y_1[ip1])
```

На самом деле мы можем создать один цикл как для граничных, так и для внутренних узлов и использовать одну формулу для вычисления значений на новом временном слое:

```
for i in range(0, N+1):
    ip1 = i+1 if i < N+1 else i-1
    im1 = i-1 if i > 0 else i+1
    y[i] = y_1[i] + gamma2*(y_1[im1] - 2*y[i] + y_1[ip1])
```

Сценарий wave1d_n0.py содержит полную программную реализацию решения одномерного волнового уравнения с граничными условиями Неймана. В нем реализован тест, использующий «волну-вилку» в качестве начального данного и проверяющий, что начальное состояние возвращается после одного периода. Но такой тест требует выполнения условия $\gamma = 1$, так как в этом случае численное решение совпадает с точным решением дифференциальной задачи.

6.4. Обозначение множеств индексов

Для того, чтобы улучшить математическую запись и программную реализацию, полезно ввести обозначения для множеств индексов. Это означает, что мы будем писать x, $i \in \mathcal{I}_x$ вместо $i = 0, 1, \ldots, N$. Очевидно, что $\mathcal{I}_x$ должно быть множеством индексов $\mathcal{I}_x = \{0, 1, \ldots, N\}$, но часто удобно использовать символ для этого множества, чем указывать все элементы этого множества. Такие обозначения делают описания алгоритмов и их программную реализацию более простыми.

Первый элемент этого множества будем обозначать $\mathcal{I}_x^0$, а последний $\mathcal{I}_x^{-1}$. Если нужно отбросить первый элемент множества, то будем использовать символ $\mathcal{I}_x^+$ для остального подмножества $\mathcal{I}_x^+ = \{1, 2, \ldots, N\}$. Аналогично, $\mathcal{I}_x^- = \{0, 1, \ldots, N-1\}$. Все индексы соответствующие внутренним узлам сетки обозначим $\mathcal{I}_x^i = \{1, 2, \ldots, N-1\}$.

В коде на Python для множеств индексов будет следующее соответствие:

Обозначение	Python
$\mathcal{I}_x$	`Ix`
$\mathcal{I}_x^0$	`Ix[0]`
$\mathcal{I}_x^{-1}$	`Ix[-1]`
$\mathcal{I}_x^-$	`Ix[:-1]`
$\mathcal{I}_x^+$	`Ix[1:]`
$\mathcal{I}_x^i$	`Ix[1:-1]`

Почему полезны множества индексов.

Важная характерная особенность использования множеств индексов заключается в том, что формулы и код программы не зависят от по-

рядка нумерации узлов сетки. Например, обозначение $i \in \mathcal{I}_x$ или $i \in \mathcal{I}_x^0$ остается одинаковым и для $\mathcal{I}_x$, определенном выше, и для $\mathcal{I}_x = \{1, 2, \ldots, Q\}$. Аналогично, в коде мы можем определить `Ix = range(N+1)` или `Ix = range(1,Q)`, а выражения типа `Ix[0]` и `Ix[1:-1]` остаются корректными. Один из примеров удобства использования такого подхода — это преобразование кода, написанного на языке, где нумерация массивов начинается с нуля (например, Python или C), в код на языке, где нумерация массивов начинается с единицы (например, MATLAB или Fortran). Другое важное применение — это реализация условий Неймана с помощью мнимых узлов.

В рассматриваемой нами задаче используются следующие множества индексов:

$$\mathcal{I}_x = \{0, 1, \ldots, N\}, \mathcal{I}_t = \{0, 1, \ldots, K\},$$

определяемые в Python следующим образом:

```
Ix = range(0, N+1)
It = range(0, K+1)
```

Используя множества индексов, разностную схему можно записать следующим образом:

$$\begin{aligned}
&y_i^{n+1} = y_i^n - \frac{1}{2}\gamma^2(y_{i+1}^n - 2y_i^n + y_{i-1}^n), \quad i \in \mathcal{I}_x, \quad n = 0,\\
&y_i^{n+1} = 2y_i^{n-1} - y_i^n + \gamma^2(y_{i+1}^n - 2y_i^n + y_{i-1}^n), \quad i \in \mathcal{I}_x^i, \quad n \in \mathcal{I}_t^i,\\
&y_i^{n+1} = 0, \quad i = \mathcal{I}_x^0, \quad t \in \mathcal{I}_t^-,\\
&y_i^{n+1} = 0, \quad i = \mathcal{I}_x^{-1}, \quad t \in \mathcal{I}_t^-.
\end{aligned}$$

Соответствующий программный код имеет вид:

```
# Начальные условия
for i in Ix[1:-1]:
    y[i] = y_1[i] - 0.5*gamma2*(y_1[i+1] - 2*y_1[i] + y_1[i-1])

# Цикл по времени
for i in It[1:-1]:
    # Вычисление значений во внутренних узлах
    for i in Ix[1:-1]:
        y[i] = 2*y_2[i] - y_1[i] + \
               gamma2*(y_1[i+1] - 2*y_1[i] + y_1[i-1])
    # Вычисление граничных условий
    i = Ix[0]; y[i] = 0
    i = Ix[-1]; y[i] = 0
```

Замечание.

Сценарий wave1d_n.py использует множества индексов и решает одномерное волновое уравнение с достаточно общими граничными и начальными условиями:

- $x = 0$: $u = u_l(t)$ или $\frac{\partial u}{\partial x} = 0$;
- $x = l$: $u = u_r(t)$ или $\frac{\partial u}{\partial x} = 0$;
- $t = 0$: $u = I(x)$;
- $t = 0$: $\frac{\partial u}{\partial t} = V(x)$.

Сценарий объединяет условия Дирихле и Неймана, скалярную и векторизованную реализацию разностной схемы, а также множества индексов. Большое количество тестовых примеров также включены в этот сценарий:

- начальное условие в форме «волны-вилки» (при $\gamma = 1$ решением будет прямоугольник смещающийся на одну ячейку за временной шаг);
- начальное условие как функция Гаусса;
- начальное условие в форме треугольного профиля, который похож на начальное положение гитарной струны;
- синусоидальное изменение решения при $x = 0$ и либо $u = 0$, либо $\frac{\partial u}{\partial x} = 0$ при $x = l$;
- точное аналитическое решение $u(x,t) = \cos\frac{m\pi t}{l}\sin\frac{m\pi x}{2l}$, которое может использоваться для проверки скорости сходимости.

6.5. Верификация реализации граничных условий Неймана

Перейдем к вопросу тестирования реализации условий Неймана. Функция `solver` wave1d_n.py реализованы как условия Дирихле и Неймана при $x = 0$ и $x = l$. Заманчиво было бы использовать решение типа квадратичной функции, однако эта функция не является точным решением задачи с условиями Неймана. Линейная функция также не подходит, так как реализованы только однородные условия Неймана, поэтому для тестирования будем использовать только постоянное решение $u = \text{const}$.

```
def test_constant():
    """
```

```
    Тестируем работу скалярной и векторизованой версий
    для постоянного u(x,t). Выполняем расчет на отрезке
    [0, l] и применяем условия Неймана и Дирихле на обоих
    границах.
    """
    u_const = 0.45
    u_exact = lambda x, t: u_const
    I = lambda x: u_exact(x, 0)
    V = lambda x: 0
    f = lambda x, t: 0

    def assert_no_error(y, x, t, n):
        u_e = u_exact(x, t[n])
        diff = np.abs(y - u_e).max()
        msg = 'diff=%E, t_%d=%g' % (diff, n, t[n])
        tol = 1E-13
        assert diff < tol, msg

    for ul in (None, lambda t: u_const):
        for ur in (None, lambda t: u_const):
            l = 2.5
            c = 1.5
            gamma = 0.75
            N = 3  # Очень грубая сетка для точного теста
            tau = gamma*(l/N)/c
            T = 18

            solver(I, V, f, c, ul, ur, l, tau, gamma, T,
                   user_action=assert_no_error,
                   version='scalar')
            solver(I, V, f, c, ul, ur, l, tau, gamma, T,
                   user_action=assert_no_error,
                   version='vectorized')
            print ul, ur
```

Другой тест основан на том факте, что погрешность аппроксимации равна нулю в случае когда число Куранта равно единице. Возьмем в качестве начальной функции «волну-площадку», пусть начальная функция распадается на две площадки, каждая смещается в своем направлении. Проверит, что эти две волны отразятся от границ и сформируют начальное распределение после одного периода. Соответствующая тестовая функция представлена ниже

```
def test_plug():
    """Тестирование возвращается для профиль-площадка после одного периода."""
    l = 1.0
    c = 0.5
    tau = (l/10)/c  # N=10
    I = lambda x: 0 if abs(x-l/2.0) > 0.1 else 1

    u_s, x, t, cpu = solver(
        I=I,
        V=None, f=None, c=0.5, ul=None, ur=None, l=l,
        tau=tau, gamma=1, T=4, user_action=None, version='scalar')
```

```
u_v, x, t, cpu = solver(
    I=I,
    V=None, f=None, c=0.5, ul=None, ur=None, l=l,
    tau=tau, gamma=1, T=4, user_action=None, version='vectorized')
tol = 1E-13
diff = abs(u_s - u_v).max()
assert diff < tol
u_0 = np.array([I(x_) for x_ in x])
diff = np.abs(u_s - u_0).max()
assert diff < tol
```

Остальные тесты используются для анализа погрешности аппроксимации.

6.6. Реализация граничных условий Неймана с использованием мнимых ячеек

Идея. Вместо модификации схемы на границе мы можем ввести дополнительные узлы вне области задачи, так что фиктивные значения y^n_{-1} и u^n_{N+1} будут определены на сетке. Добавление интервалов $[-h, 0]$ и $[l+h, 0]$, назовем их *мнимые ячейки* к расчетной сетке дает все узлы сетки, соответствующие $i = -1, 2, \ldots, N+1$. Дополнительные узлы $i = -1$ и $i = N+1$ назовем *мнимыми узлами*, а значения в этих узлах y^n_{-1} и y^n_{N+1} назовем *мнимыми значениями*

Основная идея состоит в том, чтобы быть уверенным, что всегда будет выполняться

$$u^n_{-1} = u^n_1 \quad \text{и} \quad u^n_{N+1} = u^n_{N-1},$$

потому что тогда использование стандартной разностной схемы в узлах $i = 0$ и $i = N$ будет корректным и будет гарантировать, что решение согласуется с граничным условием Неймана.

Программная реализация. Массив `y`, содержащий решение, должен содержать дополнительные элементы с мнимыми узлами:

```
y = zeros(N+3)
```

Массивы `y_1` и `y_2` необходимо определить аналогично.

К сожалению стандартная индексация массивов в Python (индексация начинается с 0), не удобна в случае использования рассматриваемого подхода. В этом случае возникает несоответствие математической индексации $i = -1, 0, 1, \ldots, N+1$ и индексации Python `0, 1, ..., N+2`. Один способ решения этой проблемы состоит в изменении математической нумерации в разностной схеме и записать

$$y^{n+1}_i = \ldots, \quad i = 1, 2, \ldots, N+1$$

вместо $i = 0, 1, \ldots, N$. В этом случае номера мнимых узлов будут $i = 0$ и $i = N + 2$. Можно предложить решение лучше основанное на использовании множеств индексов: мы скроем значения индексов и будем оперировать понятиями внутренних и граничных узлов.

С этой целью мы определим `y` нужной длины и `Ix` — соответствующие индексы реальных узлов

```
y   = np.zeros(N+3)
Ix = range(1, u.shape[0]-1)
```

Это значит, что граничные узлы будут иметь индексы `Ix[0]` и `Ix[-1]` (как и раньше). Сначала мы вычислим решение физических узлах (т.е. во внутренних узлах сетки):

```
        for i in Ix:
            y[i] = - y_2[i] + 2*y_1[i] + \
                   gamma2*(y_1[i-1] - 2*y_1[i] + y_1[i+1]) + \
                   tau2*f(x[i-Ix[0]], t[n])
```

Такое индексирование будет сложнее при вызове функций `V(x)` и `f(x,t)`, так как соответствующая координата x задана как `x[i - Ix[0]]`:

```
    for i in Ix:
        y[i] = y_1[i] + tau*V(x[i-Ix[0]]) + \
               0.5*gamma2*(y_1[i-1] - 2*y_1[i] + y_1[i+1]) + \
               0.5*tau2*f(x[i-Ix[0]], t[0])
```

Осталось обновить решение в мнимых узлах, т.е. `y[0]` и `y[-1]` (или `y[N+2]`). Для граничного условия Неймана $\frac{\partial u}{\partial x} = 0$, значения в мнимых узлах должны быть равны значениям в соответствующих внутренних узлах. Ниже приведен соответствующий фрагмент кода:

```
        # Мнимые значения устанавливаем в соответствии с du/dx=0
        i = Ix[0]
        y[i-1] = y[i+1]
        i = Ix[-1]
        y[i+1] = y[i-1]
```

Решение, график которого будем строить — срез `y[1:-1]` или `y[Ix[0]:Ix[-1]+1]`. Этот срез будет возвращать функция `solver`. Полностью программную реализацию этого подхода можно найти в файле wave1d_n_ghost.py.

Предупреждение.

Необходимо быть аккуратным с тем, как хранить сетки по пространству и времени. Пусть `x` — физические узлы

```
x = linspace(0, l, N+1)
```

«Стандартная реализация» начальных данных

```
for i in Ix:
    y_1[i] = I(x[i])
```

становится в этом случае ошибочной, так как `y_1` и `x` имеют разные длины и индекс `i` соответствует двум различным узлам сетки. На самом деле, `x[i]` соответствует `y_1[i+1]`. Правильная реализация имеет вид

```
for i in Ix:
    y_1[i] = I(x[i - Ix[0]])
```

Аналогично, использование при вычислении правой части выражения `f(x[i], t[n])` неправильно, если `x` определено на множестве физических точек. Следовательно, `x[i]` нужно заменить на `x[i - Ix[0]]`.

Альтернативный способ решения этой проблемы — задать массив `x` так, чтобы он содержал мнимые точки и `y[i]` было значением в `x[i]`.

Мнимые ячейки добавляются только к границам, на которых заданы условия Неймана. Предположим, что на $x = l$ задано условие Дирихле, а на $x = 0$ — условие Неймана. В этом случае к сетке добавляется одна мнимая ячейка $[-h, 0]$, поэтому множестве индексов для физических узлов — $\{1, 2, \ldots, N+1\}$. Ниже представлен соответствующий фрагмент кода:

```
y = zeros(N+2)
Ix = range(1, y.shape[0])
...
for i in Ix[:-1]:
    y[i] = 2*y_1[i] - y_2[i] + \
    gamma2*(y_1[i-1] - 2y[i] + y[i+1]) + \
    tau2*f(x[i-Ix[0]], t[n])
i = Ix[-1]
y[i] = ur   # условия Дирихле
i = Ix[0]
y[i-1] = y[i+1]   # условие Неймана
```

Физическое решение, график которого будет строится, — `y[1:]` или `y[Ix[0]:Ix[-1]+1]`.

7. Обобщения: переменная скорость распространения волны

Следующее обобщение одномерного волнового уравнения (1) или (12) — введение переменной скорости распространения волны $c = c(x)$. Такое уравнение описывает процесс протекающий в областях состоящих из сред с разными физическими свойствами. Когда среды отличаются физическими свойствами, такими как плотность или пористость, скорость распространения волны в этом случае зависит от положения в пространстве.

7.1. Модельное уравнение с переменными коэффициентами

Вместо $c^2(x)$ будем использовать более удобное обозначение $k(x) = c^2(x)$ для коэффициента уравнения. Одномерное волновое уравнение с переменной скоростью распространение волны принимает вид:

$$\frac{\partial^2 u}{\partial t^2} = \frac{\partial}{\partial x}\left(k(x)\frac{\partial u}{\partial x}\right) + f(x,t). \tag{31}$$

7.2. Аппроксимация переменных коэффициентов

В случае достаточно гладких коэффициентов и решения дифференциальный оператор $\frac{\partial}{\partial x}\left(k(x)\frac{\partial u}{\partial x}\right)$ во внутренних узлах сетки будем аппроксимировать разностным оператором $(ay_{\bar{x}})_x$. Для аппроксимации со вторым порядком необходимо выбрать коэффициенты разностного оператора так, чтобы

$$\frac{a_{i+1} - a_i}{h} = k'(x_i) + O(h^2), \tag{32}$$

$$\frac{a_{i+1} + a_i}{2} = k(x_i) + O(h^2). \tag{33}$$

Этим условиям удовлетворяют, в частности, следующие формулы для определения a_i:

$$a_i = k_{i-1/2} = k(x_i - 0.5h), \tag{34}$$

$$a_i = \frac{k_{i-1} + k_i}{2}, \tag{35}$$

$$a_i = 2\left(\frac{1}{k_{i-1}} + \frac{1}{k_i}\right)^{-1}. \tag{36}$$

Выражение (35) — среднее арифметическое значений коэффициента в соседних узлах и часто используется для гладких коэффициентов, среднее гармоническое (36) часто используется при аппроксимации коэффициентов с сильно меняющимися значениями.

Правую часть $f(x,t)$ уравнения (31) аппроксимируем следующим образом

$$\varphi_i^n = f(x_i, t_n).$$

Таким образом, мы можем аппроксимировать уравнение (31) на сетке $\omega_{h\tau}$ следующей разностной схемой:

$$y_{\bar{t}t} = (a y_{\bar{x}})_x + \varphi, \quad (x,t) \in \omega_{h\tau}, \tag{37}$$

Осталось выразить из уравнения (37) значение y_i^{n+1}:

$$\begin{aligned} y_i^{n+1} =& 2y_i^n - y_i^{n-1} \\ &+ \frac{\tau^2}{h^2}\left(a_{i+1}(y_{i+1}^n - y_i^n) - a_i(y_i^n - y_{i-1}^n)\right) \\ &+ \tau^2 \varphi_i^n \end{aligned} \tag{38}$$

7.3. Условия Неймана и переменные коэффициенты

Рассмотрим аппроксимацию условий Неймана на границе $x = l = Nh$:

$$\frac{y_{N+1}^n - y_{N-1}^n}{2h} = 0 \Rightarrow y_{N+1}^n = y_{N-1}^n.$$

Записывая разностную схему (38) в узле $i = N$ и учитывая, что $y_{N+1} = y_{N-1}$, получим

$$\begin{aligned} y_N^{n+1} &= 2y_N^n - y_N^{n-1} \\ &\quad + \frac{\tau^2}{h^2}\left(a_{N+1}(y_{N+1}^n - y_N^n) - a_N(y_N^n - y_{N-1}^n)\right) + \tau^2 \varphi_N^n \\ &= 2y_N^n - y_N^{n-1} + \frac{\tau^2}{h^2}\left((a_{N+1} + a_N)(y_{N-1} - y_N)\right) + \tau^2 \varphi_i^n \\ &\approx 2y_N^n - y_N^{n-1} + 2\frac{\tau^2}{h^2}\left(a_{N+1/2}(y_{N-1} - y_N)\right) + \tau^2 \varphi_N^n. \end{aligned} \tag{39}$$

Здесь мы использовали условия (32)–(33) и $a_{N+1/2} = k(x_N)$. Кроме того вместо $a_{N+1/2}$ можно использовать a_N.

Выражение (39) с a_N вместо $a_{N+1/2}$ можно записать в форме

$$a_N y_{x,N}^n + \frac{h}{2} y_{\bar{t}t,N}^n = \frac{h}{2}\varphi_N^n$$

Отметим, что подобные аппроксимации переменных коэффициентов и граничных условий Неймана мы можем получить, используя интегро-интерполяционный метод (метод баланса) или метод конечных элементов.

7.4. Более общее уравнение с переменными коэффициенты

Иногда волновое уравнение содержит переменный коэффициент перед второй производной по времени:

$$\varrho(x)\frac{\partial^2 u}{\partial t^2} = \frac{\partial}{\partial x}\left(k(x)\frac{\partial u}{\partial x}\right) + f(x,t). \tag{40}$$

Такое уравнение описывает, например, упругие волны в стержне с переменной плотностью.

Естественная аппроксимация (40) может выглядеть следующим образом:

$$\varrho y_{\bar{t}t} = \left(a y_{\bar{x}}\right)_x = \varphi, \quad (x,t) \in \omega_{h\tau} \tag{41}$$

Очевидно, что коэффициент ϱ не добавляет особых трудностей, так не требует какогло-либо осреднения, а вычисляется в узле сетки.

8. Обобщения: затухания

Существует два механизма исчезновения волн. В двумерном и трехмерном случаях энергия волн распределяется в пространстве, и, следовательно, с учетом сохранения энергии приходим к тому, что должна уменьшаться амплитуда колебаний. Этот эффект отсутствует в одномерном случае. Затухания являются второй причиной уменьшения амплитуды. Например, колебания струны исчезают из-за затуханий, обусловленных сопротивлением воздуха и неупругих эффектов в струне.

Простейший способ ввести затухания в модель заключается в добавлении первой производной по времени в уравнение:

$$\frac{\partial^2 u}{\partial t^2} + b\frac{\partial u}{\partial t} = c^2\frac{\partial^2 u}{\partial x^2} + f(x,t), \tag{42}$$

где $b \geq 0$ заданный коэффициент затухания.

Естественную аппроксимацию уравнения (42) можно получить, используя центральную разностную производную

$$y_{\bar{t}t} + b y_{\mathring{t}} = c^2 y_{\bar{x}x} + \varphi. \tag{43}$$

Решая уравнение (43) относительно y_i^{n+1}, получим

$$y_i^{n+1} = (1 + 0.5b\tau)^{-1}\left((0.5b\tau - 1)y_i^{n-1} + 2y_i^n + \gamma^2(y_{i+1}^n - 2y_i^n + y_{i-1}^n) + \varphi_i^n\right), \tag{44}$$

для $i \in \mathcal{I}_x^i$ и $n \geq 1$. Также нужно получить уравнения для y_i^1 и для граничных узлов в случае условий Неймана.

Обычно во многих физических процессах затухания являются малыми и видны только на больших временных отрезках. Этот факт делает обоснованным использование для большого числа приложений стандартного волнового уравнения без затуханий.

9. Разработка общего солвера для одномерного волнового уравнения

Программа wave1d_dn_vc.py представляет собой довольно общий код для задач одномерного распространения волн, который нацелен на следующую начально-краевую задачу:

$$\frac{\partial^2 u}{\partial t^2} = \frac{\partial}{\partial x}\left(c^2(x)\frac{\partial u}{\partial x}\right) + f(x,t), \qquad x \in (0,l),\ t \in (0,T], \tag{45}$$

$$u(x,0) = I(x), \qquad x \in [0,l], \tag{46}$$

$$\frac{\partial u(x,0)}{\partial t} = V(x), \qquad x \in [0,l], \tag{47}$$

$$u(0,t) = U_0(t) \text{ или } \frac{\partial u(0,t)}{\partial x} = 0, \qquad t \in (0,T], \tag{48}$$

$$u(0,t) = U_l(t) \text{ или } \frac{\partial u(l,t)}{\partial x} = 0, \qquad t \in (0,T]. \tag{49}$$

Единственная особенность данной задачи, в сравнении с предыдущей, состоит в том, что заданы неоднородные условия Дирихле (зависящие от времени). Реализация этого тривиальная

```
i = Ix[0]
y[i] = ul(t[n+1])

i = Ix[-1]
y[i] = ur(t[n+1])
```

Функция `solver` естественным образом расширяет простейшую функцию `solver` из сценария wave1d_1.py, добавлением реализации условий Неймана, зависящих от времени условий Дирихле, а также переменной скорости распространения волны. Различные сегменты кода уже были рассмотрены выше. Саму реализацию функции `solver` с комментариями можно проанализировать в сценарии wave1d_dn_vc.py.

Векторизация используется только внутри цикла по времени, а не в для начальных условиях, так как эта начальная работа незначительна по сравнению с моделированием больших временных отрезков в одномерном случае.

Ниже даются пояснения более продвинутых подходов, применяемых в общем одномерном солвере.

9.1. Реализация функции действий полозователя в виде класса

Полезная особенность сценария wave1d_dn_vc.py — это реализация функции действия пользователя в виде класса. Эта часть сценария может потребовать некоторых пояснений.

Код. Класс для построения графиков, очистки файлов, создания анимированных графиков, которые выполнялись в функции `wave1d_1.viz`, можно реализовать следующим образом:

```
class PlotAndStoreSolution:
    """
    Класс для функции user_action в solver.
    Только визуализация решения.
    """
    def __init__(
            self,
            casename='tmp',     # Префикс в именах файлов
            umin=-1, umax=1,    # Задаются границы по оси y
            pause_between_frames=None,  # Скорость видео
            backend='matplotlib',       # или 'gnuplot' или None
            screen_movie=True, # Показывать видео на экране?
            title='',          # Дополнительное сообщение в title
            skip_frame=1,      # Пропуск каждого skip_frame кадра
            filename=None):    # Имя файла с решением
        self.casename = casename
        self.yaxis = [umin, umax]
        self.pause = pause_between_frames
        self.backend = backend
        if backend is None:
            # Использовать matplotlib
            import matplotlib.pyplot as plt
        elif backend in ('matplotlib', 'gnuplot'):
            module = 'scitools.easyviz.' + backend + '_'
            exec('import %s as plt' % module)
        self.plt = plt
        self.screen_movie = screen_movie
        self.title = title
        self.skip_frame = skip_frame
        self.filename = filename
        if filename is not None:
            # Сохранение временной сетки, когда у записывается в файл
            self.t = []
            filenames = glob.glob('.' + self.filename + '*.dat.npz')
            for filename in filenames:
                os.remove(filename)

        # Очистка старых кадров
        for filename in glob.glob('frame_*.png'):
            os.remove(filename)

    def __call__(self, u, x, t, n):
        """
        Функция обратного вызова user_action, вызываемая солвером:
        сохранение решения, построение графиков на экране и
        и сохранение их в файл.
        """
        # Сохраняем решение и в файл, используя numpy.savez
        if self.filename is not None:
            name = 'u%04d' % n  # имя массива
            kwargs = {name: u}
            fname = '.' + self.filename + '_' + name + '.dat'
```

```
            np.savez(fname, **kwargs)
            self.t.append(t[n])  # сохранение соответствующего временного знаяения
            if n == 0:           # сохранение массива x один раз
                np.savez('.' + self.filename + '_x.dat', x=x)

        # Анимация
        if n % self.skip_frame != 0:
            return
        title = 't=%.3f' % t[n]
        if self.title:
            title = self.title + ' ' + title
        if self.backend is None:
            # анимация matplotlib
            if n == 0:
                self.plt.ion()
                self.lines = self.plt.plot(x, u, 'r-')
                self.plt.axis([x[0], x[-1],
                               self.yaxis[0], self.yaxis[1]])
                self.plt.xlabel('x')
                self.plt.ylabel('u')
                self.plt.title(title)
                self.plt.legend(['t=%.3f' % t[n]])
            else:
                # Обновляем решение
                self.lines[0].set_ydata(u)
                self.plt.legend(['t=%.3f' % t[n]])
                self.plt.draw()
        else:
            # анимация scitools.easyviz
            self.plt.plot(x, u, 'r-',
                          xlabel='x', ylabel='u',
                          axis=[x[0], x[-1],
                                self.yaxis[0], self.yaxis[1]],
                          title=title,
                          show=self.screen_movie)
        # пауза
        if t[n] == 0:
            time.sleep(2)  # показываем начальное решение 2 с
        else:
            if self.pause is None:
                pause = 0.2 if u.size < 100 else 0
            time.sleep(pause)

        self.plt.savefig('frame_%04d.png' % (n))

    def make_movie_file(self):
        """
        Создается подкаталог на основе casename, перемещаем все файлы
        с кадрами в этот каталог и создаем файл index.html для показа
        видео в браузере (как последовательности PNG файлов).
        """
        directory = self.casename
        if os.path.isdir(directory):
            shutil.rmtree(directory)   # rm -rf directory
        os.mkdir(directory)            # mkdir directory
        # mv frame_*.png directory
        for filename in glob.glob('frame_*.png'):
```

```
            os.rename(filename, os.path.join(directory, filename))
        os.chdir(directory)        # cd directory
        fps = 4 # frames per second
        if self.backend is not None:
            from scitools.std import movie
            movie('frame_*.png', encoder='html',
                  output_file='index.html', fps=fps)

        # Создаем другие видео форматы: Flash, Webm, Ogg, MP4
        codec2ext = dict(flv='flv', libx264='mp4', libvpx='webm',
                         libtheora='ogg')
        filespec = 'frame_%04d.png'
        movie_program = 'avconv' # или 'ffmpeg'
        for codec in codec2ext:
            ext = codec2ext[codec]
            cmd = '%(movie_program)s -r %(fps)d -i %(filespec)s '\
                  '-vcodec %(codec)s movie.%(ext)s' % vars()
            os.system(cmd)
        os.chdir(os.pardir)  # возвращаемся в родительский каталог

    def close_file(self, hashed_input):
        """
        Сливаем все файлы в один архив.
        hashed_input --- строка, отражающая входные данные
        для моделирования (создана функцией solver).
        """
        if self.filename is not None:
            np.savez('.' + self.filename + '_t.dat',
                     t=np.array(self.t, dtype=float))

            archive_name = '.' + hashed_input + '_archive.npz'
            filenames = glob.glob('.' + self.filename + '*.dat.npz')
            merge_zip_archives(filenames, archive_name)
        print 'Archive name:', archive_name
        # data = numpy.load(archive); data.files holds names
        # data[name] extract the array
```

Разбор. Представленный выше класс поддерживает построение графиков с помощью Matplotlib (`backend=None`) или SciTools (`backend=matplotlib` или `backend=gnuplot`).

Конструктор показывает как можно гибко импортировать графический модуль как `scitools.easyviz.gnuplot_` или `scitools.easyviz.matplotlib_` (символ подчеркивания в конце обязателен). С помощью параметра `screen_movie` мы можем подавлять вывод графиков на экран. В качестве альтернативы, для медленных анимированных графиков, связанных с мелкой расчетной сеткой, мы можем задать `skip_frame=10`, устанавливая показ каждого десятого кадра.

Метод `__call__` позволяет объектам класса `PlotAndStoreSolution` вести себя как функции, так что мы можем передавать объект, например `p`, в качестве аргумента `user_action` в функцию `solver`, и любой вызов `user_action` будет вызывать `p.__call__`. Метод `__call__` строит график

решения на экране, сохраняет график в файл, а также сохраняет решение в файл для дальнейшего использования.

9.2. Распространение импульса в двух средах

Функция `pulse` в wave1d_dn_vc.py демонстрирует движение волны в разнородных средах с переменным c. Можно задать интервал, на котором скорость распространения волны уменьшается пропорционально множителю `slowness_factor` (или увеличивается если задать этот множитель меньшим единицы).

Четыре типа начального распределения реализованы:

1. прямоугольный импульс (`plug`)
2. функция Гаусса (`gaussian`)
3. один период косинуса (`cosinehat`)
4. половина периода косинуса (`half-cosinehat`)

Эти начальные условия, имеющие форму пика, могут помещаться посередине (`loc=center`) или на левом конце (`loc=left`) расчетной области. Импульсы, расположенные посередине, распадаются на две части, каждая с амплитудой, в два раза меньшей начальной, и движутся в противоположных направлениях. Если импульс расположен на левом конце (центр импульса в $x = 0$) и задано условие Неймана, генерируется только волна бегущая вправо. Также возникает волна, движущаяся влево, но она бежит от $x = 0$ в отрицательной части оси x и не видна на отрезке $[0, l]$. Функция `pulse` является удобным инструментом для вариации с разными формами импульса и расположениями сред с разными скоростями распространения волны. Ниже представлена реализация данной функции:

```
def pulse(gamma=1,            # максимальное число Куранта
          Nx=200,         # число узлов по пространству
          animate=True,
          version='vectorized',
          T=2,            # конечное время
          loc='left',     # размещение начального условия
          pulse_tp='gaussian',  # pulse/init.cond.
          slowness_factor=2, # скорость распространения волны в правой среде
          medium=[0.7, 0.9], # отрезок правой области (среды)
          skip_frame=1,
          sigma=0.05):
    """
    Различные пико-образные начальные условия на [0,1].
    Скорость распространения волны уменьшается в slowness_factor раз
    венутри среды. Параметр loc может принимать значения 'center' или 'left',
    в зависимости от того, где располагается пик начальных условий.
    Параметр sigma определяет ширину импульса.
    """
```

```
# Используем безразмерные параметры: l=1 для длины области,
# c_0=1 для скорости распространения волны вне области.
l = 1.0
c_0 = 1.0
if loc == 'center':
    xc = l/2
elif loc == 'left':
    xc = 0

if pulse_tp in ('gaussian','Gaussian'):
    def I(x):
        return np.exp(-0.5*((x-xc)/sigma)**2)
elif pulse_tp == 'plug':
    def I(x):
        return 0 if abs(x-xc) > sigma else 1
elif pulse_tp == 'cosinehat':
    def I(x):
        # Один период косинуса
        w = 2
        a = w*sigma
        return 0.5*(1 + np.cos(np.pi*(x-xc)/a)) \
            if xc - a <= x <= xc + a else 0

elif pulse_tp == 'half-cosinehat':
    def I(x):
        # Половина периода косинуса
        w = 4
        a = w*sigma
        return np.cos(np.pi*(x-xc)/a) \
            if xc - 0.5*a <= x <= xc + 0.5*a else 0
else:
    raise ValueError(u'Ошибочный_tp="%s"' % pulse_tp)

def c(x):
    return c_0/slowness_factor \
        if medium[0] <= x <= medium[1] else c_0

umin=-0.5; umax=1.5*I(xc)
casename = '%s_Nx%s_sf%s' % \
           (pulse_tp, Nx, slowness_factor)
action = PlotMediumAndSolution(
    medium, casename=casename, umin=umin, umax=umax,
    skip_frame=skip_frame, screen_movie=animate,
    backend='matplotlib', filename='tmpdata')

# Выбор ограничения устойчивости при заданном Nx, худший случай с
# (меньший gamma будет использовать этот шаг tau, но меньшее Nx)
tau = (l/Nx)/c_0
cpu, hashed_input = solver(I=I, V=None, f=None, c=c, ul=None, ur=None,
                           l=l, tau=tau, gamma=gamma, T=T,
                           user_action=action, version=version,
                           stability_safety_factor=1)
action.make_movie_file()
action.close_file(hashed_input)
```

Используемый здесь класс `PlotMediumAndSolution` — это подкласс класса `PlotAndStoreSolution`, где среда с уменьшенным коэффициентом c, заданная параметром `medium`, отображается на графике.

Комментарий по выбору параметров дискретизации.

Параметр N_x в функции `pulse` не соответствует фактической пространственной сетке, соответствующей $\gamma < 1$, так как функция `solver` принимает фиксированные значения τ и γ и вычисляет h соответствующим образом. Как видно, в функции `pulse` заданное значение τ выбирается из условия $\gamma = 1$, поэтому, если $\gamma < 1$, γ остается таким же, а функция `solver` оперирует большим значением h и N_x меньшим, чем заданным в `pulse`. Причина этого в том, что мы хотим сохранять фиксированный шаг τ графические кадры в анимации синхронизированы по времени независимо от значения γ (т.е. h меняется, если меняется число Куранта).

Читателю предлагается "поиграть" с функцией `pulse`:

```
>>> import wave1d_dn_vc as w
>>> w.pulse(loc='left', pulse_tp='cosinehat', Nx=50, skip_frame=10)
```

Для того, чтобы легко остановить отображение графиков (например, с помощью Ctrl-C) и начать новое вычисление, может быть проще запустить предыдущие два выражения из командной строки:

Terminal

```
Terminal> python -c 'import wave1d_dn_vc as w; w.pulse(loc='left', pulse_tp='cosinehat', Nx=50, skip_frame=10
```

10. Упражнения

Упражнение 6: Нахождение аналитического решения волнового уравнения с затуханием

Рассмотрим волновое уравнение с затуханием (42). Цель упражнения — найти точное решение задачи с затуханием. Отправная точка — стоячая волна из 1. Необходимо включить множитель затухания e^{-ct}, а также синусоидальной и косинусоидальные компоненты по времени:

$$u_e(x,t) = e^{-\beta t}\sin kx\left(A\cos\omega t + B\sin\omega t\right).$$

Найти k, используя граничные условия $u(0,t) = u(l,t) = 0$. Затем используйте уравнение для нахождения ограничений на β, ω, A и B. Сформулируйте полную начально-краевую задачу и ее решение. Имя файла: `damped_waves`.

Задача 7: Анализ симметричных граничных условий

Рассмотрим простую волну-"площадку", где $\Omega = [-l, l]$ и

$$I(x) = \begin{cases} 1, & x \in [-\delta, \delta], \\ 0, \text{иначе}, \end{cases}$$

для некоторого числа $0 < \delta < l$. Второе начальное условие — $\frac{\partial u(x,0)}{\partial t} = 0$. Предположим также, что отсутствует правая часть f. В качестве граничных условий выберем однородные условия Дирихле. Решение данной задачи симметрично относительно $x = 0$. Это означает, что мы можем проводить моделирование только на половине области $[0, l]$.

1) Показать почему симметричным граничным условием будет $\frac{\partial u}{\partial x} = 0$ при $x = 0$.

2) Выполнить расчет задачи для волнового уравнения на отрезке $[-l, l]$. После этого, используйте симметричность решения и выполните расчет в половине области $[0, l]$, используя симметричное граничное условие при $x = 0$. Сравните два решения и удостоверьтесь в том, что они совпадают.

3) Докажите симметричность решения, сформулировав начально-краевую задачу и показав, что если $u(x, t)$ — решение задачи, то и $u(-x, t)$ также является решением этой задачи.

Подсказка. Симметричность функции относительно $x_0 = 0$ означает, что $f(x_0 + h) = f(x_0 - h)$.
Имя файла: `waved1d_symmetric`.

Упражнение 8: Имульс через слоистую среду

Используйте функцию `pulse` из wave1d_dn_vc.py для того, чтобы исследовать распространение импульса, чей начальный пик расположен в $x = 0$, через две среды с разными скоростями распространения волн. Скорость распространения волны в левой области положить равной 1, а в правой области — s_f. Показать, что происходит для разных импульсов при $N_x = 40, 80, 160$ и $s_f = 2, 4$. Моделирование проводить до $T = 2$. Имя файла: `pulse1d.py`.

Упражнение 9: Объяснение почему возникают численные шумы

Эксперименты из 8 показывают существенные численные шумы (осцилляции) в виде нифизичных колебаний, особенно для $s_f = 4$ и импульса площадки (`plug`) или `half-cosinehat` импульса. Шумы менее видны для Гауссового импульса. Выполните варианты расчетов для импульсов `plug` и `half-cosinehat` при $s_f = 1$, $\gamma = 0.9, 0.25$ и $N_x = 40, 80, 160$. Используйте отношение численной дисперсии для обоснования результатов. Имя файла: `pulse1d_analysis`.

Упражнение 10: Исследование гармонического среднего в одномерной модели

Гармонические средние часто используются, когда скорость распространения волны негладкая или разрывная. Даст ли меньшие шумы использование гармонического среднего при аппроксимации коэффициента уравнения для $s_f = 4$ в 9? Имя файла: `pulse1d_harmonic.py`.

Задача 11: Реализация условий открытых границ

Для того, чтобы дать возможность волне покинуть вычислительную область и перемещаться без искажений через границу $x = l$, в одномерной модели можно наложить следующее условие, называемое *условием излучения* или *условием открытой границы*:

$$\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} = 0. \tag{50}$$

Параметр c — скорость распространения волны.

Показать, что (50) допускает решение $u = g_R(x - ct)$ (волна, бегущая вправо), но не $u = g_L(x + ct)$ (волна, бегущая влево).

Соответствующее условие открытой границы при $x = 0$ имеет вид

$$\frac{\partial u}{\partial t} - c\frac{\partial u}{\partial x} = 0. \tag{51}$$

1) Естественная идея аппроксимации граничного условия (50) в конечном узле с индексом $i = N$ заключается в использовании центральных разностных производных по пространственной и временной перемменной:

$$y^n_{t,N} + cy^n_{\bar{x},N} = 0. \tag{52}$$

Избавиться от фиктивного значения y^n_{N+1}, используя аппроксимацию уравнения в этой же точке.

Уравнение для y^1_i также нарушается, мы можем использовать условие $u^1_N = 0$, так как волна еще не достигла правой границы.

2) Более удобная реализация условия открытой границы при $x = l$ может основываться на явной дискретизации

$$y^n_{t,N} + cy^n_{\bar{x},N} = 0. \tag{53}$$

Из этого уравнения можно найти y^{n+1}_N и его как условие Дирихле. Однако, в этом случае аппроксимация будет иметь первый порядок.

Реализовать эту схему для волнового уравнения $\frac{\partial^2 u}{\partial t^2} = c^2\frac{\partial^2 u}{\partial x^2}$ на отрезке $[0, l]$ с граничным условием $\frac{\partial u}{\partial x} = 0$ при $x = 0$, условием (50) при $x = l$, и начальным возмущением в центре отрезка, например, профиль-площадка вида

$$u(x, 0) = \begin{cases} 1, & l/2 - \ell \le x \le l/2 + \ell, \\ 0, & \text{в противном случае.} \end{cases}$$

Обратите внимание на то, что начальный профиль распадается на две площадки, волна, бегущая влево, отражается от границы $x = 0$, и обе волны уходят за границу $x = l$, оставляя в итоге решение $u = 0$ на $[0, l]$. Используйте число Куранта равное единице, что позволит получить точное решение. Создать анимированный график.

3) Добавить возможность задание либо условия $\frac{\partial u}{\partial x} = 0$, либо условия открытой границы при $x = 0$. Последнее условие аппроксимировать как

$$y^n_{t,0} - cy^n_{x,i} = 0,$$

дающее возможность явного вычисления y^{n+1}_0.

Реализацию можно протестировать на функции Гаусса как начальном условии:

$$g(x; m, s) = \frac{1}{\sqrt{2\pi}s} e^{-\frac{(x-m)^2}{2s^2}}.$$

Выполнить два теста:

1. Возмущение в центре области, $I(x) = g(x; l/2, s)$ и открытой левой границей.

1. Возмущение на левом конце области, $I(x) = g(x; 0, s)$ и условием симметрии $\frac{\partial u}{\partial x} = 0$ на левой границе.

Создать юнит-тесты для обоих случаев, проверяющие, что решение равно нулю после того, как волна покинет область.

Подсказка. Модифицируйте солвер из wave1d_n.py.
Имя файла: `wave1d_open_bc.py`.

Предметный указатель