Численные методы решения уравнений в частных производных

С.В. Лемешевский

07 April 2016

Contents

Курс посвящен использованию языка Python при реализации численных методов. Рассматриваются вопросы численного решения задач для обыкновенных дифференциальных уравнений, а также уравнений в частных производных. При программной реализации численных алгоритмов используются сециализированные математические пакеты.

Предупреждение: Ниже даны ссылки на материалы, которые рассматриваются на лекциях. Дана также ссылка на PDF версию курса. Все материалы созданы с помощью системы Sphinx. PDF файл генерируется с использованием LaTeX, а в HTML для отображения формул используется MathJax. В связи с этим возможны некоторые проблемы при отображении формул, а также возможны ошибки в тексте, поэтому сообщайте мне об этом, пожалуйста.

Chapter 1

Конечно-разностные схемы для обыкновенных дифференциальных уравнений

Колебательные процессы описываются дифференциальными уравнениями, решения которых представляют собой изменяющуюся со временем синусоиду. Такие решения предъявляют некоторые дополнительные (по сравнению с монотонными и очень гладкими решениями) требования к вычислительному алгоритму. Как частота, так и амплитуда колебаний должны достаточно точно воспроизводится численным методом решения. Большинство представленных в данном разделе подходов могут использоваться для построения численных методов решения уравнений в частных производных с решениями колебательного типа в многомерном случае.

Содержание:

1.1 Конечно-разностная дискретизация

Многие вычислительные проблемы, возникающие при вычислении осциллирующих решений обыкновенных дифференциальных уравнений и уравнений в частных производных могут быть проиллюстрированы на простейшем ОДУ второго порядка $u'' + \omega^2 u = 0$.

1.1.1 Базовая модель колебательного процесса

Колебательная система без затуханий и внешних сил может быть описана начальной задачей для ОДУ второго порядка

$$u'' + \omega^2 u = 0, \quad t \in (0, T], \tag{1.1}$$

$$u(0) = U, \quad u'(0) = 0. \tag{1.2}$$

Здесь ω и U — заданные постоянные. Точное решение задачи (1.1) – (1.2) имеет вид

$$u(t) = U \cos \omega t, \tag{1.3}$$

т.е. u описывает колебания с постоянной амплитудой U и угловой частотой ω. Соответствующий период колебаний равен $P = 2\pi/\omega$. Число периодов в секунду — это $f = \omega/2\pi$. Оно измеряется в герцах (Гц). Как f, так и ω описываются частоту колебаний, но ω более точно называется *угловой частотой* и измеряется в радиан/с.

В колебательных механических системах, описываемых задачей (1.1) – (1.2) u часто представляет собой координату или смещение точки в системе. Производная $u'(t)$, таким образом, интерпретируется как скорость, а $u''(t)$ — ускорение. Задача (1.1) – (1.2) описывает не только механические колебания, но и колебания в электрических цепях.

1.1.2 Разностная схема

При численном решении задачи (1.1) – (1.2) будем использовать равномерную сетку по переменной t с шагом τ:

$$\omega_\tau = \{t_n = n\tau,\ n = 0, 1, \dots N\}.$$

Приближенное решение задачи (1.1) – (1.2) в точке t_n обозначим y^n.

Простейшая разностная схема для приближенного решения задачи (1.1) – (1.2) есть

$$\frac{y^{n+1} - 2y^n + y^{n-1}}{\tau^2} = -\omega^2 y^n. \tag{1.4}$$

Кроме того необходимо аппроксимировать производную во втором начальном условии. Будем аппроксимировать ее центральную разностную производную:

$$\frac{y^1 - y^{-1}}{2\tau} = 0. \tag{1.5}$$

Для формулировки вычислительного алгоритма, предположим, что мы уже знаем значение y^{n-1} и y^n. Тогда из (1.4) мы можем выразить неизвестное значение y^{n+1}:

$$y^{n+1} = 2y^n - y^{n-1} - \tau^2\omega^2 y^n. \tag{1.6}$$

Вычислительный алгоритм заключается в последовательном применении для $n = 1, 2, \dots$.

Очевидно, что (1.6) нельзя использовать при $n = 0$, так как для вычисления y^1 необходимо знать неопределенное значение y^{-1} при $t = -\tau$. Однако, из (1.5) имеем $y^{-1} = y^1$. Подставляя последнее в (1.6) при $n = 0$, получим

$$y^1 = 2y^0 - y^1 - \tau^2\omega^2 y^0,$$

откуда

$$y^1 = y^0 - \frac{1}{2}\tau^2\omega^2 y^0. \tag{1.7}$$

В *упражнении 1* требуется использовать альтернативный способ вывода (1.7), а также построить аппроксимацию начального условия $u'(0) = V \neq 0$.

1.1.3 Вычислительный алгоритм

Для решения задачи (1.1) – (1.2) следует выполнить следующие шаги:

1. $y^0 = U$
2. вычисляем y^1, используя (1.7)
3. для $n = 1, 2, \ldots,$
   (a) вычисляем y^n, используя (1.6)

Более строго вычислительный алгоритм напишем на Python:

```
t = linspace(0, T, N+1)  # сетка по времени
tau = t[1] - t[0]        # постоянный временной шаг
u = zeros(N+1)           # решение

u[0] = U
u[1] = u[0] - 0.5*tau**2*omega**2*u[0]
for n in range(1, N):
    u[n+1] = 2*u[n] - u[n-1] - tau**2*omega**2*u[n]
```

1.1.4 Безындексные обозначения

Разностную схему можно записать, используя безындексные обозначения. Для *левой* и *правой разностных производных* соответственно имеем

$$y_{\bar{t}} \equiv \frac{y^n - y^{n-1}}{\tau}, \quad y_t \equiv \frac{y^{n+1} - y^n}{\tau}.$$

Для *второй разностной производной* получим

$$y_{\bar{t}t} = \frac{y_t - y_{\bar{t}}}{\tau} = \frac{y^{n+1} - 2y^n + y^{n-1}}{\tau^2}.$$

Для аппроксимации второго начального условия использовалась *центральная разностная производная*:

$$y_{\mathring{t}} = \frac{y^{n+1} - y^{n-1}}{2\tau}$$

1.2 Программная реализация

1.2.1 Функция-решатель (Солвер)

Алгоритм построенный в предыдущем разделе легко записать как функцию Python, вычисляющую $y^0, y^1, \ldots, y^N$ по заданным входным параметрам U, ω, τ и T:

```
import numpy as np
import matplotlib.pyplot as plt

def solver(U, omega, tau, T):
    """
    Решается задача
    u'' + omega**2*u = 0 для t из (0,T], u(0)=U и u'(0)=0,
```

```
    конечно-разностным методом с постоянным шагом tau
    """
    tau = float(tau)
    Nt = int(round(T/tau))
    u = np.zeros(Nt+1)
    t = np.linspace(0, Nt*tau, Nt+1)

    u[0] = U
    u[1] = u[0] - 0.5*tau**2*omega**2*u[0]
    for n in range(1, Nt):
        u[n+1] = 2*u[n] - u[n-1] - tau**2*omega**2*u[n]
    return u, t
```

Также будет удобно реализовать функцию для построения графиков точного и приближенного решений:

```
def u_exact(t, U, omega):
    return U*np.cos(omega*t)

def visualize(u, t, U, omega):
    plt.plot(t, u, 'r--o')
    t_fine = np.linspace(0, t[-1], 1001)  # мелкая сетка для точного решения
    u_e = u_exact(t_fine, U, omega)
    plt.hold('on')
    plt.plot(t_fine, u_e, 'b-')
    plt.legend([u'приближенное', u'точное'], loc='upper left')
    plt.xlabel('$t$')
    plt.ylabel('$u$')
    tau = t[1] - t[0]
    plt.title('$\\tau = $ %g' % tau)
    umin = 1.2*u.min();  umax = -umin
    plt.axis([t[0], t[-1], umin, umax])
    plt.savefig('tmp1.png');  plt.savefig('tmp1.pdf')
```

Соответствующая основная программа вызывающая эти функции для моделирования заданного числа периодов (`num_periods`) может иметь вид

```
U = 1
omega = 2*pi
tau = 0.05
num_periods = 5
P = 2*np.pi/tau    #  один период
T = P*num_periods
u, t = solver(U, omega, tau, T)
visualize(u, t, U, omega, tau)
```

Задание некоторых входных параметров удобно осуществлять через командную строку. Ниже представлен фрагмент кода, использующий инструмент `ArgumentParser` из модуля `argparse` для определения пар "параметр значение" (`--option value`) в командной строке:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('--U', type=float, default=1.0)
parser.add_argument('--omega', type=float, default=2*np.pi)
parser.add_argument('--tau', type=float, default=0.05)
parser.add_argument('--num_periods', type=int, default=5)
a = parser.parse_args()
U, omega, tau, num_periods = a.U, a.omega, a.tau, a.num_periods
```

Стандартный вызов основной программы выглядит следующим образом:

```
Terminal> python vib_undamped.py --num_periods 20 --tau 0.1
```

Вычисление производной $u'(t)$

В приложениях часто необходимо анализировать поведение скорости $u'(t)$. Приближенно найти ее по полученным в узлах сетки ω_τ значениям y можно, например, используя центральную разностную производную:

$$u'(t_n) \approx v^n = \frac{y^{n+1} - y^n}{2\tau} = y^n_{\mathring{t}}. \tag{1.8}$$

Эта формула используется во внутренних узлах сетки ω_τ при $n = 1, 2, \ldots, N-1$. Для $n = 0$ скорость v^0 задана начальным условием, а для $n = N$ мы можем использовать направленную (левую) разностную производную $v^N = y^N_{\bar{t}}$.

Для вычисления производной можно использовать следующий (скалярный) код:

```
v = np.zeros_like(u)  # or v = np.zeros(len(u))
# Используем центральную разностную производную во внутренних узлах
for i in range(1, len(u)-1):
     v[i] = (u[i+1] - u[i-1])/(2*tau)
# Используем начальное условие для u'(0)
v[0] = 0
# Используем левую разностную производную
v[-1] = (u[-1] - u[-2])/tau
```

Мы можем избавиться от цикла (медленного для больших N), векторизовав вычисление разностной производной. Фрагмент кода, приведенного выше, можно заменить следующей векторизованной формой:

```
v = np.zeros_like(u)
v[1:-1] = (u[2:] - u[:-2])/(2*tau)  # центральная разностная производная
v[0] = 0                            # начальное условие u'(0)
v[-1] = (u[-1] - u[-2])/tau         # левая разностная производная
```

1.2.2 Верификация реализации алгоритма

Вычисления в ручную

Простейший способ проверки правильности реализации алгоритма заключается в вычислении значений y^1, y^2 и y^3, например с помощью калькулятора и в написании функции, сравнивающей эти результаты с соответствующими результатами вычисленными с помощью функции `solver`. Представленная ниже функция `test_three_steps` демонстрирует, как можно использовать "ручные" вычисления для тестирования кода:

```
def test_three_steps():
    from math import pi
    U = 1;  omega = 2*pi;  tau = 0.1;  T = 1
    u_by_hand = np.array([
            1.000000000000000,
```

```
            0.802607911978213,
            0.288358920740053])
    u, t = solver(U, omega, tau, T)
    diff = np.abs(u_by_hand - u[:3]).max()
    tol = 1E-14
    assert diff < tol
```

Тестирование на простейших решениях

Построение тестовой задачи, решением которой является постоянная величина или линейная функция, помогает выполнять начальную отладку и проверку реализации алгоритма, так как соответствующие вычислительные алгоритмы воспроизводят такие решения с машинной точностью. Например, методы второго порядка точности часто являются точными на полиномах второй степени. Возьмем точное значение второй разностной производной $(t^2)^n_{\overline{t}t} = 2$. Решение $u(t) = t^2$ дает $u'' + \omega^2 u = 2 + (\omega t)^2 \neq 0$. Следовательно, необходимо добавить функцию источника в уравнение: $u'' + \omega^2 u = f$. Такое уравнение имеет решение $u(t) = t^2$ при $f(t) = (\omega t)^2$. Простой подстановкой убеждаемся, что сеточная функция $y^n = t_n^2$ является решением разностной схемы. Выполните *задачу 1*.

Анализ скорости сходимости

Естественно ожидать, что погрешность метода ε должна уменьшаться с уменьшением шага τ. Многие вычислительные методы (в том числе и конечно-разностные) имеют степенную зависимость погрешности ε от τ:

$$\varepsilon = M\tau^r, \tag{1.9}$$

где C и r — постоянные (обычно неизвестные), не зависящие от τ. Формула (1.9) является асимптотическим законом, верным при достаточно малом параметре τ. Насколько малом оценить сложно без численной оценки параметра r.

Параметр r называется *скоростью сходимости*.

Оценка скорости сходимости

Чтобы оценить скорость сходимости для рассматриваемой задачи, нужно выполнить

- провести m расчетов, уменьшая на каждом из них шаг в два раза: $\tau_k = 2^{-k}\tau_0$, $k = 0, 1, \ldots, m-1$,
- вычислить L_2 -норму погрешности для каждого расчета $\varepsilon_k = \sqrt{\sum_{n=0}^{N-1}(y^n - u_e(t_n))\tau_k}$,
- оценить скорость сходимости на основе двух последовательных экспериментов $(\tau_{k-1}, \varepsilon_{k-1})$ и (τ_k, ε_k), в предположении, что погрешность подчинена закону (1.9). Разделив $\varepsilon_{k-1} = M\tau_{k-1}^r$ на $\varepsilon_k = M\tau_k^r$ и решая получившееся уравнение относительно r, получим

$$r_{k-1} = \frac{\ln(\varepsilon_{k-1}/\varepsilon_k)}{\ln(\tau_{k-1}/\tau_k)}, \quad k = 0, 1, \ldots, m-1.$$

Будем надеяться, что полученные значения $r_0, r_1, \ldots, r_{m-2}$ сходятся к некоторому числу (в нашем случае к 2).

Программная реализация

Ниже приведена функция для вычисления последовательности $r_0, r_1, \ldots, r_{m-2}$.

```
def convergence_rates(m, solver_function, num_periods=8):
    """
    Возвращает m-1 эмпирическую оценку скорости сходимости,
    полученную на основе m расчетов, для каждого из которых
    шаг по времени уменьшается в два раза.
    solver_function(U, omega, tau, T) решает каждую задачу,
    для которой T, получается на основе вычислений для
    num_periods периодов.
    """
    from math import pi
    omega = 0.35; U = 0.3       # просто заданные значения
    P = 2*pi/omega              # период
    tau = P/30                  # 30 шагов на период 2*pi/omega
    T = P*num_periods

    tau_values = []
    E_values = []
    for i in range(m):
        u, t = solver_function(U, omega, tau, T)
        u_e = u_exact(t, U, omega)
        E = np.sqrt(tau*np.sum((u_e-u)**2))
        tau_values.append(tau)
        E_values.append(E)
        tau = tau/2

    r = [np.log(E_values[i-1]/E_values[i])/
         np.log(tau_values[i-1]/tau_values[i])
         for i in range(1, m, 1)]
    return r
```

Ожидаемая скорость сходимости — 2, так как мы используем конечно-разностную аппроксимации второго порядка для второй производной в уравнении и для первого начального условия. Теоретический анализ погрешности аппроксимации дает $r = 2$.

Для рассматриваемой задачи, когда τ_0 соответствует 30 временным шагам на период, возвращаемый список `r` содержит элементы равные `2.00`. Это означает, что все значения τ_k удовлетворяют ассимтотическому режиму, при котором выполнено соотношение (1.9).

Теперь мы можем написать тестовую функцию, которая вычисляет скорости сходимости и проверяет, что последняя оценка достаточно близка к 2. Здесь достаточна граница допуска 0.1.

```
def test_convergence_rates():
    r = convergence_rates(m=5, solver_function=solver, num_periods=8)
    tol = 0.1
    assert abs(r[-1] - 2.0) < tol
```

Безразмерная модель

При моделировании полезно использовать безразмерные переменные, так как в этом случае нужно задавать меньше параметров. Рассматриваемая нами задача обезразмеривается заданием переменных $\bar{t} = t/t_c$ и $\bar{u} = u/u_c$, где t_c и u_c характерные масштабы для t и u,

соответственно. Задача для ОДУ принимает вид

$$\frac{u_c}{t_c}\frac{d^2\bar{u}}{d\bar{t}^2} + u_c\bar{u} = 0, \quad u_c\bar{u}(0) = U, \quad \frac{u_c}{t_c}\frac{d\bar{u}}{d\bar{t}}(0) = 0.$$

Обычно в качестве t_c выбирается один период колебаний, т.е. $t_c = 2\pi/\omega$ и $u_c = U$. Отсюда получаем безразмерную модель

$$\frac{d^2\bar{u}}{d\bar{t}^2} + 4\pi^2\bar{u} = 0, \quad \bar{u}(0) = 1, \quad \bar{u}'(0) = 0. \tag{1.10}$$

Заметьте, что в (1.10) отсутствуют физические параметры. Таким образом мы можем выполнить одно вычисление $\bar{u}(\bar{t})$ и затем восстановить любое $u(t;\omega,U)$ следующим образом

$$u(t;\omega,U) = u_c\bar{u}(t/t_c) = U\bar{u}(\omega t/(2\pi)).$$

Расчет для безразмерной модели можно выполнить вызвав функцию `solver(U = 1, omega = 2*np.pi, tau, T)`. В этом случае период равен 1 и `T` задает количество периодов. Выбор `tau = 1./N` дает N шагов на период.

Сценарий `vib_undamped.py` содержит представленные в данном разделе примеры.

1.3 Проведение вычислительного эксперимента

На рисунке 1.1 представлено сравнение точного и приближенного решений безразмерной модели (impl-3) с шагами $\tau = 0.1$ и 0.5. Проанализировав графики, мы можем сделать следующие предположения:

- Похоже, что численное решение корректно передает амплитуду колебаний
- Наблюдается погрешность при расчете угловой частоты, которая уменьшается при уменьшении шага.
- Суммарная погрешность угловой частоты увеличивается со временем.

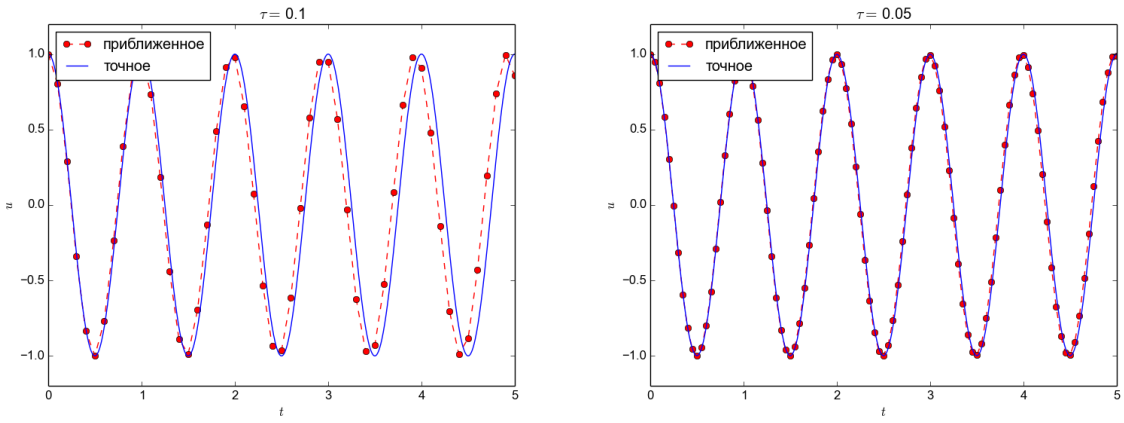


Figure1.1.: Эффект от уменьшения шага вдвое

1.3.1 Использование изменяющихся графиков

В рассматриваемой нами задаче о колебаниях следует анализировать поведение системы на больших временных интервалах. Как видно из предыдущих наблюдений погрешность

угловой частоты накапливается и становится более различимой со временем. Мы можем провести анализ на большом интервале времени, построив подвижные графики, которые могут изменяться в течение p новых вычисленных периодах решения. Пакет SciTools содержит удобный инструмент для этого: `MovingPlotWindow`. Ниже приведена функция, использующая данный инструмент:

```
def visualize_front(u, t, U, omega, savefig=False, skip_frames=1):
    """
    Строится зависимость приближенного и точного решений
    от t с использованием анимированного изображения и непрерывного
    отображения кривых, изменяющихся со временем.
    Графики сохраняются в файлы, если параметр savefig=True.
    Только каждый skip_frames-й график сохраняется (например, если
    skip_frame=10, только каждый десятый график сохраняется в файл;
    это удобно, если нужно сравнивать графики для различных моментов
    времени).
    """
    import scitools.std as st
    from scitools.MovingPlotWindow import MovingPlotWindow
    from math import pi

    # Удаляем все старые графики tmp_*.png
    import glob, os
    for filename in glob.glob('tmp_*.png'):
        os.remove(filename)

    P = 2*pi/omega  # один период
    umin = 1.2*u.min();  umax = -umin
    tau = t[1] - t[0]
    plot_manager = MovingPlotWindow(
        window_width=8*P,
        tau=tau,
        yaxis=[umin, umax],
        mode='continuous drawing')
    frame_counter = 0
    for n in range(1,len(u)):
        if plot_manager.plot(n):
            s = plot_manager.first_index_in_plot
            st.plot(t[s:n+1], u[s:n+1], 'r-1',
                    t[s:n+1], U*np.cos(omega*t)[s:n+1], 'b-1',
                    title='t=%6.3f' % t[n],
                    axis=plot_manager.axis(),
                    show=not savefig) # пропускаем окно, если savefig
            if savefig and n % skip_frames == 0:
                filename = 'tmp_%04d.png' % frame_counter
                st.savefig(filename)
                print u'Создаем графический файл', filename, 't=%g' % t[n]
                frame_counter += 1
        plot_manager.update(n)
```

Можно вызывать эту функцию в функции `main`, если число периодов при моделировании больше 10. Запуск вычислений для безразмерной модели (значения, заданные по умолчанию, для аргументов командной строки `--U` и `--omega` соответствуют безразмерной модели) для 40 периодов с 20 шагами на период выглядит следующим образом

```
Terminal> python vib_undamped.py --dt 0.05 --num_periods 40
```

Появится окно с движущимся графиком, на котором мы можем видеть изменение точного и приближенного решений со временем. На этих графиках мы видим, что погрешность угловой частоты мала в начале расчета, но становится более заметной со временем.

1.3.2 Создание анимации

Стандартные видео форматы

Функция `visualize_front` сохраняет все графики в файлы с именами: `tmp_0000.png`, `tmp_0001.png`, `tmp_0002.png` и т.д. Из этих файлов мы можем создать видео файл, например, в формате `mpeg4`:

```
Terminal> ffmpeg -r 12 -i tmp_%04d.png -c:v libx264 movie.mp4
```

Программа `f fmpeg` имеется в репозитариях Ubuntu. Можно использовать другие програ ммы для создания видео из набора отдельных графических файлов. Для г енерации других видео форматов с помощью `ffmpeg` можно использовать соответствующие кодеки и расширения для выходных файлов:

Формат	Кодек и имя файла
Flash	`-c:v flv movie.flv`
MP4	`-c:v libx264 movie.mp4`
WebM	`-c:v libx264 movie.mp4`
Ogg	`-c:v libtheora movie.ogg`

Видео файл можно проиграть каким-либо видео плейером. Также можно использовать веб-браузер, создав веб-страницу, содержащую HTML5-тег `video`:

```
<video autoplay loop controls
       width='640' height='365' preload='none'>
<source src='movie.webm'  type='video/webm; codecs="vp8, vorbis"'>
</video>
```

Современные браузеры поддерживают не все видео форматы. MP4 необходим для просмотра на устройствах Apple, которые используют браузер Safari. WebM — предпочтительный формат для Chrome, Opera, Firefox и IE v9+. Flash был популярен раньше, но старые браузеры, которые использовали Flash могут проигрывать MP4. Все браузеры, которые работают с форматом Ogg, могут также воспроизводить WebM. Это означает, что для того, чтобы видео можно было просмотреть в любом браузере, это видео должно быть доступно в форматах MP4 и WebM. Соответствующий HTML код представлен ниже:

```
<video autoplay loop controls width='640' height='365' preload='none'>
<source src='movie.mp4'   type='video/mp4; codecs="avc1.42E01E, mp4a.40.2"'>
<source src='movie.webm'  type='video/webm; codecs="vp8,     vorbis"'>
</video>
```

Формат MP4 должен идти первым для того, чтобы устройства Apple могли корректно загружать видео.

Предупреждение: Для того, чтобы быть уверенным в том, что отдельные графические кадры в итоге показывались в правильном порядке, необходимо нумеровать файлы используя нули вначале номера (0000, 0001, 0002 и т.д.). Формат `%04d` задает отображение целого числа в поле из 4 символов, заполненном слева нулями.

Проигрыватель набора PNG файлов в браузере

Команда `scitools movie` может создать видео проигрыватель для набора PNG так, что можно будет использовать браузер для просмотра "видео". Преимущество такой реализации в том, что пользователь может контролировать скорость изменения графиков. Команда для генерации HTML с проигрывателем набора PNG файлов `tmp_*.png` выглядит следующим образом:

```
Terminal> scitools movie output_file=vib.html fps=4 tmp_*.png
```

Параметр `fps` управляет скоростью проигрывания видео (*количество фреймов в секунду*).

Для просмотра видео достаточно загрузить страницу `vib.html` в какой-либо браузер.

Создание анимированных GIF файлов

Из набора PNG файлов можно также создать анимированный GIF, используя программу `convert` программного пакета :

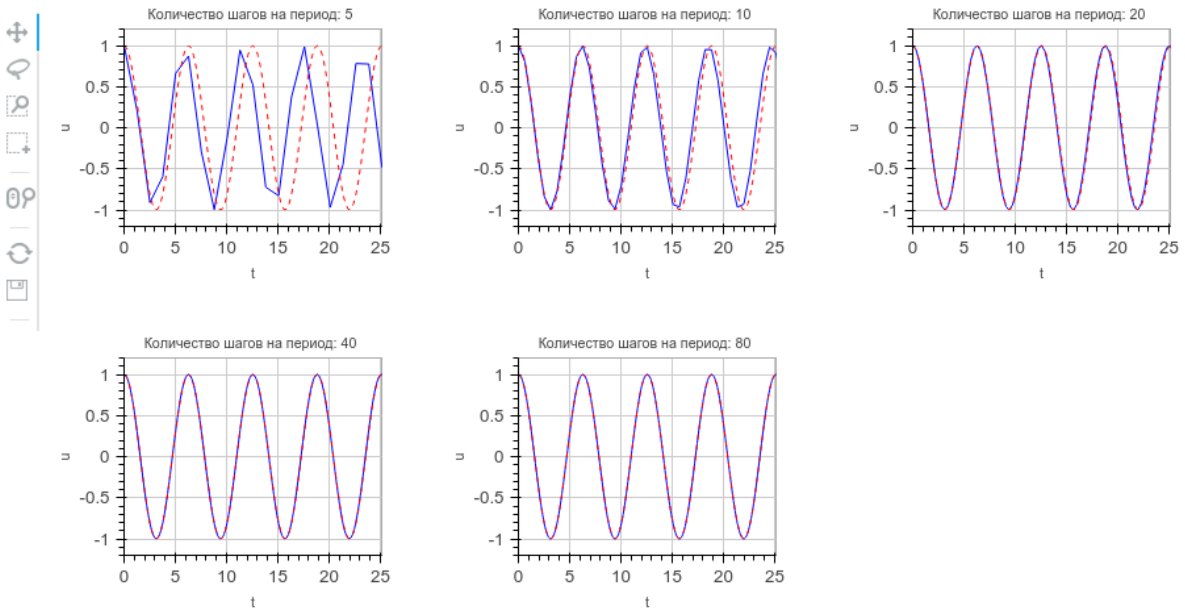
```
convert -delay 25 tmp_vib*.png tmp_vib.gif
```

Параметр `delay` устанавливает задержку между фреймами, измеряемую в 1/100 с, таким образом 4 фрейма в секунду здесь задается задержкой 25/100 с. Отметим, что в нашем случае расчета 40 периодов с шагом $\tau = 0.05$, процесс создания GIF из большого набора PNG файлов является ресурсоемким, поэтому такой подход не стоит использовать. Анимированный GIF может быть подходящим, когда используется небольшое количество фреймов, нужно анализировать каждый фрейм и проигрывать видео медленно.

1.3.3 Использование Bokeh для сравнения графиков

Вместо динамического изменения графиков, можно использовать средства для расположения графиков на сетке с помощью мыши. Например, мы можем расположить четыре периода на графиках, а затем с помощью мыши прокручивать остальные временные отрезки. Графическая библиотека предоставляет такой инструментарий, но графики должны просматриваться в браузере. Библиотека имеет отличную документацию, поэтому здесь мы покажем, как она может использоваться при сравнении набора графиков функции $u(t)$, соответствующих длительному моделированию.

Допустим, что мы хотим выполнить эксперименты для серии значений τ. Нам нужно построить совместные графики приближенного и точного решения для каждого шага τ и расположить их на сетке:



Далее мы можем перемещать мышью кривую в одном графике, в других кривые будут смещаться автоматически. Пример такого интерактивного поведения можно увидеть на по следующей ссылке: .

Функция, генерирующая html страницу с графиками с использованием библиотеки Bokeh по заданным спискам массивов `u` и соответствующих массивов `t` для различных вариантов расчета, представлена ниже

```
def bokeh_plot(u, t, legends, U, omega, t_range, filename):
        """
        Строится график зависимости приближенного решения от t с
        использованием библиотеки Bokeh.
        u и t - списки (несколько экспериментов могут сравниваться).
        легенды содержат строки для различных пар u,t.
    """
        if not isinstance(u, (list,tuple)):
                u = [u]
        if not isinstance(t, (list,tuple)):
                t = [t]
        if not isinstance(legends, (list,tuple)):
                legends = [legends]

        import bokeh.plotting as plt
        plt.output_file(filename, mode='cdn', title='Comparison')
        # Предполагаем, что все массивы t имеют одинаковые размеры
        t_fine = np.linspace(0, t[0][-1], 1001)  # мелкая сетка для точного решения
        tools = 'pan,wheel_zoom,box_zoom,reset,'\
                'save,box_select,lasso_select'
        u_range = [-1.2*U, 1.2*U]
        font_size = '8pt'
        p = []  # список графических объектов
        # Создаем первую фигуру
        p_ = plt.figure(
                width=300, plot_height=250, title=legends[0],
                x_axis_label='t', y_axis_label='u',
                x_range=t_range, y_range=u_range, tools=tools,
                title_text_font_size=font_size)
```

```
        p_.xaxis.axis_label_text_font_size=font_size
        p_.yaxis.axis_label_text_font_size=font_size
        p_.line(t[0], u[0], line_color='blue')
        # Добавляем точное решение
        u_e = u_exact(t_fine, U, omega)
        p_.line(t_fine, u_e, line_color='red', line_dash='4 4')
        p.append(p_)
        # Создаем оставшиеся фигуры и добавляем их оси к осям первой фигуры
        for i in range(1, len(t)):
                p_ = plt.figure(
                        width=300, plot_height=250, title=legends[i],
                        x_axis_label='t', y_axis_label='u',
                        x_range=p[0].x_range, y_range=p[0].y_range, tools=tools,
                        title_text_font_size=font_size)
                p_.xaxis.axis_label_text_font_size = font_size
                p_.yaxis.axis_label_text_font_size = font_size
                p_.line(t[i], u[i], line_color='blue')
                p_.line(t_fine, u_e, line_color='red', line_dash='4 4')
                p.append(p_)

        # Располагаем все графики на сетке с 3 графиками в строке
        grid = [[]]
        for i, p_ in enumerate(p):
                grid[-1].append(p_)
                if (i+1) % 3 == 0:
                        # Новая строка
                        grid.append([])
        plot = plt.gridplot(grid, toolbar_location='left')
        plt.save(plot)
        plt.show(plot)
```

Приведем также пример использования функции `bokeh_plot`:

```
def demo_bokeh():
    """Решаем обезразмеренное ОДУ u'' + u = 0."""
    omega = 1.0         # обезразмеренная задача (частота)
    P = 2*np.pi/omega  # период
    num_steps_per_period = [5, 10, 20, 40, 80]
    T = 40*P        # Время моделирования: 40 периодов
    u = []          # список с приближенными решениями
    t = []          # список с соответствующими сетками
    legends = []
    for n in num_steps_per_period:
        tau = P/n
        u_, t_ = solver(U=1, omega=omega, tau=tau, T=T)
        u.append(u_)
        t.append(t_)
        legends.append(u'Количество шагов на период: %d' % n)
    bokeh_plot(u, t, legends, U=1, omega=omega, t_range=[0, 4*P],
               filename='tmp.html')
```

1.3.4 Практический анализ решения

Для колебательной функции, аналогичной представленной на рисунке 1.1, мы можем вычислить амплитуду и частоту (или период) на основе моделирования. Мы пробегаем дискретное множество точек решения (t_n, y^n) и находим все точки экстремумов.

Расстояние между двумя последовательными точками максимума (или минимума) можно использовать для оценки локального периода, при этом половина разницы между максимальным и ближайшим к нему минимальным значениями y дают оценку локальной амплитуды.

Локальный максимум — это точки, где выполнено условие

$$y^{n-1} < y^n > y^{n+1}, \quad n = 1, 2, \ldots, N.$$

Аналогично определяются точки локального минимума

$$y^{n-1} > y^n < y^{n+1}, \quad n = 1, 2, \ldots, N.$$

Ниже приведена функция определения локальных максимумов и минимумов

```
def minmax(t, u):
    """
    Вычисляются все локальные минимумы и максимумы сеточной функции
    u(t_n), представленной массивами u и t. Возвращается список минимумов
    и максимумов вида (t[i],u[i]).
    """
    minima = []; maxima = []
    for n in range(1, len(u)-1, 1):
        if u[n-1] > u[n] < u[n+1]:
            minima.append((t[n], u[n]))
        if u[n-1] < u[n] > u[n+1]:
            maxima.append((t[n], u[n]))
    return minima, maxima
```

Два возвращаемых объекта — списки кортежей.

Пусть (t_k, e^k), $k = 0, 1, \ldots, M-1$ — последовательность всех M точек максимума, где t_k — момент времени и e^k — соответствующее значение сеточной функции y. Локальный период можно определить как $p_k = t_{k+1} - t_k$, что на языке Python можно реализовать следующим образом:

```
def periods(extrema):
    """
    По заданному списку (t,u) точек минимума или максимума возвращается
    массив соответствующих локальных периодов.
    """
    p = [extrema[n][0] - extrema[n-1][0]
         for n in range(1, len(extrema))]
    return np.array(p)
```

Зная минимумы и максимумы, мы можем определить локальные амплитуды через разницы между соседними точками максимумов и минимумов:

```
def amplitudes(minima, maxima):
    """
    По заданным спискам точек локальных минимумов и максимумов
    возвращается массив соответсвующих локальных амплитуд.
    """
    # Сравнивается первый максимум с первым минимумом и т.д.
    a = [(abs(maxima[n][1] - minima[n][1]))/2.0
         for n in range(min(len(minima),len(maxima)))]
    return np.array(a)
```

Так как a[k] и p[k] соответствуют k -тым оценкам амплитуды и периода, соответственно, удобно отобразить графически зависимость значений a и p от индекса k.

При анализе больших временных рядов выгодно вычислять и визуализировать p и a вместо y для того, чтобы получить представление о распространении колебаний. Покажем как это сделать для безразмерной задачи при $\tau = 0.1, 0.5, 0.01$. Пусть заготовлена следующая функция:

```
def plot_empirical_freq_and_amplitude(u, t, U, omega):
    """
    Находит эмпирически угловую частоту и амплитуду при вычислениях,
    зависящую от u и t. u и t могут быть массивами или (в случае
    нескольких расчетов) многомерными массивами.
    Одно построение графика выполняется для амплитуды и одно для
    угловой частоты (на легендах названа просто частотой).
    """
    from vib_empirical_analysis import minmax, periods, amplitudes
    from math import pi
    if not isinstance(u, (list,tuple)):
        u = [u]
        t = [t]
    legends1 = []
    legends2 = []
    for i in range(len(u)):
        minima, maxima = minmax(t[i], u[i])
        p = periods(maxima)
        a = amplitudes(minima, maxima)
        plt.figure(1)
        plt.plot(range(len(p)), 2*pi/p)
        legends1.append(u'Частота, case%d' % (i+1))
        plt.hold('on')
        plt.figure(2)
        plt.plot(range(len(a)), a)
        plt.hold('on')
        legends2.append(u'Амплитуда, case%d' % (i+1))
    plt.figure(1)
    plt.plot(range(len(p)), [omega]*len(p), 'k--')
    legends1.append(u'Точная частота')
    plt.legend(legends1, loc='lower left')
    plt.axis([0, len(a)-1, 0.8*omega, 1.2*omega])
    plt.savefig('tmp1.png');  plt.savefig('tmp1.pdf')
    plt.figure(2)
    plt.plot(range(len(a)), [U]*len(a), 'k--')
    legends2.append(u'Точная амплитуда')
    plt.legend(legends2, loc='lower left')
    plt.axis([0, len(a)-1, 0.8*U, 1.2*U])
    plt.savefig('tmp2.png');  plt.savefig('tmp2.pdf')
    plt.show()
```

Мы можем написать небольшую программу для создания графиков:

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

from vib_undamped import solver, plot_empirical_freq_and_amplitude
from math import pi

tau_values = [0.1, 0.5, 0.01]
u_cases = []
```

```
t_cases = []

for tau in tau_values:
        # Расчитываем безразмерную модель для 40 периодов
        u, t = solver(U = 1, omega = 2*pi, tau = tau, T = 40)
        u_cases.append(u)
        t_cases.append(t)

plot_empirical_freq_and_amplitude(u_cases, t_cases, U = 1, omega = 2*pi)
```

На рисунке 1.2 представлен результат работы программы: очевидно, что уменьшение шага расчета τ существенно улучшает угловую частоту, при этом амплитуда тоже становиться более точной. Линии для $\tau = 0.01$, соответствующие 100 шагам на период, сложно отличить от точных значений.

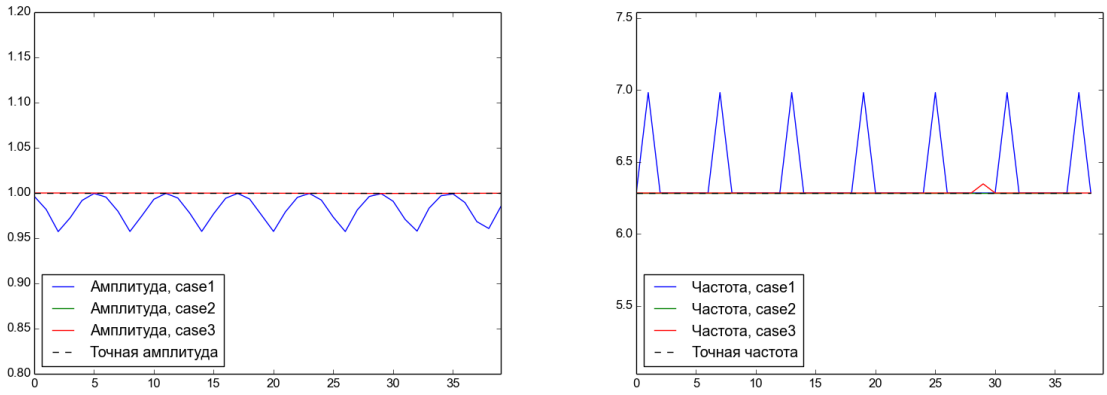


Figure1.2.: Эмпирические амплитуды и угловые частоты для трех значений временного шага

1.4 Анализ конечно-разной схемы

1.4.1 Вывод решения конечно-разностной схемы

Как мы видели в предыдущем разделе погрешность частоты растет со временем. Оценим эту погрешность теоретически. Проведем анализ на основе точного решения дискретной задачи. Разностное уравнение *(1.6)* — однородное с постоянными коэффициентами. Известно, что такие уравнения имеют решения вида $y^n = cq^n$, где q — некоторое число, определяемое из разностного уравнения, а постоянная c определяется из начального условия ($c = U$). Здесь верхний индекс n в y^n обозначает временной слой, а в q^n — степень.

Будем искать q в виде

$$q = e^{i\tilde{\omega}t},$$

и решим задачу относительно $\tilde{\omega}$. Напомним, что $i = \sqrt{-1}$. Имеем

$$q^n = e^{\tilde{\omega}\tau n} = e^{i\tilde{\omega}t} = \cos(\tilde{\omega}t) + i\sin(\tilde{\omega}t).$$

В качестве физически обоснованного численного решения возьмем действительную часть этого комплексного выражения.

Вычисления дают

$$\begin{aligned} y^n_{\bar{t}t} &= \frac{y^{n+1} - 2y^n + y^{n-1}}{\tau^2} \\ &= U\frac{q^{n+1} - 2q^n + q^{n-1}}{\tau^2} \\ &= \frac{U}{\tau^2}\left(e^{i(\tilde{\omega}t+\tau)} - 2e^{i(\tilde{\omega}t)} + e^{i(\tilde{\omega}t-\tau)}\right) \\ &= Ue^{i(\tilde{\omega}t)}\frac{1}{\tau^2}\left(e^{i\tau} + e^{-i\tau} - 2\right) \\ &= Ue^{i(\tilde{\omega}t)}\frac{2}{\tau^2}\left(\cos(\tilde{\omega}\tau) - 1\right) \\ &= -Ue^{i(\tilde{\omega}t)}\frac{4}{\tau^2}\sin^2\left(\frac{\tilde{\omega}\tau}{2}\right) \end{aligned}$$

Подставляя $y^n = Ue^{\tilde{\omega}\tau n}$ в *(1.6)* и учитывая последнее выражение, получим

$$-Ue^{i(\tilde{\omega}t)}\frac{4}{\tau^2}\sin^2\left(\frac{\tilde{\omega}\tau}{2}\right) + \omega^2 Ue^{i(\tilde{\omega}t)} = 0.$$

Разделив последнее выражение на $Ue^{i(\tilde{\omega}t)}$, получим

$$\frac{4}{\tau^2}\sin^2\left(\frac{\tilde{\omega}\tau}{2}\right) = \omega^2.$$

Отсюда

$$\sin^2\left(\frac{\tilde{\omega}\tau}{2}\right) = \left(\frac{\omega\tau}{2}\right)^2$$

и, следовательно, имеем

$$\tilde{\omega} = \pm\frac{2}{\tau}\arcsin\left(\frac{\omega\tau}{2}\right). \tag{1.11}$$

Из (1.11) видно, что численная частота $\tilde{\omega}$ никогда не равна точной ω. Для того, чтобы понять насколько хороша аппроксимация (1.11), используем разложение в ряд Тейлора для малых τ:

```
>>> from sympy import *
>>> tau, omega = symbols('tau omega')
>>> omega_tilde_e = 2/tau*asin(omega*tau/2)
>>> omega_tilde_series = w_tilde_e.series(tau, 0, 4)
>>> print omega_tilde_series
omega + tau**2*omega**3/24 + O(tau**4)
```

Таким образом, имеем

$$\tilde{\omega} = \omega\left(1 + \frac{1}{24}\omega^2\tau^2\right) + \mathcal{O}(\tau^4). \tag{1.12}$$

Погрешность численного значения частоты имеет второй порядок по τ и стремится к нулю при $\tau \to 0$. Из (1.12) видно, что $\tilde{\omega} > \omega$, так как слагаемое $\omega^3\tau/24 > 0$. Это слагаемое вносит наибольший вклад в погрешность. Слишком большая численная частота дает слишком быстро колеблющийся профиль и, таким образом, решение как бы запаздывает, это хорошо видно в левой части рисунка 1.1.

На рисунке 1.3 представлены графики дискретной частоты вычисленной по (1.11) и ее приближения (1.12) для $\omega = 1$.

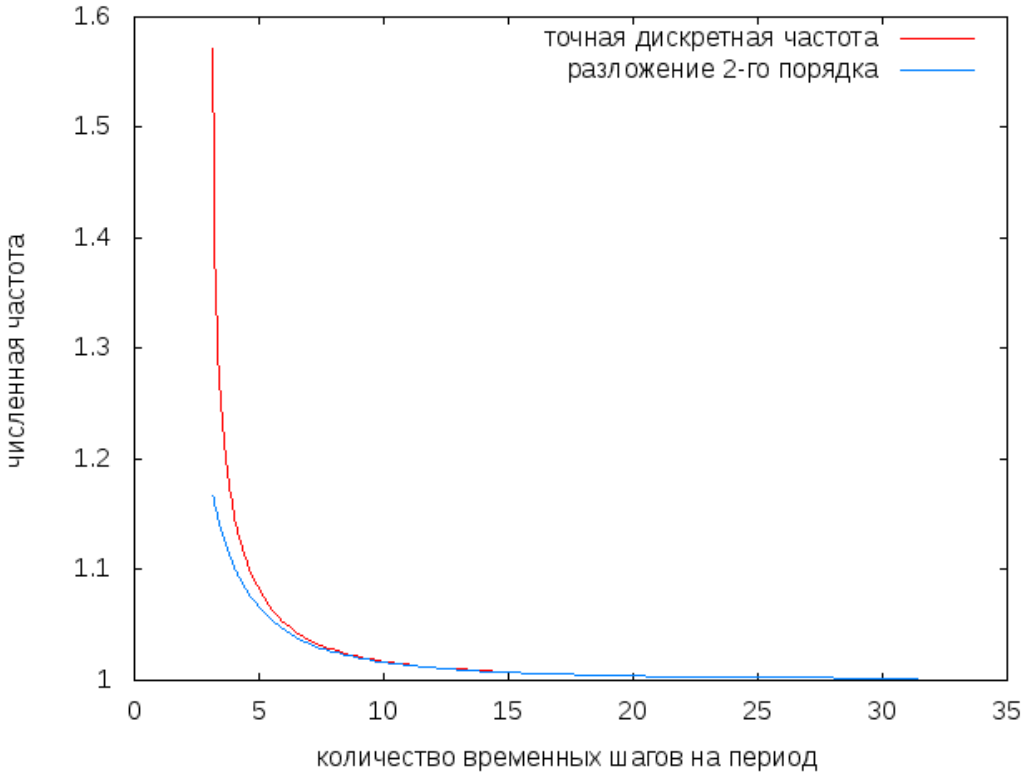


Figure1.3.: Точная дискретная частота и ее разложение в ряд второго порядка

1.4.2 Точное дискретное решение

Возможно, чем то, что $\tilde{\omega} = \omega + \mathcal{O}(\tau^2)$, результат заключается в том, что мы нашли точное дискретное решение задачи:

$$y^n = U\cos(\tilde{\omega}\tau n), \quad \tilde{\omega} = \frac{2}{\tau}\arcsin\left(\frac{\omega\tau}{2}\right). \tag{1.13}$$

Теперь мы можем вычислить сеточную функцию погрешности:

$$\begin{aligned} e^n &= u_e(t_n) - u^n = U\cos(\omega\tau n) - U\cos(\tilde{\omega}\tau n) \\ &= -2U\sin\left(\frac{t}{2}(\omega-\tilde{\omega})\right)\sin\left(\frac{t}{2}(\omega+\tilde{\omega})\right). \end{aligned} \tag{1.14}$$

Построенная сеточная функция погрешности идеальна для целей тестирования поэтому необходимо реализовать тест на основе (1.13), выполнив *упражнение 2*.

1.4.3 Сходимость

Для того, чтобы показать, что приближенное решение сходится к точному, т.е. $e^n \to 0$ при $\tau \to 0$, воспользуемся (1.11):

$$\lim_{\tau\to 0} = \lim_{\tau\to 0}\frac{2}{\tau}\arcsin\left(\frac{\omega\tau}{2}\right) = \omega.$$

Это можно проверить, например, с помощью `sympy`:

```
>>> import sympy as sym
>>> tau, omega = sym.symbols('tau omega')
>>> sym.limit((2/tau)*sym.asin(omega*tau/2),tau,0,dir='+')
omega
```

1.4.4 Глобальная погрешность

Проведем анализ глобальной погрешности. Разложим сеточную функцию погрешности (1.14). Воспользуемся для этого пакетом `sympy`:

```
>>> from sympy import *
>>> omega_tilde_e = 2/tau*asin(omega*tau/2)
>>> omega_tilde_series = omega_tilde_e.series(tau, 0, 4)
>>> omega_tilde_series
omega + omega**3*tau**2/24 + O(tau**4)
```

Можно использовать команду `removeO()`, чтобы избавиться от слагаемого `O()`:

```
>>> omega_tilde_series = omega_tilde_series.removeO()
>>> omega_tilde_series
omega**3*tau**2/24 + omega
```

Используя выражение для $\tilde{\omega}$ и разлагая погрешность в ряд, получим

```
>>> error = cos(omega*t) - cos(omega_tilde_series*t)
>>> error.series(tau, 0, 6)
omega**3*t*tau**2*sin(omega*t)/24 + omega**6*t**2*tau**4*cos(omega*t)/1152 + O(tau**6)
```

Так как нас интересует главное слагаемое в разложении (слагаемое с наименьшей степенью τ), воспользуемся методом `.as_leading_term(tau)`, чтобы выделить это слагаемое:

```
>>> error.series(tau, 0, 6).as_leading_term(tau)
omega**3*t*tau**2*sin(omega*t)/24
```

Последний результат означает, что глобальная погрешность в точке t пропорциональна $\omega^3 t\tau$. Учитывая, что $t = n\tau$ и $\sin(\omega t) \leq 1$, получим

$$e^n = \frac{1}{24} n\omega^3\tau^3.$$

Это главный член погрешности *в точке*.

Нас интересует накапливаемая глобальная оценка погрешности, которую можно вычислить как ℓ^2 норму погрешности e^n:

$$\|e^n\|_2^2 = \tau \sum_{n=0}^{N} \frac{1}{24^2} n^2\omega^6\tau^6 = \frac{1}{24^2}\tau^7 \sum_{n=0}^{N} n^2.$$

Сумма $\sum_{n=0}^{N} n^2$ примерно равен $\frac{1}{3}N^3$. Заменяя N на T/τ, получим

$$\|e^n\|_2 = \frac{1}{24}\sqrt{\frac{T^3}{3}}\omega^3\tau^2.$$

Таким образом, мы получили, что глобальная (интегральная) погрешность также пропорциональна τ^2.

1.4.5 Устойчивость

Как мы помним численное решение имело правильную постоянную амплитуду, но ошибку в частоте. Однако, постоянная амплитуда бывает не всегда. Отметим, что если τ достаточно большая величина, величина аргумента функции arcsin в (1.12) может быть больше 1, т.е. $\omega\tau/2 > 1$. В этом случае $\arcsin(\omega\tau/2)$ и, следовательно, $\tilde{\omega}$ являются комплексными:

```
>>> omega = 1.
>>> tau = 3
>>> asin(omega*tau/2)
1.5707963267949 - 0.962423650119207*I
```

Пусть $\tilde{\omega} = \tilde{\omega}_r + i\tilde{\omega}_i$. Так как $\arcsin(x)$ имеет отрицательную мнимую часть при $x > 1$, $\tilde{\omega}_i < 0$, то $e^{i\tilde{\omega}t} = e^{-\tilde{\omega}t}e^{i\tilde{\omega}t}$, что означает экспоненциальный рост со временем, так как $e^{-\tilde{\omega}t}$ при $\tilde{\omega} < 0$ имеет положительную степень.

Условие устойчивости

Мы должны исключить рост амплитуды, потому что такой рост отсутствует в точном решении. Таким образом, мы должны наложить *условие устойчивости*: аргумент арксинуса должен давать действительное значение $\tilde{\omega}$. Условие устойчивости выглядит следующим образом:

$$\frac{\omega\tau}{2} \leq 1 \rightarrow \tau \leq \frac{2}{\omega}. \tag{1.15}$$

Возьмем $\omega = 2\pi$ и выберем $\tau > \pi^{-1} = 0.3183098861837907$. На рисунке 1.4 представлен результат расчета при $\tau = 0.3184$, которое незначительно отличается от критического значения: $\tau = \pi^{-1} + 9.01 \cdot 10^{-5}$.

1.4.6 О точности при границе устойчивости

Ограничение на временной шаг $\tau < 2/\omega$ кажется неудачным. Хотелось бы использовать больший шаг для расчета, чтобы ускорить его. При граничном значении шага из условия устойчивости имеем $\arcsin(\omega\tau/2) = \arcsin(1) = \pi/2$ и, следовательно, $\tilde{\omega} = \pi/\tau$. Соответствующий период численного решения $\tilde{P} = 2\pi/\tilde{\omega} = 2/\tau$, которое означает, что экстремумы численного решения находятся на расстоянии одного временного шага. Это самая короткая волна, которую можно воспроизвести на сетке. Другими словами, нет необходимости использовать больший, чем задан ограничением, шаг по времени при счете.

Кроме того, мы видим при счете, что ошибка угловой частоты существенна: на рисунке 1.5 показаны приближенное и точное решения при $\omega = 2\pi$ и $\tau = 2/\omega = \pi^{-1}$. Уже после одного периода у численного решения наблюдается минимум там, где у точного максимум. Погрешность в частоте при τ выбирается на границе устойчивости равна: $\omega - \tilde{\omega} = \omega(1 - \pi/2) \approx -0.57\omega$. Соответствующая погрешность периода: $P - \tilde{P} \approx 0.36P$. После m периодов погрешность становиться уже $0.36mP$. Эта ошибка достигает половины периода при $m = 1/(2 \cdot 0.36) \approx 1.38$, что теоретически объясняет результаты расчета, представленные на рисунке 1.5. Следовательно, временной шаг τ следует выбирать как можно меньшим, чтобы добиться поддающихся интерпретации результатов.

Краткие выводы

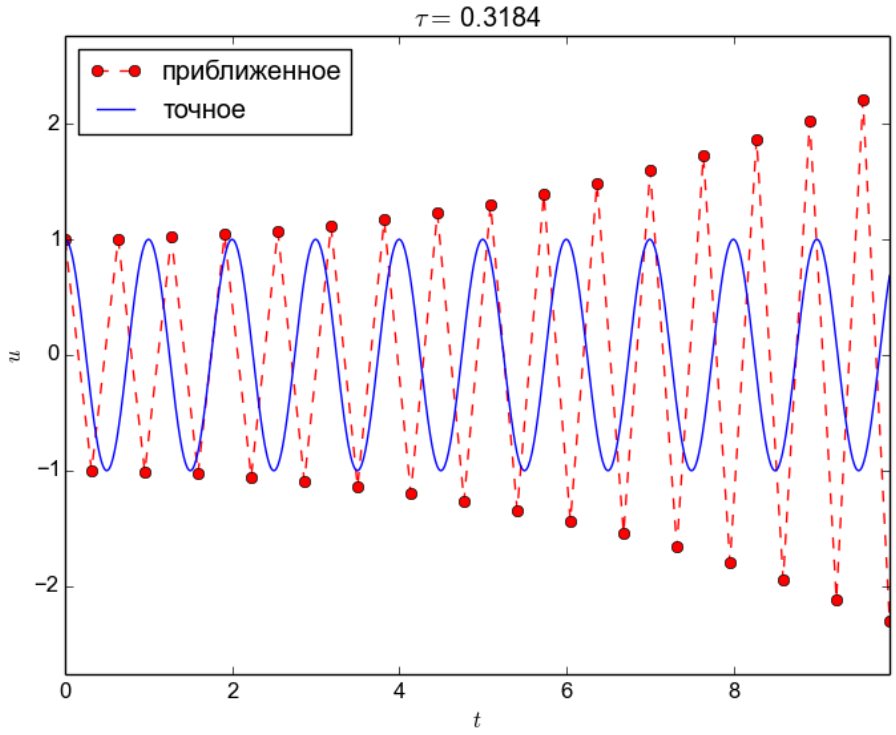


Figure1.4.: Неустойчивое решение

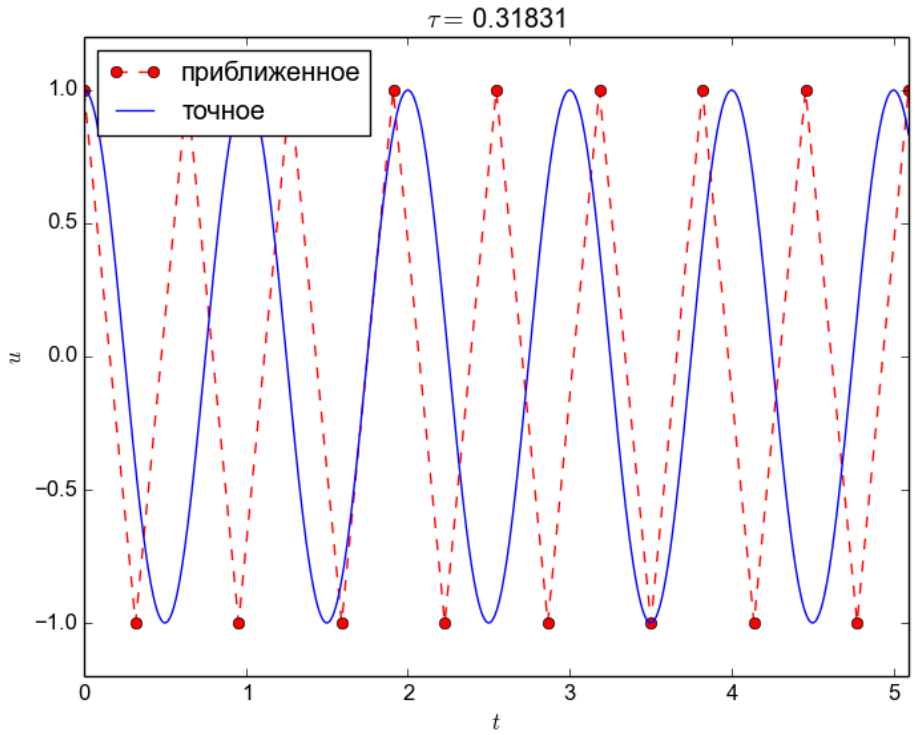


Figure1.5.: Численное решение при $\tau = 2/\omega$.

1. Ключевой параметр в формулах $p = \omega\tau$. Пусть период колебаний $P = 2\pi/\omega$, количество временных шагов на период $N_P = P/\tau$. Тогда $p = \omega\tau = 2\pi N_P$, т.е. основным параметром является количество временных шагов на период. Наименьшее возможное $N_P = 2$, т.е. $p \in (0, \pi]$.
2. Если $p \le 2$, то амплитуда численного решения постоянна.
3. Отношение численной угловой частоты к точной есть $\tilde{\omega}/\omega \approx 1 + \frac{p^2}{24}$. Погрешность $\frac{p^2}{24}$ приводит к смещенным пикам численного решения, и это погрешность расположения пиков растет линейно со временем.

1.5 Обобщения: затухание, нелинейные струны и внешние воздействия

Рассмотрим обобщение задачи, рассмотренной в разделе *Конечно-разностная дискретизация*, учитывающее возможные затухание $f(u')$, нелинейную пружину (или сопротивление) $s(u)$ и некоторое внешнее воздействие $F(t)$:

$$\begin{aligned} mu'' + f(u') + s(u) = F(t), \quad t \in (0, T], \\ u(0) = U, \quad u'(0) = V. \end{aligned} \tag{1.16}$$

Здесь $m, f(u'), s(u), F(t), U, V$ и T — входные параметры.

Будем рассматривать два основных типа затуханий (силы трения): линейное $f(u') = bu'$ и квадратичное $f(u') = bu'|u'|$. Пружинные системы часто характеризуются линейным затуханием, при этом сопротивление воздуха описывается квадратичным затуханием. Сила сжатие пружины часто линейны: $s(u) = cu$, однако бывают и нелинейными, наиболее известный пример — силы тяжести, действующие на маятник, которые описываются нелинейным слагаемым $s(u) \sim \sin(u)$.

1.5.1 Разностная схема для линейного затухания

Для приближенного решения уравнения (1.16) в случае линейного затухания будем использовать следующую разностную схему:

$$\begin{aligned} my^n_{\bar{t}t} + by^n_{\mathring{t}} + s(y^n) = F^n, \\ n = 1, 2, \ldots, N-1, \\ y^0 = U, \quad y^0_{\mathring{t}} = \frac{y^1 - y^{-1}}{2\tau} = V. \end{aligned} \tag{1.17}$$

Перепишем задачу (1.17) в индексной форме:

$$\begin{aligned} m\frac{y^{n+1} - 2y^n + y^{n-1}}{\tau^2} + b\frac{y^{n+1} - y^{n-1}}{2\tau} + s(y^n) = F^n, \\ n = 1, 2, \ldots, N-1, \\ y^0 = U, \quad \frac{y^1 - y^{-1}}{2\tau} = V. \end{aligned} \tag{1.18}$$

Решая разностное уравнение (1.18) относительно неизвестной y^{n+1}, получим следующую рекуррентную формулу:

$$y^{n+1} = \frac{2my^n + (0.5\tau - m)y^{n-1} + \tau^2(F^n - s(y^n))}{m + 0.5b\tau}, \quad n = 1, 2, \ldots, N-1. \tag{1.19}$$

При $n = 0$ с учетом второго начального условия имеем

$$y^1 = y^0 + \tau V + \frac{\tau^2}{2m}(F^0 - s(y^0) - bV). \tag{1.20}$$

1.5.2 Разностная схема для квадратичного затухания

Пусть $f(u') = bu'|u'|$. Аппроксимацию $f(u')$ выполним, основываясь на использовании геометрического среднего:

$$(w^2)^n \approx w^{n-1/2} w^{n+1/2},$$

где w — некоторая сеточная функция. Погрешность при использовании геометрического среднего имеет порядок $O(\tau^2)$, такой же как и при аппроксимации второй производной. При $w = u'$ имеем

$$(u'|u'|)^n \approx u'(t_{n+1/2})|u'(t_{n-1/2})|.$$

Для аппроксимации u' в точках $t_{n\pm1}$ воспользуемся направленными разностями, которые имеют второй порядок аппроксимации относительно полуцелых точек:

$$u'(t_{n+1/2}) \approx \frac{y^{n+1} - y^n}{\tau} = y^n_t, \quad u'(t_{n-1/2}) \approx \frac{y^n - y^{n-1}}{\tau} = y^n_{\bar{t}}.$$

Таким образом, получим разностное уравнение

$$m y^n_{\bar{t}t} + b y^n_t y^n_{\bar{t}} + s(y^n) = F^n, \quad n = 1, 2, \ldots, N-1, \tag{1.21}$$

которая является линейной относительно y^{n+1}:

$$y^{n+1} = \frac{2my^n - my^{n-1} + by^n|y^n - y^{n-1}| + \tau^2(F^n - s(y^n))}{m + b|y^n - y^{n-1}|}. \tag{1.22}$$

При использовании аппроксимации второго начального условия из (1.17) мы получим сложное нелинейное выражение для y^1. Однако, можно построить аппроксимацию первого начального, линейную относительно y^1. Для $t = 0$ имеем $u'(0)|u'(0)| = bV|V|$. Используя это выражение в уравнении (1.21) и значение $u(0) = U$ при аппроксимации уравнения (1.16), записанного при $t = 0$, получим

$$m y^1_{\bar{t}t} + bV|V| + s(U) = F^0.$$

Отсюда, учитывая второе начальное условие из (1.17), получим выражение для вычисления y^1

$$y^1 = y^0 + \tau V + \frac{\tau^2}{2m}(F^0 - mV|V| - s(U)). \tag{1.23}$$

1.5.3 Программная реализация

Алгоритмы для линейного и квадратичного затуханий, построенные в предыдущих разделах, аналогичны алгоритму для незатухающей модели. Отличие только в формулах для y^1 и y^{n+1}.

Таким образом, для приближенного решения задачи необходимо выполнить следующие шаги:

1. $y^0 = U$;
2. вычисляем y^1, используя (1.20) для линейного затухания или (1.23) для квадратичного затухания;
3. для $n = 1, 2, \ldots, N-1$:
   (a) вычисляем y^{n+1}, используя (1.19) для линейного затухания или (1.22) для квадратичного затухания.

Соответствующая функция `solver` представлена ниже:

```
def solver(U, V, m, b, s, F, tau, T, damping='linear'):
    """
    Решает задачу m*u'' + f(u') + s(u) = F(t) for t in (0,T],
    u(0)=U и u'(0)=V,
    конечно-разностной схемой с шагом tau.
    Если затухание 'линейно', то f(u')=b*u, если затухание 'квадратичное',
    то f(u')=b*u'*abs(u').
    F(t) и s(u) --- функции Python.
    """
    tau = float(tau); b = float(b); m = float(m) # avoid integer div.
    N = int(round(T/tau))
    u = np.zeros(N+1)
    t = np.linspace(0, N*tau, N+1)

    u[0] = U
    if damping == 'linear':
        u[1] = u[0] + tau*V + tau**2/(2*m)*(-b*V - s(u[0]) + F(t[0]))
    elif damping == 'quadratic':
        u[1] = u[0] + tau*V + \
               tau**2/(2*m)*(-b*V*abs(V) - s(u[0]) + F(t[0]))

    for n in range(1, N):
        if damping == 'linear':
            u[n+1] = (2*m*u[n] + (b*tau/2 - m)*u[n-1] +
                      tau**2*(F(t[n]) - s(u[n])))/(m + b*tau/2)
        elif damping == 'quadratic':
            u[n+1] = (2*m*u[n] - m*u[n-1] + b*u[n]*abs(u[n] - u[n-1])
                      + tau**2*(F(t[n]) - s(u[n])))/\
                      (m + b*abs(u[n] - u[n-1]))
    return u, t
```

1.5.4 Верификация реализации алгоритма

Постоянное решение

Для отладки и начальной верификации часто полезно использовать постоянное решение. Выбор $u_e(t) = U$ дает $V = 0$. Подставляя в уравнение, получим $F(t) = s(U)$ при любом выборе функции f. Так как разностная производная от константы равна нулю, то постоянное решение удовлетворяет также разностному уравнению. Следовательно, константа должна воспроизводиться с машинной точностью. Этот тест реализован в функции

```
def test_constant():
    """Тестирование постоянного решения."""
```

```
    u_exact = lambda t: U
    U = 1.2; V = 0; m = 2; b = 0.9
    omega = 1.5
    s = lambda u: omega**2*u
    F = lambda t: omega**2*u_exact(t)
    tau = 0.2
    T = 2
    u, t = solver(U, V, m, b, s, F, tau, T, 'linear')
    difference = np.abs(u_exact(t) - u).max()
    tol = 1E-13
    assert difference < tol

    u, t = solver(U, V, m, b, s, F, tau, T, 'quadratic')
    difference = np.abs(u_exact(t) - u).max()
    assert difference < tol
```

Линейное решение

Теперь в качестве тестового решения выберем линейную функцию: $u_e(t) = ct + d$. Начальное условие $u(0) = U$ дает $d = U$, а из второго начального условия $u'(0) = V$ получим, что $c = V$. Подставляя $u_e(t) = Vt + U$ в уравнение с линейным затуханием, имеем

$$0 + bV + s(Vt + U) = F(t),$$

а для квадратичного затухания:

$$0 + bV|V| + s(Vt + U) = F(t).$$

Так как все разностные аппроксимации используемые для u' точны для линейных функций, то линейная функция u_e также является решением разностной задачи.

Квадратичное решение

Функция $u_e(t) = bt^2 + Vt + U$ с произвольной постоянной b удовлетворяет начальным данным и уравнению с соответствующим выбором $F(t)$. Такая функция является также решением разностного уравнения с линейным затуханием. Однако, полином второй степени от t не удовлетворяет разностному уравнению в случае квадратичного затухания.

Выполните *упражнение 3*.

1.5.5 Визуализация

Функции для визуализации будут существенно отличаться от случая незатухающего решения, так как мы не знаем точного решения. Кроме того, у нас нет тех параметров, которые мы могли оценить в случае затухающих колебаний (периода колебаний, угловой частоты и т.п.). Поэтому пользователь должен задавать значение T и ширину окна.

Сценарий `vib.py` содержит несколько функций для визуализации решения.

1.5.6 Интерфейс командной строки

Функция `main` также существенно отличается от сценария, используемого для незатухающих колебаний, так как мы должны задавать дополнительные данные $s(u)$ и $F(t)$. Кроме того, нужно задавать T и ширину окна (вместо количества периодов). Для того чтобы понять можем мы строить один график для всего временного интервала или отображать только некоторую последнюю часть временного интервала, можно воспользоваться функцией `plot_empricial_freq_and_amplitude` для оценки количества локальных максимумов. Это количество сейчас возвращается функцией и используется в функции `main`.

```
def main():
    import argparse
    parser = argparse.ArgumentParser()
    parser.add_argument('--U', type=float, default=1.0)
    parser.add_argument('--V', type=float, default=0.0)
    parser.add_argument('--m', type=float, default=1.0)
    parser.add_argument('--b', type=float, default=0.0)
    parser.add_argument('--s', type=str, default='u')
    parser.add_argument('--F', type=str, default='0')
    parser.add_argument('--tau', type=float, default=0.05)
    parser.add_argument('--T', type=float, default=10)
    parser.add_argument('--window_width', type=float, default=30.,
                        help='Number of periods in a window')
    parser.add_argument('--damping', type=str, default='linear')
    parser.add_argument('--savefig', action='store_true')
    parser.add_argument('--SCITOOLS_easyviz_backend', default='matplotlib')
    a = parser.parse_args()
    from scitools.std import StringFunction
    s = StringFunction(a.s, independent_variable='u')
    F = StringFunction(a.F, independent_variable='t')
    U, V, m, b, tau, T, window_width, savefig, damping = \
       a.U, a.V, a.m, a.b, a.tau, a.T, a.window_width, a.savefig, \
       a.damping

    u, t = solver(U, V, m, b, s, F, tau, T, damping)
    num_periods = plot_empirical_freq_and_amplitude(u, t)
    num_periods = 4
    tit = 'tau = %g' % tau
    if num_periods <= 40:
        plt.figure()
        visualize(u, t, title=tit)
    else:
        visualize_front(u, t, window_width, savefig)
        visualize_front_ascii(u, t)
    show()
```

Сценарий `vib.py` содержит представленный выше фрагмент кода и решает модельную задачу (1.16). В качестве примера использования `vib.py` рассмотрим случай, когда $I = 1$, $V = 0$, $m = 1$, $s(u) = \sin(u)$, $F(t) = 3\cos(4t)$, $\tau = 0.05$ и $T = 140$. Соответствующий вызов сценария будет выглядеть следующим образом:

```
Terminal> python vib.py --s 'sin(u)' --F '3*cos(4*t)' --T 140
```

1.6 Упражнения и задачи

1.6.1 Упражнения

Упражнение 1: Использование ряда Тейлора для вычисления y^1

Альтернативный способ вывода *(1.7)* для вычисления y^1 заключается в использовании следующего ряда Тейлора:

$$u(t_1) \approx u(0) + \tau u'(0) + \frac{\tau}{2}u''(0) + O(\tau^3)$$

Используя уравнение *(1.1)* и начальное условие для производной $u'(0) = 0$, покажите, что такой способ также приведет к *(1.7)*. Более общее условие для $u'(0)$ имеет вид $u'(0) = V$. Получите формулу для вычисления y^1 двумя способами.

Упражнение 2: Использование точного дискретного решения для тестирования

Написать тестовую функцию в отдельном файле, которая использует точное дискретное решение *(4.3)* для проверки реализации функции `solver`.

Упражнение 3: Использование линейной и квадратичной функций для тестирования

Упражнение является обобщением *задачи 1* на расширенную задачу *(5.1)* на случаи, когда затухание линейное или квадратичное. Решите несколько подзадач и посмотрите как меняются результаты и настройки программы для случаев разных затуханий. При модификации кода из *задачи 1*, используйте `sympy`, который выполнит основную часть работы для анализа обобщенной задачи.

Упражнение 4: Показать линейный рост фазы со временем

Рассмотрим точное и приближенное решения $I\cos(\omega t)$ и $I\cos(\tilde{\omega}t)$, соответственно. Определить погрешность фазы как задержку по времени пика I точного решения и соответствующего пика приближенного решения после m периодов колебаний. Показать, что эта погрешность зависит линейно от m.

Упражнение 5: Улучшить точность регуляризацией частоты

Согласно *(4.2)* численная частота отклоняется от точной на величину $\omega^3\tau^2/24 > 0$. Замените параметр `omega` в функции `solver` из `vib_undamped.py` выражением $\omega * (1 - 1./24) * \omega^2\tau^2$ и проанализируйте как такая регуляризация влияет на точность.

Упражнение 6: Визуализация аппроксимации разностных производных для косинуса

Введем следующую величину

$$E = \frac{u^n_{\bar{t}t}}{u''(t_n)}$$

для измерения погрешности аппроксимации второй разностной производной. Вычислить E для функции вида $u(t) = \exp(i\omega t)$ (i — мнимая единица) и показать, что

$$E = \left(\frac{2}{\omega\tau}\right)^2 \sin^2\left(\frac{\omega\tau}{2}\right).$$

Построить график зависимости E от $p = \omega\tau \in [0, \pi]$. Отклонение кривой от единицы показывает погрешность аппроксимации. Также разложите E в ряд Тейлора по p до четвертой степени, используя `sympy`.

Упражнение 7: Минимизация использования памяти

Сценарий `vib.py` хранит все значения приближенного решения $y^0, y^1, \ldots, y^N$ в памяти, что удобно для последующего построения графиков. Сделать версию этого сценария, где только три последних значения y^{n+1}, y^n, y^{n-1} хранятся в памяти. Организуйте запись каждых посчитанных пар (t_{n+1}, y^{n+1}) в файл. Реализуйте визуализацию данных из файла.

1.6.2 Задачи

Задача 1: Использование линейной и квадратичной функций для тестирования

Рассмотрим задачу для ОДУ:

$$u'' + \omega^2 u = f(t), \quad u(0) = U, \quad u'(0) = V, \quad t \in (0, T].$$

Аппроксимируем уравнение разностной схемой $y^n_{\bar{t}t} + \omega^2 y^n = f^n$.

1. Вывести уравнение для нахождения приближенного решения y^1 на первом временном шаге.
2. Для тестирования реализации алгоритма воспользуемся методом пробных функций. Выберем $u_e(t) = ct + d$. Найти c и d из начальных условий. Вычислить соответствующую функцию источника f. Покажите, что u_e является точным решением соответствующей разностной схемы.
3. Используйте `sympy` для выполнения символьных вычислений из пункта 2. Ниже представлен каркас такой программы:

```
import sympy as sym
V, t, U, omega, tau = sym.symbols('V t U omega tau')  # глобальные символы
f = None  # глобальная переменная для функции источника ОДУ

def ode_source_term(u):
```

```
    """
    Возвращает функцию источника ОДУ, равную u'' + omega**2*u.
    u --- символьная функция от t."""
    return sym.diff(u(t), t, t) + omega**2*u(t)

def residual_discrete_eq(u):
    """
    Возвращает невязку разностного уравнения на заданной u.
    """
    R = ...
    return sym.simplify(R)

def residual_discrete_eq_step1(u):
    """
    Возвращает невязку разностного уравнения на первом шаге
    на заданной u.
    """
    R = ...
    return sym.simplify(R)

def DtDt(u, tau):
        """
        Возвращает вторую разностную производную от u.
        u --- символьная функция от t.
    """
    return ...

def main(u):
        """
        Задавая некоторое решение u как функцию от t, используйте метод
        пробных функций для вычисления функции источника f и проверьте
        является ли u решением и разностной задачи.
    """
    print '=== Проверка точного решения: %s ===' % u
    print "Начальные условия u(0)=%s, u'(0)=%s:" % \
          (u(t).subs(t, 0), sym.diff(u(t), t).subs(t, 0))

    # Метод пробных функций требует подбора f
    global f
    f = sym.simplify(ode_lhs(u))

    # Невязка разностной задачи (должна быть 0)
    print 'residual step1:', residual_discrete_eq_step1(u)
    print 'residual:', residual_discrete_eq(u)

def linear():
    main(lambda t: V*t + U)

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

Chapter 2

Конечно-разностные схемы для волнового уравнения

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:

2.1 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:

$$\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2}, \quad x \in (0, l), \quad t \in (0, T], \tag{2.1}$$

$$u(x, 0) = I(x), \quad x \in [0, l], \tag{2.2}$$

$$\frac{\partial u(x, 0)}{\partial t} = 0, \quad x \in [0, l], \tag{2.3}$$

$$u(0, t) = 0, \quad t \in (0, T], \tag{2.4}$$

$$u(l, t) = 0, \quad t \in (0, T]. \tag{2.5}$$

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

Уравнение (2.1) 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 (2.2) начальную форму струны, а

условие (2.3) означает, что начальная скорость струны равна нулю. Кроме того уравнение (2.1) дополняется граничными условиями (2.4) и (2.5). Эти два условия означают, что струна закреплена на концах, т.е. перемещения равны нулю.

Перейдем к построению конечно-разностной аппроксимации задачи (2.1) – (2.5).

2.1.1 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 переменному 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. рисунок 2.1)

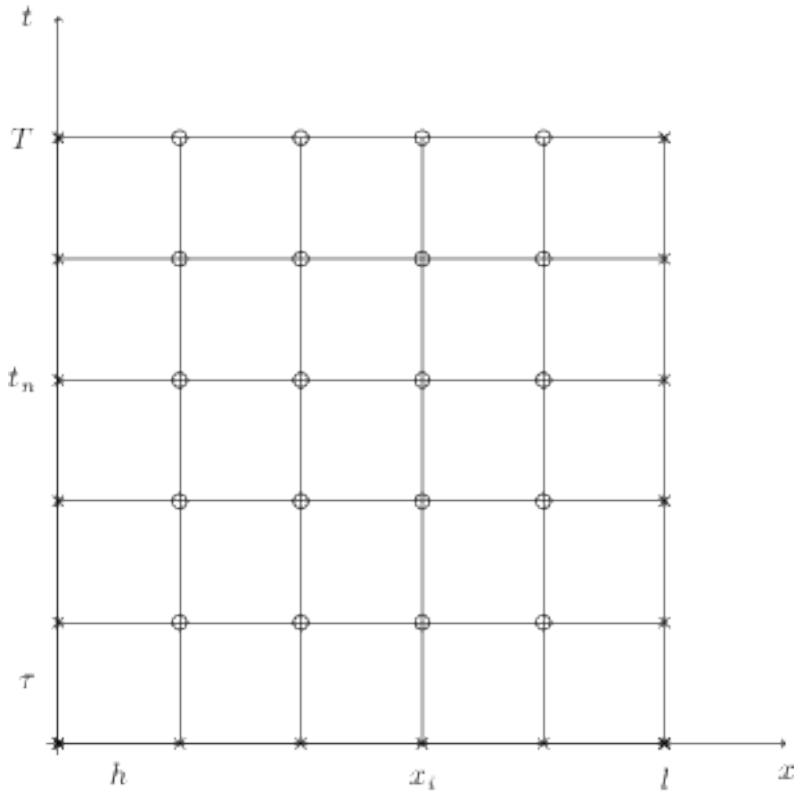


Figure2.1.: Пространственно-временная сетка $\omega_{h\tau}$

x.. index:: Узлы Узлы; граничные Узлы; внутренние

Узлы (x_i, t_n), принадлежащие отрезкам $I_0 = \{0 \le x \le l,\ t = 0\}$, $I_1 = \{x = 0,\ 0 \le t \le T\}$, $I_1 = \{x = l,\ 0 \le t \le T\}$ называются *граничными узлами* сетки $\omega_{h\tau}$, а остальные узлы — *внутренними*. На рисунке 2.1 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 (2.1), это пятиточечный шаблон, изображенный на рисунке 2.2. 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.

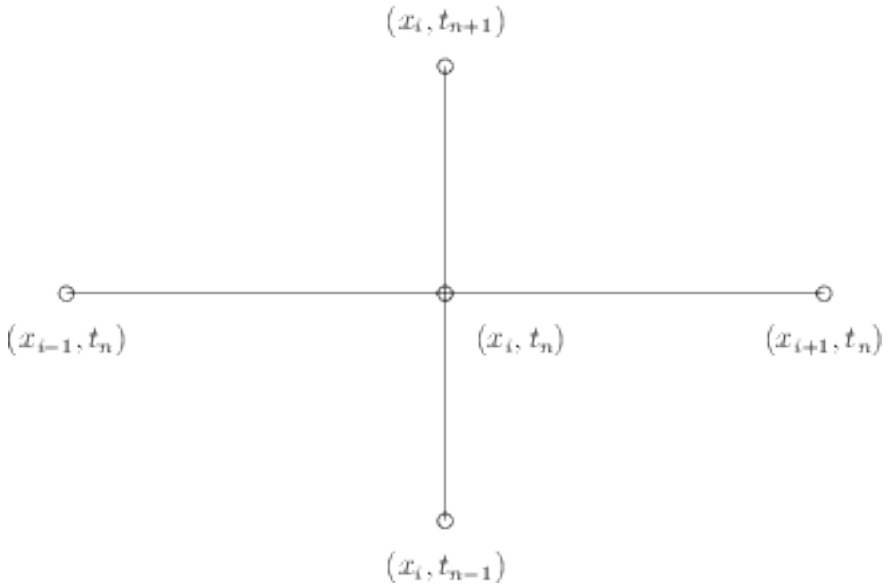


Figure2.2.: Минимальный шаблон трехслойной разностной схемы

2.1.2 Разностная схема

Простейшей разностной аппроксимацией уравнения (2.1) и граничных условий (2.4) и (2.5) является следующая система уравнений:

$$\begin{gathered}\frac{y_i^{n+1} - 2y_i^n + y_i^{n-1}}{\tau^2} = \frac{y_{i+1}^n - 2y_i^n + y_{i-1}^n}{h^2}, \\ i = 1, 2, \ldots, N-1,\ n = 1, 2, \ldots, K,\end{gathered} \tag{2.6}$$

$$y_0^{n+1} = y_N^{n+1} = 0, \quad n = 0, 1, \ldots, K-1. \tag{2.7}$$

Разностное уравнение (2.6) имеет второй порядок погрешности аппроксимации по τ и по h. Решение y_i^{n+1} выражается явным образом через значения на предыдущих слоях:

$$\begin{gathered}y_i^{n+1} = 2y_i^n - y_i^{n-1} + \gamma^2(y_{i+1}^n - 2y_i^n + y_{i-1}^n), \\ i = 1, 2, \ldots, N-1,\ n = 1, 2, \ldots, K-1.\end{gathered} \tag{2.8}$$

Здесь мы ввели параметр

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

который называют *числом Куранта*.

Для начала счета по (2.8) должны быть заданы значения y_i^0, y_i^1, $i = 0, 1, \ldots, N$. Из первого начального условия (2.2) сразу получаем

$$y_i^0 = I(x_i), \quad i = 0, 1, \ldots, N. \tag{2.9}$$

2.1.3 Аппроксимация второго начального условия

Простейшая замена второго начального условия (2.3) уравнением $(y_i^1 - y_i^0)/\tau = 0$ имеет лишь первый порядок аппроксимации по τ. Поскольку уравнение (2.6) аппроксимирует уравнение (2.1) со вторым порядком, желательно, чтобы и разностное начальное условие также имело второй порядок аппроксимации. Построим такую аппроксимацию. Уравнение

$$\frac{y_i^1 - y_i^{-1}}{2\tau} = 0, \tag{2.10}$$

аппроксимирует уравнение $\frac{\partial u}{\partial t} = 0$ со вторым порядком. Чтобы найти значения y_i^{-1} запишем уравнение (2.6) при $n = 0$:

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

Из (2.10) имеем $y^{-1} = y^1$. Отсюда получаем

$$y_i^1 = y_i^0 + \frac{\gamma^2}{2}\left(y_{i+1}^0 - 2y_i^0 + y_{i-1}^0\right). \tag{2.11}$$

Совокупность уравнений (2.6), (2.7), (2.9) и (2.11) составляет разностную схему, аппроксимирующую исходную задачу (2.1) – (2.5).

2.1.4 Вычислительный алгоритм

Теперь мы можем сформулировать вычислительный алгоритм:

1. Вычисляем y_i^0, используя (2.9).
2. Вычисляем y_i^1, используя (2.11) и задаем граничные условия (2.7) при $n = 0$.
3. Для всех временных слоев $n = 1, 2, \ldots, K-1$
   (a) находим y_i^{n+1}, используя (2.8)
   (b) задаем граничные условия (2.7).

2.1.5 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

```
# 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


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])

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])
```

```
# 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
```

2.2 Верификация программной реализации

Прежде чем реализовывать алгоритм, удобно добавить в уравнение *(1.1)* слагаемое, описывающее источник (правую часть), что даст свободу в выборе тестовых задач для верификации алгоритма.

2.2.1 Неоднородное уравнение

Рассмотрим следующую смешанную задачу для неоднородного волнового уравнения:

$$\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2} + f(x,t), \quad x \in (0,l), \quad t \in (0,T], \tag{2.12}$$

$$u(x,0) = I(x), \quad x \in [0,l], \tag{2.13}$$

$$\frac{\partial u(x,0)}{\partial t} = V(x), \quad x \in [0,l], \tag{2.14}$$

$$u(0,t) = 0, \quad t \in (0,T], \tag{2.15}$$

$$u(l,t) = 0, \quad t \in (0,T], \tag{2.16}$$

Аппроксимируя задачу (**??**) – (2.16) (аналогично случаю однородного уравнения) разностной схемой второго порядка аппроксимации на сетке $\omega_{h\tau}$, получим рекуррентное соотношение

$$\begin{gathered} y_i^{n+1} = 2y_i^n - y_i^{n-1} + \gamma^2(y_{i+1}^n - 2y_i^n + y_{i-1}^n) + \tau^2 f_i^n, \\ i = 1,2,\ldots,N-1, \ n = 1,2,\ldots,K-1. \end{gathered} \tag{2.17}$$

Кроме того аппроксимируя начальное условие (2.14) 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

$$y_i^1 = y_i^0 + \frac{\gamma^2}{2}\left(y_{i+1}^0 - 2y_i^0 + y_{i-1}^0\right) + \frac{\tau^2}{2} f_i^0. \tag{2.18}$$

2.2.2 Использование аналитического решения

Многие волновые задачи описывают синусоидальные по времени и пространству. Например, исходная задача *(1.1) – (1.5)* допускает точное решение

$$u_e(x,t) = A\sin\frac{\pi x}{l}\cos\frac{\pi ct}{l} \tag{2.19}$$

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 (2.19) 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.

2.2.3 Пробные функции

Преимущество использования метода пробных функций заключается в том, что мы можем тестировать все варианты в задаче (**??**) – (2.16). 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 (**??**), 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

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

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:

$$\begin{aligned} E = \|e_i^n\|_{\ell^2} = \left(\sum_{n=0}^{K} \tau \sum_{i=0}^{K} (e_i^n)^2\right)^{1/2}, \\ e_i^n = u_e(x_i, t_n) - y_i^n. \end{aligned} \tag{2.20}$$

$$E = \|e_i^n\|_{\ell^\infty} = \max_{i,n} |e_i^n|. \tag{2.21}$$

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 ($n = K$):

$$\begin{aligned} E = \|e_i^K\|_{\ell^2} = \left(\sum_{i=0}^{K} (e_i^K)^2\right)^{1/2}, \\ e_i^K = u_e(x_i, t_K) - y_i^K. \end{aligned} \tag{2.22}$$

$$E = \|e_i^K\|_{\ell^\infty} = \max_{0 \le i \le N} |e_i^K|. \tag{2.23}$$

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

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

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.

2.2.4 Построение точного решения разностной схемы

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):

$$u_e(x,t) = x(l-x)(1+0.5t), \tag{2.24}$$

которое дает $f(x,t) = 2(1+t)c^2$. Это решение удовлетворяет однородным граничным условиям (2.15) и (2.15), а также начальным условиям (2.13) с $I(x) = x(l-x)$ и (2.14) с $V(x) = 0.5x(l-x)$.

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

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

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

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 (2.18).

Таким образом, точное решение дифференциальной задачи (2.24) 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.

Примечание: 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, $\gamma = 1$!

2.3 Программная реализация

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

Основной вычислительный алгоритм представленный в пунктах *Вычислительный алгоритм* и *Эскиз программной реализации* 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.

2.3.1 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.

2.3.2 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]

            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
```

2.3.3 Верификация: точное решение — полином второй степени

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

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

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

2.3.4 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 u
    animate=True,              # Расчет с анимацией?
    tool='matplotlib',         # 'matplotlib' или 'scitools'
    solver_function=solver,    # Функция, реализующая алгоритм расчета
    ):
    """Запуск солвера и визуализации u 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 удобным. 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` `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 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: Flasg, 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 > 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 > 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.

2.3.5 Запуск варианта расчета

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

$$I(x) = \begin{cases} ax/x_0, & x < x_0, \\ a(l-x)/(l-x_0), x \geq x_0. \end{cases} \tag{2.25}$$

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`.

2.3.6 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

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

Отбрасывая черту сверху переменных, приходим к безразмерному волновому уравнению

$$\frac{\partial^2 u}{\partial t^2} = \frac{\partial^2 u}{\partial x^2}, \tag{2.26}$$

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 (2.25) 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 = 1 = 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 $u(x,t) = A\cos(kx)\sin(omegat)$, 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 $\tilde{P} = 2/n$ $(n > 1)$.

2.4 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.

2.4.1 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 $(u_0, u_1, \ldots, u_{n-1})$ из массива, в котором элементы сдвинуты на один индекс вперед (см. рисунок 2.3). 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*.

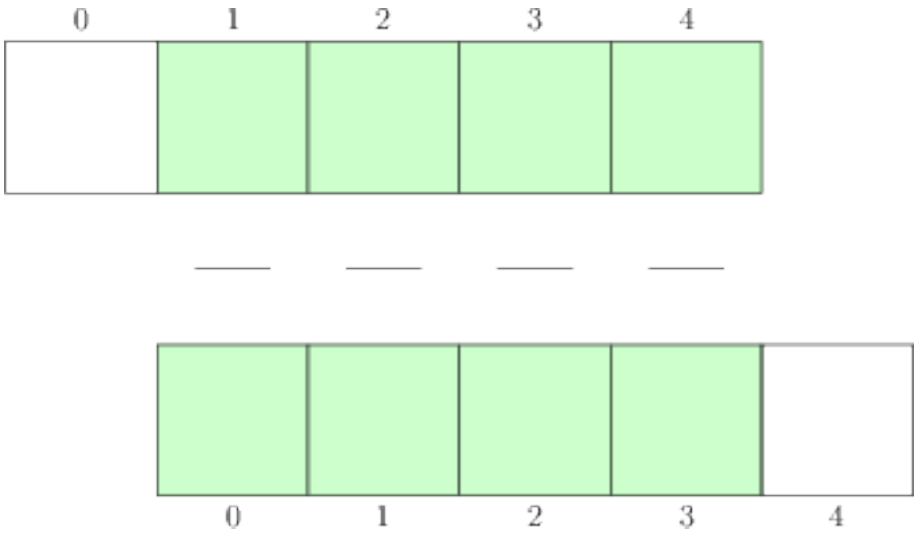


Figure2.3.: Иллюстрация вычитания срезов двух массивов

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.

2.4.2 Конечно-разностные схемы, выраженные в срезах

Перейдем к векторизации вычислительного алгоритма, математическое описание которого дано в разделе *Вычислительный алгоритм*, а программная реализация описана в разделе *Функция-солвер*. 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).

2.4.3 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, ...)
```

2.4.4 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 *Визуализация: анимация решения*. 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 u
    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.

2.4.5 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.

Осторожно: 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`.

2.5 Обобщения

2.5.1 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.

Граничные условия Неймана

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

$$\frac{\partial u}{\partial \mathbf{n}} \equiv \mathbf{n} \cdot \nabla u = 0, \tag{2.27}$$

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

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

Аппроксимация производной на границе

Построим аппроксимацию граничного условия (2.27) со вторым порядком аппроксимации по пространственной переменной. Для этого воспользуемся центральной разностной производной:

$$y^n_{\mathring{x},0} = \frac{y^n_1 - y^n_{-1}}{2h} = 0. \tag{2.28}$$

Проблема заключается в том, что y_{-1}^n не является расчетным значением, так как задано в нерасчетном узле не принадлежащем сетке. Однако, если мы объединим (2.28) с разностным уравнением *(1.8)*, записанной в узле $i = 0$:

$$y_0^{n+1} = 2y_0^n - y_0^{n-1} + \gamma^2(y_1^n - 2y_0^n + y_{-1}^n), \tag{2.29}$$

мы можем исключить фиктивное значение y_{-1}^n. Учитывая (2.28), имеем $y_{-1}^n = y_1^n$. Подставив последнее в (2.29), получим модифицированное уравнение в граничной точке y_0^{n+1}:

$$y_0^{n+1} = 2y_0^n - y_0^{n-1} + 2\gamma^2(y_1^n - 2y_0^n) \tag{2.30}$$

На рисунке 2.4 представлен шаблон схемы на левой границе области с учетом аппроксимации условия Неймана.

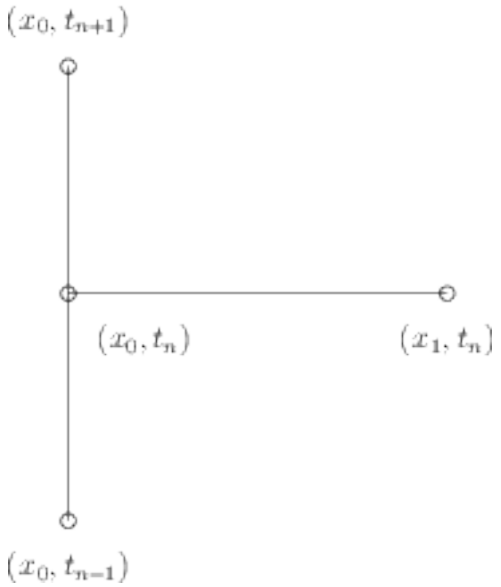


Figure2.4.: Модифицированный шаблон на левой границе для аппроксимации условия Неймана

Аналогично, получаем апрокссимацию условия (2.27) на правой границе $x = l$:

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

Объединяя последнее с разностным уравнением *(1.8)* при $i = N$ получим модифицированное уравнение на правой границе:

$$y_N^{n+1} = 2y_N^n - y_N^{n-1} + 2\gamma^2(y_{N-1}^n - y_N^n).$$

Кроме того, на границах нужно построить модификацию разностного уравнения *(1.11)* для вычисления значений на первом временном шаге.

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_{-1}^n на y_1^n на левой границе и y_{N+1}^n на y_{N-1}^n на правой границе. Эти

наблюдения могут легко 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.

Обозначение множеств индексов

Для того, чтобы улучшить математическую запись и программную реализацию, полезно ввести обозначения для множеств индексов. Это означает, что мы будем писать $x, \in \mathcal{I}_x$ 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:

Notation	Python
$\mathcal{I}_x$	`Ix`
$\mathcal{I}'_x$	`Ix[0]`
$\mathcal{I}_x^{-\infty}$	`Ix[-1]`
$\mathcal{I}^-_x$	`Ix[:-1]`
$\mathcal{I}^+_x$	`Ix[1:]`
$\mathcal{I}^)_x$	`Ix[1:-1]`

Почему полезны множества индексов

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

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

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
```

Примечание: 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.

Верификация реализации граничных условий Неймана

Перейдем к вопросу тестирования реализации условий Неймана. Функция `solver` сценария `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.

Реализация граничных условий Неймана с использованием мнимых ячеек

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

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

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)    # Массив с решением на новом слое n+1
    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]) + \
```

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:

```
        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`.

Осторожно: 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]`.

2.5.2 Переменная скорость распространения волны

Следующее обобщение одномерного волнового уравнения *(1.1)* или *(2.1)* — 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.

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:

$$\frac{\partial^2 u}{\partial t^2} = \frac{\partial}{\partial x}\left(k(x)\frac{\partial u}{\partial x}\right) + f(x,t). \tag{2.31}$$

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

$$\begin{aligned} \frac{a_{i+1} - a_i}{h} &= k'(x_i) + O(h^2), \\ \frac{a_{i+1} + a_i}{2} &= k(x_i) + O(h^2). \end{aligned} \tag{2.32}$$

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

$$a_i = k_{i-1/2} = k(x_i - 0.5h), \tag{2.33}$$

$$a_i = \frac{k_{i-1} + k_i}{2}, \tag{2.34}$$

$$a_i = 2\left(\frac{1}{k_{i-1}} + \frac{1}{k_i}\right)^{-1} \tag{2.35}$$

Выражение (2.34) — среднее арифметическое значений коэффициента в соседних узлах и часто используется для гладких коэффициентов, среднее гармоническое (2.35) часто используется при аппроксимации коэффициентов с сильно меняющимися значениями.

Правую часть $f(x,t)$ уравнения (2.31) аппроксимируем следующим образом

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

Таким образом, мы можем аппроксимировать уравнение (2.31) на сетке $\omega_{h\tau}$ следующей разностной схемой:

$$y_{\bar{t}t} = (ay_{\bar{x}})_x + \varphi, \quad (x,t) \in \omega_{h\tau}, \tag{2.36}$$

Осталось выразить из уравнения (2.36) значение y_i^{n+1}:

$$\begin{aligned} y_i^{n+1} =& 2y_i^n - y_i^{n-1} \\ &+ \frac{\tau^2}{h^2}\left(a_{i+1}(y_{i+1}^n - y_i^n) - a_i(y_i^n - y_{i-1}^n)\right) \\ &+ \tau^2\varphi_i^n \end{aligned} \tag{2.37}$$

Условия Неймана и переменные коэффициенты

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

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

Записывая разностную схему (2.37) в узле $i = N$ и учитывая, что $y_{N+1} = y_{N-1}$, получим

$$\begin{aligned} y^{n+1}_N &= 2y^n_N - y^{n-1}_N \\ &\quad + \frac{\tau^2}{h^2}\left(a_{N+1}(y^n_{N+1} - y^n_N) - a_N(y^n_N - y^n_{N-1})\right) + \tau^2\varphi^n_N \\ &= 2y^n_N - y^{n-1}_N + \frac{\tau^2}{h^2}\left((a_{N+1} + a_N)(y_{N-1} - y_N)\right) + \tau^2\varphi^n_i \\ &\approx 2y^n_N - y^{n-1}_N + 2\frac{\tau^2}{h^2}\left(a_{N+1/2}(y_{N-1} - y_N)\right) + \tau^2\varphi^n_N. \end{aligned} \tag{2.38}$$

Здесь мы использовали условия (2.32) и $a_{N+1/2} = k(x_N)$. Кроме того вместо $a_{N+1/2}$ можно использовать a_N.

Выражение (2.38) с a_N вместо $a_{N+1/2}$ можно записать в форме

$$a_N y^n_{x,N} + \frac{h}{2} y^n_{\bar{t}t,N} = \frac{h}{2}\varphi^n_N$$

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

2.6 Разработка общего солвера для одномерного волнового уравнения с переменными

Сценарий `wave1d_dn_vc.py` — это достаточно общий код, который предназначен для решения следующей начально-краевой задачи:

$$\frac{\partial^2 u}{\partial t^2} = \frac{\partial}{\partial x}\left(c^2(x)\frac{\partial u}{\partial x}\right) + f(x,t), \quad x \in (0,l),\ t \in (0,T], \tag{2.39}$$

$$u(x,0) = I(x), \quad x \in [0,l], \tag{2.40}$$

$$\frac{\partial u(x,0)}{\partial t} = V(x), \quad x \in [0,l], \tag{2.41}$$

$$u(0,t) = \mu_1(t) \quad \text{или} \quad \frac{\partial u(0,t)}{\partial x} = 0, \tag{2.42}$$

$$u(l,t) = \mu_2(t) \quad \text{или} \quad \frac{\partial u(l,t)}{\partial x} = 0, \tag{2.43}$$

Здесь присутствует только одно нововведение — переменные условия Дирихле. Их программная реализация тривиальна:

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

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

Функция `solver` в данном сценарии является естественным обобщением простейшей функции `solver` из `wave1d_v.py`, расширенной граничными условиями Неймана, переменными условиями Дирихле, а также переменной скоростью волны. Файл `wave1d_dn_vc.py` снабжен комментариями, которые поясняют реализацию этих расширений и мы не будем их приводить здесь.

Векторизация реализована только в цикле по времени, а не для расчета начальных условий, так как при они требуют незначительных затрат по сравнению с расчетом большого временного интервала.

Следующий пункты поясняют некоторые более продвинутые методы программирования, использованные в представленном сценарии.

2.6.1 Функция действий пользователя как класс

Полезная особенность сценария `wave1d_dn_vc.py` — это определение функции `user_action` в качестве класса. Эта часть сценария может потребовать некоторого пояснения.

Программный код

Класс, приспособленный для построения графиков, очистки файлов, создания видео файлов, так как делается в функции `wave1d_vc.viz`, можно реализовать следующим образом:

```
class PlotAndStoreSolution:
    """
    Класс для функции user_action в солвере.
    Визуализация и сохранение только приближенного решения.
    """
    def __init__(
        self,
        casename='tmp',     # Префикс в именах файлов
        umin=-1, umax=1,    # Фиксированные границы оси y
        pause_between_frames=None,  # Скорость воспроизведения
        backend='matplotlib',       # или 'gnuplot' или None
        screen_movie=True, # Показывать изменение решения на экране?
        title='',           # Дополнительная информация в заголовке
        skip_frame=1,       # Пропуск каждого skip_frame кадра
        filename=None):     # Имя файла, содержащего решение
        self.casename = casename
        self.yaxis = [umin, umax]
        self.pause = pause_between_frames
        self.backend = backend
        if backend is None:
            # Используем 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:
            # Сохраняем сетку по времени, когда u записано в файл
            self.t = []
            filenames = glob.glob('.' + self.filename + '*.dat.npz')
            for filename in filenames:
                os.remove(filename)

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

    def __call__(self, u, x, t, n):
            """
            Функция обратного вызова user_action, вызываемая функцией solver:
        Сохранить решение, построить график на экране и сохранить в файл.
        """
            # Сохранение решения u в файл, используя numpy.savez
            if self.filename is not None:
                    name = 'u%04d' % n
                    kwargs = {name: u}
                    fname = '.' + self.filename + '_' + name + '.dat'
                    np.savez(fname, **kwargs)
                    self.t.append(t[n])  # сохранение соответствующего значения времени
                    if n == 0:           # сохранение x
                            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 с помощью 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 новое решение
                            self.lines[0].set_ydata(u)
                            self.plt.legend(['t=%.3f' % t[n]])
                            self.plt.draw()
            else:
                    # анимация с помощью 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)  # начальные условия отображаются в течение 2 сек.
            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, перемещаются все кадры
            с графиками в этот подкаталог, и генерируется index.html для
            просмотра видео в браузере (как последовательность PNG файлов).
        """
            # Создаем HTML в подкаталоге
            directory = self.casename
            if os.path.isdir(directory):
                    shutil.rmtree(directory)
            os.mkdir(directory)

            for filename in glob.glob('frame_*.png'):
                    os.rename(filename, os.path.join(directory, filename))
            os.chdir(directory)
            fps = 4 # кадров в секунду
            if self.backend is not None:
                    from scitools.std import movie
                    movie('frame_*.png', encoder='html',
                          output_file='index.html', fps=fps)

            # Создаем видео в других форматах: Flash, Webm, Ogg, MP4
            codec2ext = dict(flv='flv', libx264='mp4', libvpx='webm',
                             libtheora='ogg')
            filespec = 'frame_%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)
            os.chdir(os.pardir)  # возвращаемся в родительский каталог

    def close_file(self, hashed_input):
            """
            Объединяем все файлы, созданные numpy.savez в один архив.
        hashed_input --- строка, отражающая входные данные для данного
            расчета (создана функцией solver).
            """
            if self.filename is not None:
                    np.savez('.' + self.filename + '_t.dat',
                             t=np.array(self.t, tauype=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
```

Разбор кода

Класс `PlotAndStoreSolution` поддерживает построение графиков с помощью Matplotlib (`backend=None`) или SciTools (`backend=matplotlib` или `backend=gnuplot`) для максимальной гибкости.

Конструктор `__init__` демонстрирует, как можно достаточно гибко импортировать инструмент построения графиков: либо `scitools.easyviz.gnuplot_` или `scitools.easyviz.matplotlib_` (отметим знак подчеркивания в конце — он обязателен). С помощью параметра `screen_movie` мы можем запретить показ всех кадров на экране. В качестве альтернативы, для медленно меняющихся графиков с мелкой сеткой, можно установить параметр `skip_frame=10`, что означает, что на экране будет показан каждый десятый кадр.

Метод `__call__` делает так, что экземпляр класса `PlotAndStoreSolution` ведет себя как функция. Таким образом мы можем передавать экземпляр класса, скажем `p`, в качестве аргумента `user_action` функции `solver`, и любое обращение к `user_action` будет вызывать `p.__call__`. Метод `__call__` строит график решения на экране, сохраняет графики и решение в файлы для дальнейшего использования.

2.6.2 Распространение импульса в двух средах

Функция `pulse` из `wave1d_dn_vc.py` иллюстрирует движение волны в неоднородной среде, т.е. когда c меняется в зависимости от x. Можно задать интервал, где скорость волны уменьшается в `slowness_factor` раз (или возрастает, если задать этот коэффициент меньшим единицы).

Доступны четыре типа начальных условий:

1. прямоугольный импульс (`plug`);
2. функция Гаусса (`gaussian`);
3. «шапочка», содержащая один период функции косинус (`cosinehat`);
4. половина периода косинуса (`half-cosinehat`).

Эти начальные условия импульсной формы можно разместить посередине (`loc='center'`) или около левой границы (`loc='left`) области моделирования. В случае расположения импульса посередине, он распадается на два импульса с амплитудами, равными половине начальной, движущимися в противоположных направлениях. В случае расположения импульса слева (его пик находится в точке $x = 0$) и использования условия симметричности $\partial u/\partial x = 0$, образуется только импульс движущийся вправо.

Функция `pulse` — удобный инструмент для использования разных форм волны и положений сред с разными скоростями распространения волны.

Ниже представлен программный код этой функции:

```
def pulse(gamma=1,            # максимум числа Куранта
          N=200,
          animate=True,
          version='vectorized',
          T=2,              # конечное время
          loc='left',       # расположение начальных условий
          pulse_tp='gaussian',  # pulse/init.cond.
          slowness_factor=2, # скорость волны в правой среде
          medium=[0.7, 0.9], # область правой среды
          skip_frame=1,
          sigma=0.05,
          ):
        """
        Разные начальные данные в форме пика на [0,1].
        Скорость распространения волны возрастает на slowness_factor
        внутри среды. Параметр loc может принимать значения 'center' или 'left',
    в зависимости от того, где начальный пик располагается.
    Параметр sigma задает ширину пика.
    """
        # Используем параметры масштабирования: l=1 для длины отрезка, c_0=1
        # для скорости волны вне среды.
        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):
                        # One period of a cosine
                        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('Неправильный pulse_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_N%s_sf%s' % \
                   (pulse_tp, N, slowness_factor)
        action = PlotMediumAndSolution(
                medium, casename=casename, umin=umin, umax=umax,
                skip_frame=skip_frame, screen_movie=animate,
                backend=None, filename='tmpdata')

        # Выбираем границу устойчивости по заданному N, наихудший случай с
        # (наименьшее значение gamma будет использовать это значение tau,
        # но меньшее значение N)
        tau = (1/N)/c_0
        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.file_close()
```

Используемый здесь класс `PlotMediumAndSolution` — это подкласс класса `PlotAndStoreSolution`, где среда с уменьшенной скоростью распространения волны `c`, что задается интервалом `medium`, визуализируется на графике.

Комментарий о выборе параметров дискретизации

Аргумент `N` в функции `pulse` не соответствует действительному количеству узлов пространственной сетки, удовлетворяющему условию $\gamma < 1$, так как функция `solver` принимает фиксированные значения τ и γ и вычисляет h в соответствии с условием устойчивости. Как видно в функции `pulse` шаг по времени τ задается при $\gamma = 1$, таким образом, если $\gamma < 1$, параметр τ останется таким же, но функция `solver` работает с большим значением h и, соответственно, с N меньшим, чем задается при вызове `pulse`. Причина этого заключается в том, что мы хотим *всегда* использовать фиксированное значение τ, чтобы кадры видео были всегда синхронизированы по времени не зависимо от значения γ (т.е. h изменяется, когда меняется число Куранта).

Для того, чтобы попробовать использовать функцию `pulse` можно сделать следующее:

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

Вместо того, чтобы остановить воспроизведение графика с помощью комбинации клавиш Ctrl-C и запуска нового расчета, возможно, проще выполнить две предыдущих команды в командной строке следующим образом:

```
Terminal> python -c 'import wave1D_dn_vc as w; w.pulse(loc="left", pulse_tp="cosinehat", N=50, every
```

2.7 Задания

2.7.1 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.

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`.

Упражнение 3: Использование класса для функции действий пользователя

Переделать *упражнение 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 *упражнение 2*).

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. *упражнение 2* или *упражнение 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.

Упражнение 5: Найти аналитическое решение волнового уравнения с затуханием

Рассмотрим волновое уравнение с затуханием

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

Цель — найти точное решение задачи с затуханием. Отправная точка — это решение типа стоячей волны (см. *упражнение 1*). Необходимо добавить множитель затухания $e^{-\beta t}$ а также обе компоненты по времени (косинус и синус):

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

Найти k, используя граничные условия $u(0,t) = u(l,t) = 0$. Затем, используя уравнение, найти константы β, ω, A и B. Поставить полную начально-краевую задачу и дать ее точное решение.

Упражнение 6: Пропустить импульс через слоистую среду

Используйте функцию `pulse` из сценария `wave1d_dn_vc.py` для исследования прохождения импульса с пиком в точке $x = 0$ через две среды с разными скоростями волны. Скорость (масштабированная) в левой среде равна 1, а в правой среде скорость равна s_f. Опишите, что происходит с Гауссовым импульсом, импульсом-«шапочкой», половиной импульса-«шапочки» и импульсом-«площадкой» для количества интервалов $N = 40, 80, 160$ и $s_f = 2, 4$. Расчет проводить до $T = 2$.

Упражнение 7: Пояснить почему возникают шумы

Эксперименты из *упражнения 6* показывают значительные численные шумы в виде нефизических волн, особенно для $s_f = 4$ и для импульса-«площадки» или половины импульса-«шапочки». Меньше всего шумы видны для Гауссова импульса. Выполните расчеты для импульса-«площадки» или половины импульса-«шапочки» с $s_f = 1$, $\gamma = 0.9, 0.25$ и $N = 40, 80, 160$. Используя численное дисперсионное соотношение поясните наблюдения.

Упражнение 8: Исследовать гармоническое усреднение в одномерном волновом уравнении

Гармоническое среднее часто используется, когда скорость распространения волны является негладкой или разрывной функцией. Даст ли гармоническое среднее меньший шум для случая $s_f = 4$ из *упражнения 7*.

Упражнение 9: Реализация периодических условий

Часто вызывает интерес изучение движения волны на большом интервале времени и большом расстоянии. Прямой подход — использовать очень большую область. Но это может привести к большим вычислениям в частях расчетной области, где волны не могут быть замечены. Более эффективный подход — позволить бегущей вправо волне покидать расчетную область и, в то же время, позволить появляться ей на левой границе. Такие граничные условия называются *периодическими*.

Граничное условие на правой границе $x = l$ — открытое граничное условие (см. *задачу 2*), что позволяет бегущей вправо волне покидать область расчета. На левой границе, $x = 0$, применим вначале расчета либо симметричные условия (см. *задачу 1*), либо открытые граничные условия.

Начальная волна распадается на две и либо отражается, либо уходит за область расчета при $x = 0$. Цель данного упражнения следовать за бегущей вправо волной. Это можно сделать с помощью периодических граничных условий. Это означает, что когда бегущая вправо волна достигает границы $x = l$, открытые граничные условия позволят ей покинуть расчетную область, в то же время мы воспользуемся граничным условием на $x = 0$, которое возвращает исходящую волну в область расчета. Это периодическое условие имеет вид $u(0) = u(l)$. Переключение с симметричного или открытого граничного условия на левой границе на периодическое граничное условие может происходить, когда $u(l, t) > \epsilon$, где $\epsilon = 10^{-14}$ может быть подходящим значением для определения, что бегущая вправо волна достигла границы $x = l$.

Открытые граничные условия можно аппроксимировать как в *задаче 2*.

Реализовать описанные граничные условия и протестировать их на двух начальных профилях: «площадка» $u(x, 0) = 1$ для $x \leq 0.1$, $u(x, 0) = 0$ для $x > 0.1$; функция Гаусса в центре области $u(x, 0) = \exp(-\frac{1}{2}(x - 0.5)^2/0.05)$. Расчетная область — единичный отрезок $[0, 1]$. Выполнить эти тесты для чисел Куранта 1 и 0.5. Скорость распространения волны считать постоянной. Создать видео для четырех вариантов. Докажите, что решение корректно.

Упражнение 10: Сравнить аппроксимации условий Неймана

Рассматриваем одномерное волновое уравнение $\frac{\partial^2 u}{\partial t^2} = \frac{\partial}{\partial x}\left(k(x)\frac{\partial u}{\partial x}\right)$. Условие Неймана при $x = 0$ можно аппроксимировать аналогично (5.7<wave-nonhom-fds-neuman>).
Цель упражнения изучить скорость сходимости при разных способах аппроксимации условий Неймана.

1. В качестве тестовой задачи возьмем $k(x) = 1 + (x + l/2)^4$. Правую часть f подобрать так, чтобы решение имело простейшую форму, скажем $u(x,t) = \cos(\pi x/l)\cos(\omega t)$ при $\omega = 1$. Провести численный эксперимент и найти скорость сходимости, используя аппроксимацию аналогичную (5.7<wave-nonhom-fds-neuman>).
2. Использовать коэффициент $k(x) = 1 + \cos(\pi x/l)$, симметричный относительно $x = 0, l$ и найти скорость сходимости при использовании a_N вместо $a_{N+1/2}$ в (5.7<wave-nonhom-fds-neuman>).
3. Третья подход основывается на более удобной, но менее точной аппроксимации, использующей направленные разности. Построить соответствующую разностную схему и реализовать ее. Выполнить численные эксперименты 1) и 2) для получения скорости сходимость.
4. Четвертый способ можно получить записав схему в виде

$$y^n_{tt,i} = \frac{1}{h}\left((k(x)u_{\mathring{x}})^n_{i+1/2} - (k(x)u_{\mathring{x}})^n_{i-1/2}\right) + f^n_i$$

и разместить границу в узле $x_{i+1/2}$, $i = N$ вместо точной физической границы. В этом случае мы можем просто положить $(k(x)u_{\mathring{x}})^n_{i+1/2} = 0$. Построить соответствующую разностную схему, используя данный подход. Реализация граничного условия в $l - h/2$ имеет порядок $O(h^2)$, но нужен ответ на вопрос как влияет на скорость сходимости смещение границы. Вычислить погрешность как обычно во внутренних узлах сетки и использовать $k(x)$ из 1) и 2).

2.7.2 Проекты

Проект 1: 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
```

2.7.3 Задачи

Задача 1: Исследовать симметричные граничные условия

Рассмотрим простейшую волну-«площадку» в области $\Omega = [-l, l]$ и

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

для некоторого числа $0 < \delta < l$. Второе начальное условие — однородное ($\frac{\partial u(x,0)}{\partial t} = 0$). Правая часть в волновом уравнении f отсутствует. Граничные условия можно задать также однородными. Решение задачи симметрично относительно $x = 0$. Это означает, что можем моделировать процесс только в половине области $[0, l]$.

1. **Покажите, что граничными условиями симметричности будут** $\frac{\partial u}{\partial x} = 0$ при $x = 0$.
2. **Выполните расчеты для полной волновой задачи в области** $[-l, l]$**.** Затем, воспользуйтесь симметричностью решения и выполните решение в половине области $[0, l]$, используя соответствующее граничное условие при $x = 0$. Сравните два решения и убедитесь что они совпадают.
3. **Докажите свойство симметричности решения,** поставив полную начально-краевую задачу и показав, что если $u(x, t)$ — решение, то $u(-x, t)$ также является решением.

Задача 2: Реализовать открытые границы

Для того, чтобы «позволить» волне покинуть расчетную область и беспрепятственно пройти границу $x = l$, можно в одномерной задаче положить следующее граничное условие, называемое *условием излучения* или *открытые граничные условия*:

$$\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} = 0. \tag{2.44}$$

Параметр c — скорость распространения волны.
Показать, что (2.44) допускает решение $u(x,t) = g_R(x - ct)$ (волна, бегущая вправо), но не $u(x,t) = g_L(x + ct)$. Это означает, что (2.44) моделирует любую бегущую вправо волну $g_R(x - ct)$, свободно проходящую через границу.
Соответствующие открытые граничные условия для $x = 0$ имеют вид

$$\frac{\partial u}{\partial t} - c\frac{\partial u}{\partial x} = 0. \tag{2.45}$$

1. Естественная идея дискретизации (2.44) в узле с номером $i = N$ — использовать центральные разностные производные по времени и пространству:

$$y^n_{\mathring{t},i} + cy^n_{\mathring{x},i} = 0, \quad i = N. \tag{2.46}$$

   Исключить фиктивные значения y^n_{N+1} используя дискретизацию уравнение в этом же узле.
   Уравнение для первого слоя, y^1_i, вообще говоря, также должно измениться, но мы можем использовать условие $y^1_N = 0$, так как еще не достигла правой границы.
2. Более удобно использовать реализацию граничных условий при $x = l$ на основе явной аппроксимации:

$$y^n_{t,i} + cy^n_{\bar{x},i} = 0, \quad i = N. \tag{2.47}$$

   Используя это уравнение, можно найти y^{n+1}_N и использовать его как граничное условие Дирихле. Однако, такая аппроксимация имеет первый порядок.
   Реализовать эту схему для волнового уравнения $\frac{\partial^2 u}{\partial t^2} = c^2\frac{\partial^2 u}{\partial x^2}$ в области $x = l$, граничное условие $\frac{\partial u}{\partial x} = 0$ при $x = 0$ и условием (2.44) при $x = l$, начальное возмущение в середине области, например, профиль-«площадка»:

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

3. Добавить возможность задавать либо условие Неймана либо открытое граничное условие на левой границе. Последнее условие аппроксимируется:

$$y^n_{t,i} - cy^n_{x,i} = 0, \quad i = 0, \tag{2.48}$$

   дающее явную аппроксимацию граничного значения y^{n+1}_0.
   Реализацию можно тестировать на Гауссовом начальном распределении:

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

   Запустить два теста:

- genindex
- genindex
- search

Выполнение упражнений и задач

В конце каждого раздела даны упражнения и задачи. Для их выполнения студенту необходимо зарегистрироваться на Github, создать репозитарий, в котором каждому разделу должна соответствовать директория, внутри которой располагаются упражнения и задачи (каждая в своей директории) из соответствующего раздела.

Примерная структура репозитария

```
repo
|- fdmforode
   |- ex-1
   |- ex-2
   |- ex-3
      .
      .
   |- pr-1
.
.
```

Алфавитный указатель