Разностные схемы для ОДУ колебаний

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

Институт математики НАН Беларуси

Mar 11, 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 $n = 1, 2, \dots$.

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 $n = 1, 2, \dots,$

(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:

$$u'(t_n) \approx v^n = \frac{y^{n+1} - y^n}{2\tau} = y^n_{\mathring{t}}. \qquad (8)$$

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
```

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 $(t^2)_{\bar{t}t}^n = 2$. 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 2.

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: $\tau_k = 2^{-k}\tau_0$, $k = 0, 1, \dots, m-1$,
- 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 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.

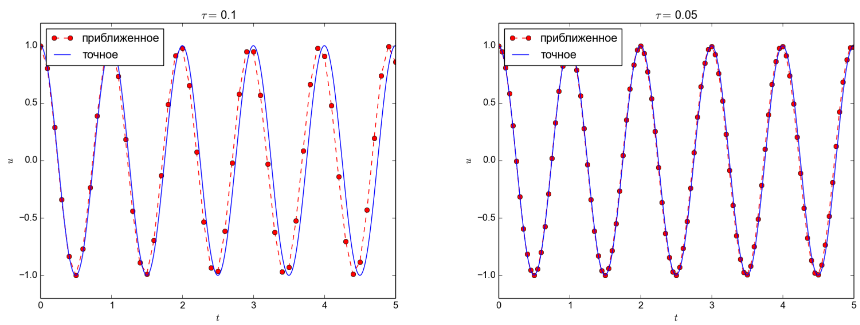


Рис. 1: Эффект от уменьшения шага вдвое

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 содержит удобный инструмент для этого: MovingPlotWindow. Ниже приведена функция, использующая данный инструмент:

```
def visualize_front(u, t, U, omega, savefig=False, skip_frames=1):
    """
    Стороится 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,
        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 репозитариях 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.

Warning.

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:

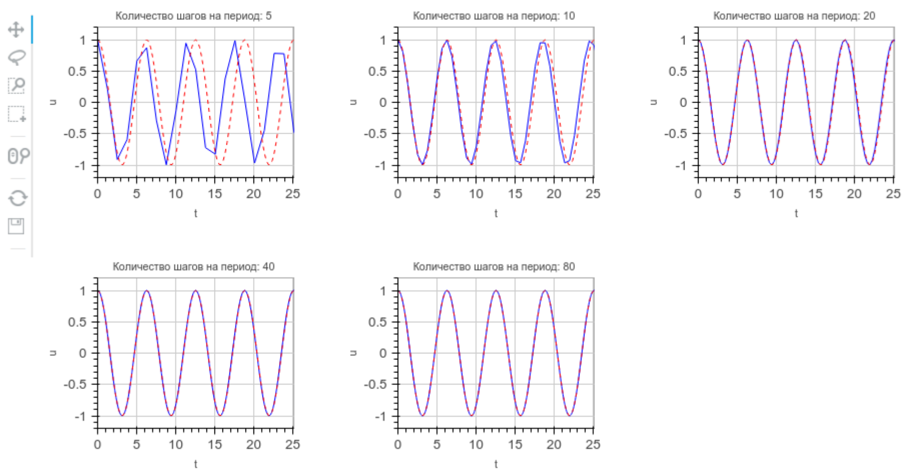
```
Terminal> convert -delay 25 tmp_*.png tmp_vib.gif
```

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 $\tau = 0.05$, 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 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:

```
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(u'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 соотвествующих 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.

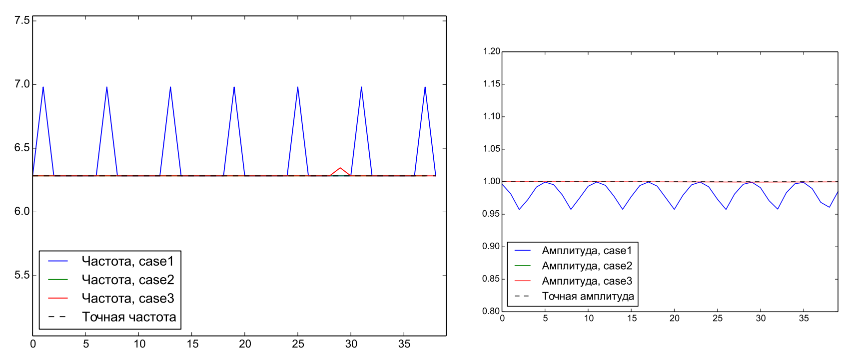


Рис. 2: Эмпирические амплитуды и угловые частоты для трех значений временного шага.

6 Упражнения и задачи

Exercise 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.

Problem 2: 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 $y^n_{\bar{t}t} + \omega^2 y^n = f^n$.
2. Вывести уравнение для нахождения приближенного решения y^1 на первом временном шаге.
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