Нестационарные задачи математической физики

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

Институт математики НАН Беларуси

May 21, 2019

Содержание

1. Смешанные задачи для нестационарных уравнений

Одним из базовых уравнений математической физики является одномерное параболическое уравнение второго порядка. В прямоугольнике

$$\bar{Q}_T = \bar{\Omega} \times [0, T], \quad \bar{\Omega} = \{x \,:\, 0 \le x \le l\}$$

рассматривается уравнение

$$\frac{\partial u}{\partial t} = \frac{\partial}{\partial x}\left(k(x)\frac{\partial u}{\partial x}\right) + f(x,t), \quad x \in \Omega,\ t \in (0, T], \tag{1}$$

дополненное граничными (первого рода)

$$u(0,t) = u(l,t) = 0, \quad t \in (0, T], \tag{2}$$

и начальными

$$u(x,0) = I(x), \quad 0, \quad x \in \bar{\Omega}, \tag{3}$$

условиями.

Для простоты изложения мы рассматриваем простейшие однородные граничными условия и зависимость коэффициента k только от пространственной переменной, причем $k(x) \geq \kappa > 0$.

Вместо условий первого рода (1) могут задаваться другие граничные условия. Например, граничные условия третьего рода:

$$-k(0)\frac{\partial u}{\partial x}(0,t) + \sigma_1(t)u(0,t) = \mu_1(t), \tag{4}$$

$$k(l)\frac{\partial u}{\partial x}(l,t) + \sigma_2(t)u(l,t) = \mu_2(t), \quad t \in (0,T] \tag{5}$$

К нестационарным уравнениям математической физики также относится гиперболическое уравнение второго порядка. В одномерном по пространству случае ищется решение уравнения

$$\frac{\partial^2 u}{\partial t^2} = \frac{\partial}{\partial x}\left(k(x)\frac{\partial u}{\partial x}\right) + f(x,t), \quad x \in \Omega,\ t \in (0,T]. \tag{6}$$

Для однозначного определения решения этого уравнения помимо граничных условий (2) задаются два начальных условия

$$u(x,0) = I(x), \quad \frac{\partial u}{\partial t}(x,0) = V(x), \quad x \in \bar{\Omega}. \tag{7}$$

2. Разностные схемы для параболического уравнения

2.1. Явная схема

Первый шаг процедуры построения разностной задачи состоит в дискретизации области $\bar{Q}_T$. Будем использовать равномерную пространственно-временную сетку

$$\bar{\omega} = \bar{\omega}_h \times \bar{\omega}_\tau, \quad \bar{\omega}_h = \{x_i = ih,\ i = 0,1,\dots,N_x\},$$

$$\bar{\omega}_\tau = \{t_n = n\tau,\ n = 0,1,\dots,N_t\}$$

Для аппроксимации уравнения (1) используем явную разностную схему

$$\frac{y^{n+1} - y^n}{\tau} = -Ay^n + \varphi^n, \quad n = 0,1,\dots,N_t. \tag{8}$$

Здесь через $y^n = y_i^n$ обозначено приближенное решение в узле сетки $(x_i, t_n) \in \omega$, сеточный оператор A определен для сеточных функций $y = 0$ при $x \in \partial\omega_h$ (граничные узлы сетки ω_h) следующим образом:

$$Ay^n = -(ay^n_{\bar{x}})_{x,i} = -\frac{1}{h}\left(a_{i+1}\frac{y^n_{i+1} - y^n_i}{h} - a_i\frac{y^n_i - y^n_{i-1}}{h}\right), \quad x_i \in \omega_h.$$

Для задания коэффициента a можно использовать выражения

$$\begin{aligned}
a_i &= k(x_i - 0.5h), \\
a_i &= 0.5(k(x_i) + k(x_{i-1})), \\
a_i &= \left(\frac{1}{h} \int_{x_{i-1}}^{x_i} \frac{dx}{k(x)} \right)^{-1}.
\end{aligned}$$

Ключевое свойство явной разностной схемы (8) заключается в том, что она легко решается. Считаем, что приближенное решение y^n на временном слое t_n известно для всех $x \in \bar{\omega}_h$. Следовательно, неизвестным в (8) является y^{n+1} для $x \in \omega_h$. Решая (8) относительно y^{n+1} получим рекуррентное соотношение

$$y_i^{n+1} = y_i^n + \frac{\tau}{h} \left(a_{i+1} \frac{y_{i+1}^n - y_i^n}{h} - a_i \frac{y_i^n - y_{i-1}^n}{h} \right) + \tau \varphi_i^n \quad (9)$$

Замечание.

В случае постоянного коэффициента $k(x) = a$ соотношение (9) запишется в виде

$$y_i^{n+1} = y_i^n + F \left(y_{i+1}^n - 2y_i^n + y_{i-1} \right) + \tau \varphi_i^n. \quad (10)$$

Здесь введен параметр (*сеточное число Фурье*)

$$F = \frac{a\tau}{h^2} \quad (11)$$

Таким образом, вычислительный алгоритм следующий:

1. Вычисляем y_i^0 для всех $i = 0, 1, \ldots, N_x$
2. Для $n = 1, 2, \ldots, N_t$:
   (a) применяем (9) для всех внутренних пространственных узлов ($i = 1, 2, \ldots, N_x - 1$)
   (b) устанавливаем граничные значения $y_i^{n+1} = 0$ для $i = 0$ и $i = N_x$.

Реализация. Файл parab1d.py[1] содержит функцию `solver_Ex_simple` для численного решения одномерного параболического уравнения с однородными граничными условиями и постоянным коэффициентом $k(x) = a$ по явной схеме:

```
def solver_Ex_simple(I, a, f, L, tau, F, T):
    """
```

[1] `src-time-dep/parab1d.py`

```
    Простейшая реализация явной разностной схемы для приближенного решения
    параболического уравнения.
    """
    cpu = 0  # Для подсчета процессорного времени

    Nt = int(round(T/float(tau)))
    t = np.linspace(0, Nt*tau, Nt+1)  # сетка по времени
    h = np.sqrt(a*tau/F)
    Nx = int(round(L/float(h)))
    x = np.linspace(0, L, Nx+1)     # сетка по пространству

    # Для уверенности шаги по пространству и времени согласуем с сетками
    h = x[1] - x[0]
    tau = t[1] - t[0]

    u = np.zeros(Nx+1)
    u_n = np.zeros(Nx+1)

    # устанавливаем начальные условия u(x,0) = I(x)
    for i in range(0, Nx+1):
        u_n[i] = I(x[i])

    for n in range(1, Nt):
        # Вычисляем приближенное решение во внутренних узлах сетки
        for i in range(1, Nx):
            u[i] = u_n[i] + F*(u_n[i+1] - 2*u_n[i] + u_n[i-1]) + tau*f(x[i], t[n])

        # Задаем граничные условия
        u[0] = 0
        u[Nx] = 0

        # Обновляем переменные перед следующим шагом
        u_n, u = u, u_n

    cpu = time.perf_counter()
    return u, x, t, cpu
```

Для ускорения расчета можно воспользоваться векторными вычислениями, т.е. заменить явный цикл

```
for i in range(1, Nx):
    u[i] = u_n[i] + F*(u_n[i+1] - 2*u_n[i] + u_n[i-1])
    + tau*f(x[i], t[n])
```

арифметикой над срезами

```
u[1:Nx] = u_n[1:Nx] + F*(u_n[2:Nx+1] - 2*u_n[1:Nx] + u[0,Nx-1])
+ tau*f(x[1:Nx], t[n])
```

или

```
u[1:-1] = u_n[1:-1] + F*(u_n[2:] - 2*u_n[1:-1] + u[0,-2])
+ tau*f(x[1:-1], t[n])
```

F — ключевой параметр при дискретизации параболического уравнения.

Параметр F — безразмерная величина, которая объединяет физический параметр задачи, a, и параметры дискретизации h и τ. Свойства явной разностной схемы зависят от величины F.

Функция `solver_Ex` реализует как скалярную так и векторизованную версии. Кроме того, функция `solver_Ex` имеет в функцию обратного вызова так что пользователь может обрабатывать решение на каждом временном слое. Функция обратного вызова имеет вид `user_action(u, x, t, n)`, где `u` — массив с решением на `n`-ном временном слое, `x` — пространственная сетка, `t` — временная сетка.

```
def solver_Ex(I, a, f, L, tau, F, T, user_action=None, version='scalar'):
    """
    Векторизованная реализация явной схемы
    """
    cpu = 0  # Для подсчета процессорного времени

    Nt = int(round(T/float(tau)))
    t = np.linspace(0, Nt*tau, Nt+1)  # сетка по времени
    h = np.sqrt(a*tau/F)
    Nx = int(round(L/float(h)))
    x = np.linspace(0, Nx*h, Nx+1)    # сетка по пространству

    # Для уверенности шаги по пространству и времени согласуем с сетками
    h = x[1] - x[0]
    tau = t[1] - t[0]

    u = np.zeros(Nx+1)
    u_n = np.zeros(Nx+1)

    # устанавливаем начальные условия u(x,0) = I(x)
    for i in range(0, Nx+1):
        u_n[i] = I(x[i])

    for n in range(0, Nt+1):
        # Вычисляем приближенное решение во внутренних узлах сетки
        if version == 'scalar':
            for i in range(1, Nx):
                u[i] = u_n[i] + F*(u_n[i+1] - 2*u_n[i] + u_n[i-1]) + tau*f(x[i], t[n])
        elif version == 'vectorized':
            u[1:Nx] = u_n[1:Nx] + F*(u_n[2:Nx+1] - 2*u_n[1:Nx] + u_n[0:Nx-1]) + tau*f(x[1:Nx], t[n])
        else:
            raise ValueError('version = {}'.format(version))

        # Задаем граничные условия
        u[0] = 0
```

```
        u[Nx] = 0

        if user_action is not None:
            user_action(u, x, t, n)

        # Обновляем переменные перед следующим шагом
        u_n, u = u, u_n

    cpu = time.perf_counter()
    return u, x, t, cpu
```

Верификация.

Точное решение дискретного уравнения. Перед тем как использовать решатели, нам нужно разработать подходящие тестовые примеры для проверки реализации алгоритма. Оказывается, что функция, линейная по времени и квадратичная по пространству, в точности удовлетворяет явной разностной схеме. Учитывая однородные граничные условия, мы можем воспользоваться решением

$$u(x,t) = 5tx(l-x).$$

Возьмем постоянный коэффициент $k(x) = \alpha$. Подставляя его в уравнение (1), получим функцию источника

$$f(x,t) = 10\alpha t + 5x(l-x)$$

Легко проверить, что, если подставить $u(x,t)$ и $f(x,t)$ в явную схему (8), то получится тождество.

Вычисление правой части легко автоматизировать, используя `sympy`:

```
import sympy as sp
x, t, a, l = sp.symbols('x t a l')
u = x*(l-x)*5*t

def pde(u):
    return sp.diff(u, t) - a*sp.diff(u, x, x)

f = sp.simplify(pde(u))
```

Используя такой подход, можно выбирать любое выражение для `u` и в случае постоянного коэффициента автоматически получать выражение для функции источника.

Для использования функций при численном решении, требуется, чтобы `u` и `f` были функциями Python. Например, точное решение должно быть функцией `u_exact(x, t)`, а правая часть `f(x, t)`. Параметры `a` и `l` в определении `u` и `f` являются символами и должны быть заменены на объекты типа `float` в функциях Python. Это можно сделать, определяя `a` и `l` как числа и подставляя их вместо символов. Соответствующий код может выглядеть следующим образом:

```
a = 0.5
L = 1.5
u_exact = sym.lambdify([x, t], u.subs('L', L).subs('a', a), modules='numpy')
f = sym.lambdify([x, t], f.subs('l', l).subs('a', a), modules='numpy')
I = lambda x: u_exact(x, 0)
```

Здесь мы также создали функцию начальных данных `I`.

Идея теста заключается в том, чтобы наше решение точно (с машинной точностью) воспроизводилось реализованным расчетным кодом. Для этих целей в файле test_solver_ex.py.py[2] мы создадим тестовые функции для сравнения точного и приближенного решений в конце временного интервала. Одну для тестирования скалярной реализации:

```
def test_scalar():
    # Определяем u_exact, f, I
    import sympy as sp
    x, t, a, L = sp.symbols('x t a L')
    u = x*(L-x)*5*t

    def pde(u):
        return sp.diff(u, t) - a*sp.diff(u, x, x)

    f = sp.simplify(pde(u))
    a = 0.5
    L = 1.5

    u_exact = sp.lambdify([x, t], u.subs('L', L).subs('a', a), modules='numpy')
    f = sp.lambdify([x, t], f.subs('L', L).subs('a', a), modules='numpy')
    I = lambda x: u_exact(x, 0)

    h = L/3.
    F = 0.5
    tau = F*h**2/a

    u, x, t, cpu = solver_Ex(I=I, a=a, f=f, L=L, tau=tau, F=F, T=2,
                             user_action=None, version='scalar')
    u_e = u_exact(x, t[-1])
    diff = abs(u_e - u).max()
    tol = 1E-14
    msg = 'Максимальная ошибка solver_Ex, scalar: {}'.format(diff)
    assert diff < tol, msg
```

Вторую для тестирования векторизованной реализации:

```
def test_vectorized():
    # Определяем u_exact, f, I
    import sympy as sp
    x, t, a, L = sp.symbols('x t a L')
    u = x*(L-x)*5*t
```

[2] `src-time-dep/test_solver_ex.py`

```python
def pde(u):
    return sp.diff(u, t) - a*sp.diff(u, x, x)

f = sp.simplify(pde(u))
a = 0.5
L = 1.0

u_exact = sp.lambdify([x, t], u.subs('L', L).subs('a', a), modules='numpy')
f = sp.lambdify([x, t], f.subs('L', L).subs('a', a), modules='numpy')
I = lambda x: u_exact(x, 0)

h = L/10.
F = 0.5
tau = F*h**2/a

u, x, t, cpu = solver_Ex(I=I, a=a, f=f, L=L, tau=tau, F=0.5, T=2,
                         user_action=None, version='vectorized')
u_e = u_exact(x, t[-1])
diff = abs(u_e - u).max()
tol = 1E-14
msg = 'Максимальная ошибка solver_Ex, vectorized: {}'.format(diff)
assert diff < tol, msg
```

Для запуска тестов можно воспользоваться утилитой `py.test`. Для этого в каталоге где собраны файлы с именами начинающимися с `test_` достаточно запустить команду

Terminal

```
py.test -s -v
```

В этом случае будут выполнены все функции с именами вида `test_*()` из всех файлов, которые также начинаются с `test_`. Тестовые функции (их имена имеют вид `test_*()`, можно создавать в файлах с любыми названиями. В этом случае или если мы хотим запустить тесты из одного файла, команда запуска тестов будет выглядеть так

Terminal

```
py.test -s -v test_solver_ex.py
```

В случае удачного прохождения тестов вывод будет примерно таким

Terminal

```
collected 2 items

test_solver_ex.py::test_solver_Ex_scalar PASSED
test_solver_ex.py::test_solver_Ex_vectorized PASSED
```

В случае провала теста будет выведено сообщение которое передается в качестве второго параметра команды `assert`.

Проверка порядка сходимости. Если выбранное решение не является точным для разностной схемы, мы можем проверить порядок сходимости. Для параболического уравнения порядки сходимости по времени и пространству разные. Для численной погрешности вида

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

имеем $r = 1, p = 2$ (C_t и C_x — неизвестные постоянные). Для упрощения, введем один параметр дискретизации δ:

$$\delta = \tau, \quad h = K\delta^{r/p},$$

где K — некоторая постоянная. Это позволяет нам выделить только один параметр дискретизации

$$E = C_t \delta^r + C_x (K\delta^{r/p})^p = \tilde{C}\delta^r, \quad \tilde{C} = C_t + C_x K^p.$$

В нашем случае $r = 1$.

Используя точное решение дифференциальной задачи, мы вычислим погрешность E на последовательности уточненных сеток и посмотрим, выполняется ли $r = 1$. Мы не будем оценивать константы C_t и C_x, так как нас не интересуют их значения.

В случае постоянного коэффициента $k(x) = \alpha$ для выбора K воспользуемся условием устойчивости явной схемы:

$$\frac{\alpha\tau}{h^2} \le \frac{1}{2} \Rightarrow h \ge \sqrt{2\alpha}\delta^{1/2}$$

Поэтому, выбирая $K = \sqrt{2\alpha}$, мы обеспечим в эксперименте выполнение условия устойчивости.

Далее для оценки порядка сходимости нам нужно выполнить следующие шаги.

1. Вычисление погрешностей на последовательности сеток. Для этого зададим начальный параметр дискретизации δ_0, и выполним эксперимент уменьшая δ: $\delta_k = 2^k \delta_0$, $k = 0, 1, \ldots, m$. Для каждого эксперимента мы должны сохранять E и δ. Стандартно для оценки погрешности используются нормы сеточных функций:

$$E = \|e^n\|_2 = \left(\sum_{n=0}^{N_t} \sum_{i=0}^{N_x} (e_i^n)^2 h\tau, \right)^{1/2}, \quad e_i^n = u_{\mathrm{e}}(x_i, t_n) - y_i^n, \tag{12}$$

$$E = \|e^n\|_\infty = \max_{i,n} |e_i^n|. \tag{13}$$

Можно также использовать норму на одном временном шаге, например, в конечный момент времени T ($n = N_t$):

$$E = \|e^n\|_2 = \left(\sum_{i=0}^{N_x} (e_i^n)^2 h \right)^{1/2}, \tag{14}$$

$$E = \|e^n\|_\infty = \max_{0 \le i \le N_x i} |e_i^n|. \tag{15}$$

2. Пусть E_k — погрешность, полученная на эксперименте с номером k, а δ_k — соответствующий параметр дискретизации. Положив $E_k = D\delta_k^r$, мы можем оценить r сравнивая последовательные эксперименты:

$$E_{k+1} = D\delta_{k+1}^r,$$
$$E_k = D\delta_k^r.$$

Отсюда получим

$$r = \frac{\ln(E_{k+1}/E_k)}{\ln(\delta_{k+1}/\delta_k)}.$$

Так как r зависит от k добавим индекс к r: r_k, где $k = 0, 1, \ldots, m-2$, если проведено m экспериментов: (δ_0, E_0), (δ_1, E_1), …, (δ_{m-1}, E_{m-1}).

В нашей рассматриваемой аппроксимации параболического уравнения $r = 1$ и отсюда значения r_i должны стремиться к 1 с ростом k.

2.2. Численный эксперимент

Если представленные выше тестовые функции выполняются без ошибок, мы имеем основания предполагать, что реализация нашего алгоритма верная. Следующий шаг — проведение численного эксперимента.

Рассмотрим задачу, где x/L — новая пространственная переменная, а at/L^2 — новая временная переменная. Правая часть (источник) f отсутствует, а u ограничена величиной $\max_{x\in[0,L]} |I(x)|$. В этом случае параболическое уравнение принимает вид

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

на отрезке $[0, 1]$ с граничными условиями $u(0) = 0$ и $u(1) = 0$. Протестируем два начальных условия: разрывную площадку

$$I(x) = \begin{cases} 0, & |x - L/2| > 0.1 \\ 1, & |x - L/2| \le 0.1 \end{cases}$$

и гладкую функцию Гаусса

$$I(x) = e^{-\frac{1}{2\sigma^2}(x-L/2)^2}$$

Функции `plug` и `gaussian` в файле parab1d.py[3] запускают решение двух соответствующих задач:

```python
def plug(scheme='Ex', F=0.5, Nx=50):
    L = 1.
    a = 1.
    T = 0.1
    # Вычисляем tau по Nx и F
    h = L/Nx
```

[3] `src-time-dep/parab1d.py`

```python
    tau = F*h**2/a

    def I(x):
        if abs(x - L/2) > 0.1:
            return 0.
        else:
            return 1.

    def f(x, t):
        return 0.0

    cpu = viz(I, a, f, L, tau, F, T,
              u_min=-0.1, u_max=1.1,
              scheme=scheme, animate=True,
              framefiles=True)
    print('Время расчета: {}'.format(cpu))
# End plug
# Start gaussian
def gaussian(scheme='Ex', F=0.5, Nx=50, sigma=0.05):
    L = 1.
    a = 1.
    T = 0.01
    # Вычисляем tau по Nx и F
    h = L/Nx
    tau = F*h**2/a

    def I(x):
        return np.exp(-0.5*((x - L/2)**2)/(sigma**2))

    def f(x, t):
        return np.zeros_like(x)

    cpu = viz(I, a, f, L, tau, F, T,
              u_min=-0.1, u_max=1.1,
              scheme=scheme, animate=True,
              framefiles=True)
    print('Время расчета: {}'.format(cpu))
```

Эти функции используют функцию `viz` для запуска солвера и визуализации решения посредством использования функции обратного вызова для рисования:

```python
def viz(I, a, f, L, tau, F, T, u_min, u_max,
        scheme='Ex', animate=True, framefiles=True):
    def plot_u(u, x, t, n):
        plt.clf()
        plt.plot(x, u, 'r-')
        plt.axis([0., L, u_min, u_max])
        plt.title('$t=${}'.format(t[n]))
        if framefiles:
            plt.savefig('tmp_frame{:04d}.png'.format(n))
            plt.close()

    user_action = plot_u if animate else lambda u, x, t, n: None
    u, x, t, cpu = eval('solver_'+scheme)(I, a, f, L, tau, F, T,
                                          user_action=user_action)
    return cpu
```

Отметим, что функция `viz` посредством функции обратного вызова также может сохранять графики решений на каждом временном слое в файлы формата `png`. Эти файлы можно использовать для генерации видео файлов.

3. Задачи

Задача 1: Задача третьего рода

Написать программу для численного решения краевой задачи для одномерного параболического уравнения с краевыми условиями третьего рода:

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + f(x,t), \quad 0 < x < l,\ 0 < t \le T,$$

$$-\frac{\partial u}{\partial x}(0,t) + \sigma_1(t)u(0,t) = \mu_1(t), \quad \frac{\partial u}{\partial x}(l,t) + \sigma_2(t)u(l,t) = \mu_2(t),\ 0 < t \le T,$$

$$u(x,0) = I(x), \quad 0 \le x \le l.$$

на равномерной по пространству и времени сетке при использовании явной разностной схемы. Протестируйте скорость сходимости реализованного метода при $l = 1$, $\sigma_1 = \sigma_2 = 0, 10, 100$ на точном решении

$$u(x,t) = \sin t(1 + 2x - 3x^2).$$

Предметный указатель