Разностные схемы для ОДУ колебаний

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

Институт математики НАН Беларуси

Mar 30, 2017

Аннотация

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. 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. Базовая модель колебательного процесса

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

$$u'' + \omega^2 u = 0, \quad t \in (0, T], \tag{1}$$

$$u(0) = U, \quad u'(0) = 0. \tag{2}$$

Здесь ω и U — заданные постоянные. Точное решение задачи (1) – (2) имеет вид

$$u(t) = U \cos \omega t, \tag{3}$$

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) – (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) – (2) описывает не только механические колебания, но и колебания в электрических цепях.

1.2. Разностная схема

При численном решении задачи (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) – (2) в точке t_n обозначим y^n.

Простейшая разностная схема для приближенного решения задачи (1) – (2) есть

$$\frac{y^{n+1} - 2y^n + y^{n-1}}{\tau^2} = -\omega^2 y^n. \tag{4}$$

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

$$\frac{y^1 - y^{-1}}{2\tau} = 0. \tag{5}$$

Для формулировки вычислительного алгоритма, предположим, что мы уже знаем значение y^{n-1} и y^n. Тогда из (4) мы можем выразить неизвестное значение y^{n+1}:

$$y^{n+1} = 2y^n - y^{n-1} - \tau^2\omega^2 y^n. \tag{6}$$

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

Очевидно, что (6) нельзя использовать при $n = 0$, так как для вычисления y^1 необходимо знать неопределенное значение y^{-1} при $t = -\tau$. Однако, из (5) имеем $y^{-1} = y^1$. Подставляя последнее в (6) при $n = 0$, получим

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

откуда

$$y^1 = y^0 - \frac{1}{2}\tau^2\omega^2 y^0. \tag{7}$$

В 1 требуется использовать альтернативный способ вывода (7), а также построить аппроксимацию начального условия $u'(0) = V \neq 0$.

1.3. Вычислительный алгоритм

Для решения задачи (1) – (2) следует выполнить следующие шаги:

1. $y^0 = U$
2. вычисляем y^1, используя (7)
3. для $n = 1, 2, \ldots$,
   (a) вычисляем y^n, используя (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]
```

1.4. 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*:

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

2. Программная реализация

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:

```
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 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
```

2.2. 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:

$$u'(t_n) \approx v^n = \frac{y^{n+1} - y^n}{2\tau} = y_t^n. \tag{8}$$

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 $v^N = y_{\bar{t}}^N$.

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
```

3. Верификация реализации алгоритма

3.1. 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
```

3.2. 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 $(t^2)^n_{tt} = 2$. 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 8.

3.3. 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:

$$\varepsilon = M\tau^r, \tag{9}$$

где C и r — постоянные (обычно неизвестные), не зависящие от τ. Формула (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 (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 (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
```

4. 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

$$\frac{d^2\bar{u}}{d\bar{t}^2} + 4\pi^2\bar{u} = 0, \quad \bar{u}(0) = 1, \quad \bar{u}'(0) = 0. \tag{10}$$

Заметьте, что в (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[1] содержит представленные в данном разделе примеры.

5. Проведение вычислительного эксперимента

На рисунке 1 представлено сравнение точного и приближенного решений безразмерной модели (10) 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.

[1] `src-fdm-for-ode/vib_undamped.py`

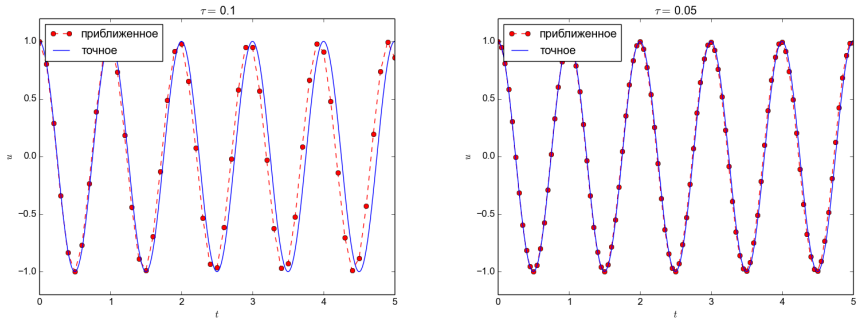


Рис. 1: Эффект от уменьшения шага вдвое

5.1. 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[2] 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]
```

[2] https://github.com/hplgit/scitools

```
    plot_manager = MovingPlotWindow(
        window_width=8*P,
        dt=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)

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=u'Сравнение с помощью Bokeh')
    # 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)


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'Шагов на период: %d' % n)
    bokeh_plot(u, t, legends, U=1, omega=omega, t_range=[0, 4*P],
               filename='bokeh.html')

if __name__ == '__main__':
    main()
#    demo_bokeh()
#    raw_input()
```

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.

5.2. 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 `ffmpeg` имеется в репозиториях Ubuntu. Можно использовать другие программы для создания видео из набора отдельных графических файлов. Для генерации 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 ImageMagick[3]:

```
Terminal> convert -delay 25 tmp_*.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

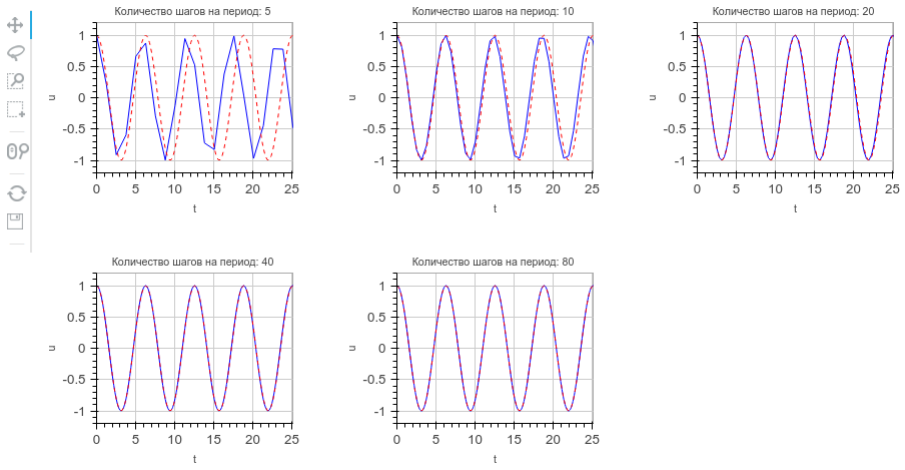
[3] `http://www.imagemagick.org`

$\tau = 0.05$, 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.

5.3. 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 Bokeh[4] 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.

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:

[4] `http://bokeh.pydata.org`

```
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=u'Сравнение с помощью Bokeh')
    # 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('Шагов на период: %d' % n)
    bokeh_plot(u, t, legends, U=1, omega=omega, t_range=[0, 4*P],
               filename='bokeh.html')
```

5.4. Практический анализ решения

Для колебательной функции, аналогичной представленной на 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 рис. 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.

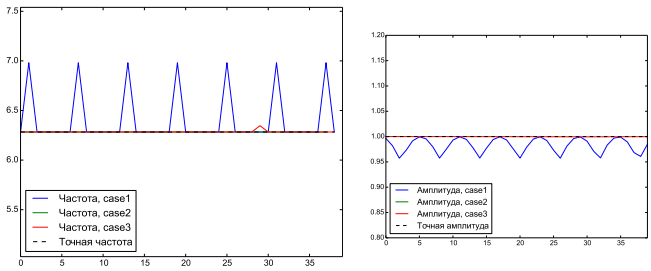


Рис. 2: Эмпирические амплитуды и угловые частоты для трех значений временного шага.

6. Анализ конечно-разностной схемы

6.1. 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 (4) — 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 (4) 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

$$
\tilde{\omega} = \pm\frac{2}{\tau}\arcsin\left(\frac{\omega\tau}{2}\right). \tag{11}
$$

Из (11) видно, что численная частота $\tilde{\omega}$ никогда не равна точной ω. Для того, чтобы понять насколько хороша аппроксимация (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

$$\tilde{\omega} = \omega \left(1 + \frac{1}{24}\omega^2\tau^2\right) + \mathcal{O}(\tau^4). \tag{12}$$

Погрешность численного значения частоты имеет второй порядок по τ и стремится к нулю при $\tau \to 0$. Из (12) видно, что $\tilde{\omega} > \omega$, так как $\omega^3\tau/24 > 0$. 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 рис. 1.

На 3 представлены графики дискретной частоты вычисленной по (11) и ее приближения (12) для $\omega = 1$.

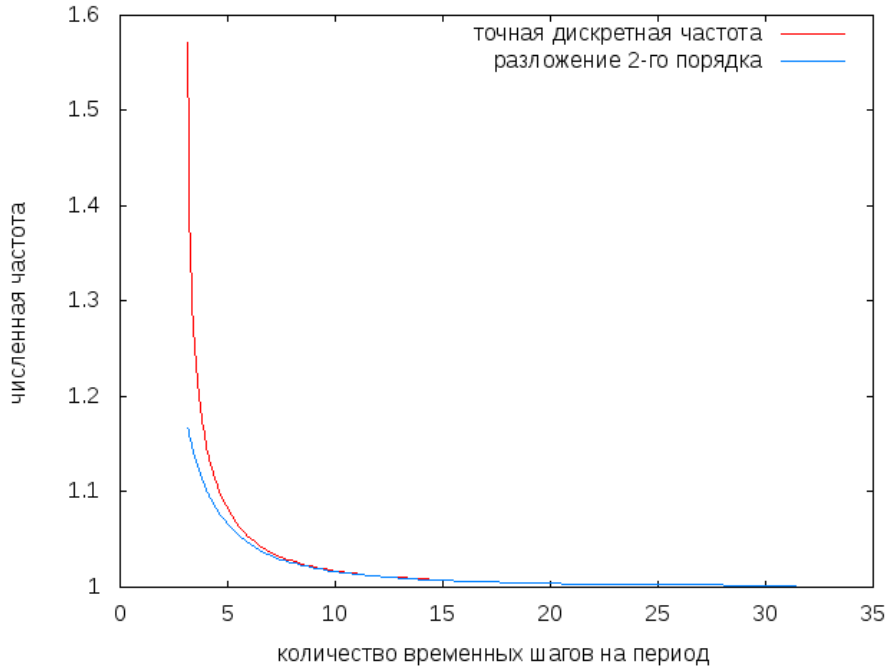


Рис. 3: Точная дискретная частота и ее разложение в ряд второго порядка.

6.2. Точное дискретное решение

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

$$y^n = U\cos(\tilde{\omega}\tau n), \quad \tilde{\omega} = \frac{2}{\tau}\arcsin\left(\frac{\omega\tau}{2}\right). \tag{13}$$

Теперь мы можем вычислить сеточную функцию погрешности:

$$e^n = u_e(t_n) - u^n = U\cos(\omega\tau n) - U\cos(\tilde{\omega}\tau n)$$
$$= -2U\sin\left(\frac{t}{2}(\omega-\tilde{\omega})\right)\sin\left(\frac{t}{2}(\omega+\tilde{\omega})\right). \quad (14)$$

Построенная сеточная функция погрешности идеальна для целей тестирования поэтому необходимо реализовать тест на основе (13), выполнив 2.

6.3. Сходимость

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

6.4. Глобальная погрешность

Проведем анализ глобальной погрешности. Разложим сеточную функцию погрешности (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.

6.5. 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 (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:

$$\frac{\omega\tau}{2} \leq 1 \rightarrow \tau \leq \frac{2}{\omega}. \tag{15}$$

Возьмем $\omega = 2\pi$ и выберем $\tau > \pi^{-1} = 0.3183098861837907$. На рис. 4 представлен результат расчета при $\tau = 0.3184$, которое незначительно отличается от критического значения: $\tau = \pi^{-1} + 9.01 \cdot 10^{-5}$.

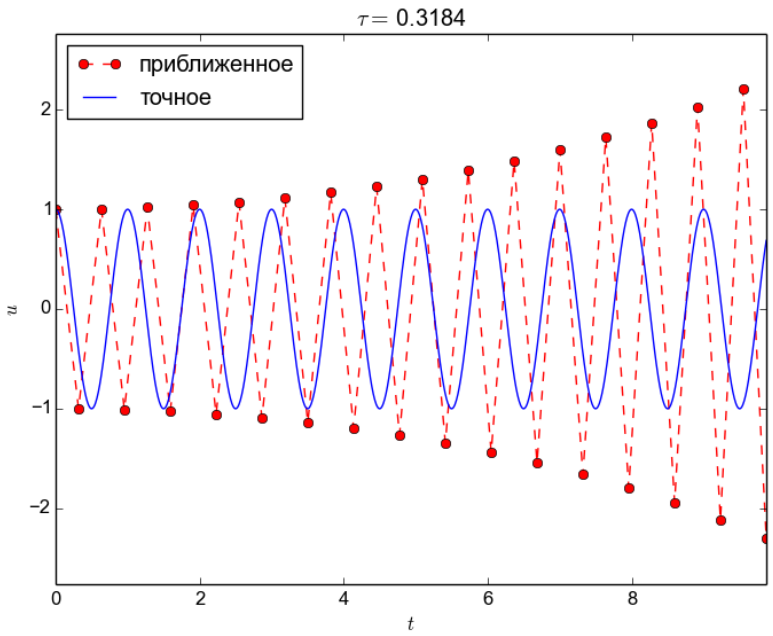


Рис. 4: Неустойчивое решение.

6.6. 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 рис. 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 рис. 5. Следовательно, временной шаг τ следует выбирать как можно меньшим, чтобы добиться поддающихся интерпретации результатов.

Выводы.

Из анализа точности и устойчивости можно сделать некоторые выводы:

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 $p \leq 2$, 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.

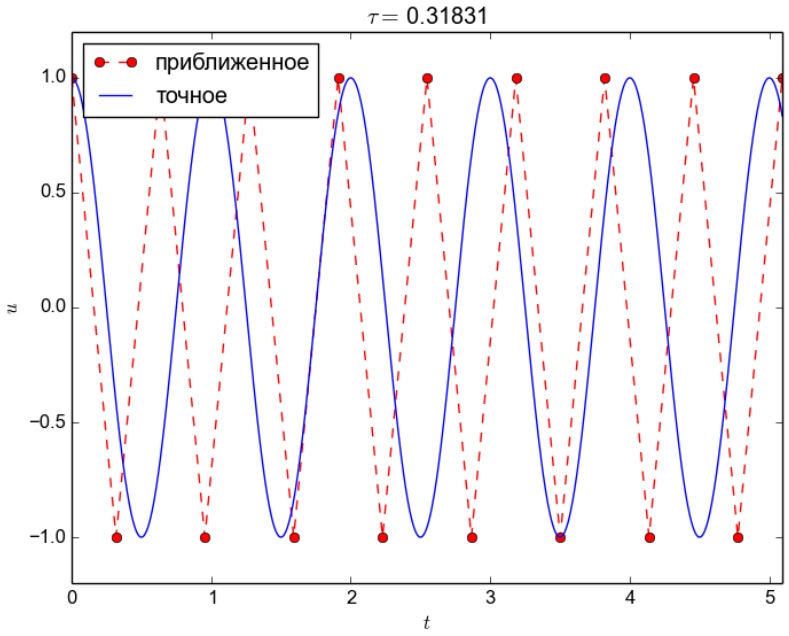


Рис. 5: Численное решение при $\tau = 2/\omega$.

7. Обобщения: затухание, нелинейные струны и внешние воздействия

Рассмотрим обобщение задачи, рассмотренной в разделе 1, учитывающее возможные затухания $f(u')$, нелинейную пружину (или сопротивление) $s(u)$ и некоторое внешнее воздействие $F(t)$:

$$mu'' + f(u') + s(u) = F(t), \quad t \in (0, T], \tag{16}$$

$$u(0) = U, \quad u'(0) = V. \tag{17}$$

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 $s(u) \sim \sin(u)$.

7.1. Разностная схема для линейного затухания

Для приближенного решения уравнения (16)–(17) в случае линейного затухания будем использовать следующую разностную схему:

$$my^n_{\bar{t}t} + by^n_{\mathring{t}} + s(y^n) = F^n, \quad n = 1, 2, \ldots, N-1, \tag{18}$$

$$y^0 = U, \quad y^0_{\mathring{t}} = \frac{y^1 - y^{-1}}{2\tau} = V. \tag{19}$$

Перепишем задачу (18) – (19)в индексной форме:

$$m\frac{y^{n+1} - 2y^n + y^{n-1}}{\tau^2} + b\frac{y^{n+1} - y^{n-1}}{2\tau} + s(y^n) = F^n, \quad n = 1, 2, \ldots, N-1, \tag{20}$$

$$y^0 = U, \quad \frac{y^1 - y^{-1}}{2\tau} = V. \tag{21}$$

Решая разностное уравнение (20) относительно неизвестной y^{n+1}, получим следующую рекуррентную формулу:

$$y^{n+1} = \frac{2my^n + (0.5\tau - m)y^{n-1} + \tau^2(F^n - s(y^n))}{m + 0.5b\tau}, \quad n = 1, 2, \ldots, N-1. \tag{22}$$

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

$$y^1 = y^0 + \tau V + \frac{\tau^2}{2m}(F^0 - s(y^0) - bV). \tag{23}$$

7.2. 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:

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

Таким образом, получим разностное уравнение

$$my_{\bar{t}t}^n + by_t^n y_{\bar{t}}^n + s(y^n) = F^n, \quad n = 1, 2, \ldots, N-1, \tag{24}$$

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

$$y^{n+1} = \frac{2my^n - my^{n-1} + by^n|y^n - y^{n-1}| + \tau^2(F^n - s(y^n))}{m + b|y^n - y^{n-1}|}. \tag{25}$$

При использовании аппроксимации второго начального условия из (19) мы получим сложное нелинейное выражение для y^1. Однако, можно построить аппроксимацию, линейную относительно y^1. Для $t = 0$ имеем $u'(0)|u'(0)| = bV|V|$. Используя это выражение в уравнении (24) и значение $u(0) = U$ при аппроксимации уравнения (16), записанного при $t = 0$, получим

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

Отсюда, учитывая второе начальное условие из (19), получим выражение для вычисления y^1

$$y^1 = y^0 + \tau V + \frac{\tau^2}{2m}(F^0 - mV|V| - s(U)). \tag{26}$$

7.3. 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 (23) для линейного затухания или (26) для квадратичного затухания;
3. для $n = 1, 2, \ldots, N-1$:
   (a) вычисляем y^{n+1}, используя (22) для линейного затухания или (25) 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 'liniear', то f(u')=b*u, если затухание 'quadratic',
    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
```

7.4. 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
```

Линейная функция как 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.

Квадратичная функция как 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.

7.5. 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[5] содержит несколько функций для визуализации решения.

7.6. 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 ок-

[5] `src-fdm-for-ode/vib.py`

на (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)
    plt.show()
```

Сценарий vib.py[6] содержит представленный выше фрагмент кода и решает модельную задачу (16)–(17). качестве примера использования vib.py[7] 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:

[6] src-fdm-for-ode/vib.py
[7] src-fdm-for-ode/vib.py

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

8. Упражнения и задачи

Упражнение 1: Использование ряда Тейлора для вычисления y^1

Альтернативный способ вывода (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) и начальное условие для производной $u'(0) = 0$, покажите, что такой способ также приведет к (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 (13) для проверки реализации функции `solver`.

Упражнение 3: Использование линейной и квадратичной функций для тестирования

Упражнение является обобщением задачи 8 на более общую задачу (16) в случаях, 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 8, 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 (12) численная частота отклоняется от точной на величину $\omega^3\tau^2/24 > 0$. Замените параметр `omega` в функции `solver` из vib_undamped.py[8] 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

$$E = \frac{u^n_{tt}}{u''(t_n)}$$

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[9] 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.

Задача 8: Использование линейной и квадратичной функций для тестирования

Рассмотрим задачу для ОДУ:

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

1. Аппроксимируем уравнение разностной схемой $y^n_{tt} + \omega^2 y^n = f^n$.
2. Вывести уравнение для нахождения приближенного решения y^1 на первом временном шаге.

[8] `src-fdm-for-ode/vib_undamped.py`
[9] `src-fdm-for-ode/vib.py`

3. 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.

4. Используйте `sympy` для выполнения символьных вычислений из пункта 2. Ниже представлен каркас такой программы:

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

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()
```

Предметный указатель