Курсовая работа по теме «Обыкновенные дифференциальные уравнения»

Дмитрий Джус

4 марта 2008 г.

Содержание

Часть I
Постановка задачи

1 Математический аспект

Настоящая курсовая работа посвящена построению приближённого решения краевой задачи для дифференциального уравнения вида

$$\frac{d^2u}{dx^2} + n(x)u(x) = 0 \tag{1.1}$$

По условию, $n(x)$ представляет собой кусочно-непрерывную функцию, равную константе k^2 на интервалах $(-\infty; 0)$ и $(a; +\infty)$ и непрерывной известной функции на $[0; a]$. На границах $[0; a]$ решение и его первая производная должны удовлетворять условиям сшивания:

$$u(0-0) = u(0+0) \qquad u'(0-0) = u'(0+0) \tag{1.2a}$$
$$u(a-0) = u(a+0) \qquad u'(a-0) = u'(a+0) \tag{1.2b}$$

2 Физический аспект

Задача соответствует возбуждения плоского слоя неоднородной немагнитной среды электромагнитным полем, вектор электрического поля распространяется вдоль оси x. k имеет смысл волнового числа в однородной части пространства, $n(x)$ — показатель преломления неоднородной среды.

Из физических соображений, решение справа от слоя $[0; a]$ должно иметь вид $u(x) = Be^{ikx}$, а слева — $u(x) = e^{ikx} + Ae^{-ikx}$. Задача состоит в приближённом построении решения внутри неоднородного слоя и отыскании комплексных констант A и B (называемых, соответственно, коэффициентами отражения и прохождения электромагнитного поля).

Действительная область значений функции $n(x)$ соответствует отсутствию поглощения энергии в среде, так что критерием правильности полученных приближённых значений служит энергетическое тождество:

$$|A|^2 + |B|^2 = 1 \tag{2.1}$$

Часть II
Решение

3 Общие замечания

Предлагаемые к реализации в курсовой работае приближённые методы были описаны на алгоритмическом языке Scheme (простом и строго функциональном диалекте Лиспа).

4 Решение методом построения фундаментальной матрицы

4.1 Описание

Дифференциальное уравнение второго порядка (1.1) можно представить в матричной форме, введя $v(x) = \frac{du}{dx}$:

$$\frac{d}{dx}\begin{pmatrix} u(x) \\ v(x) \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ -n(x) & 0 \end{pmatrix}\begin{pmatrix} u(x) \\ v(x) \end{pmatrix} \tag{4.1}$$

Решение системы (4.1) с матрицей $A(x) = \left(\begin{smallmatrix} 0 & 1 \\ -n(x) & 0 \end{smallmatrix}\right)$ можно найти в общем виде, если построена фундаментальная матрица $\Omega(x)$:

$$\Omega(x) = \begin{pmatrix} \omega_{11}(x) & \omega_{12}(x) \\ \omega_{21}(x) & \omega_{22}(x) \end{pmatrix} \tag{4.2}$$

$\Omega(x)$ является решением следующей матричной задачи Коши:

$$\frac{d}{dx}\Omega(x) = A(x)\Omega(x) \tag{4.3}$$

с условием $\Omega(0) = \left(\begin{smallmatrix} 1 & 0 \\ 0 & 1 \end{smallmatrix}\right)$.
Решение (4.1) для $\forall x \in [0; a]$ записывается в виде

$$\begin{pmatrix} u(x) \\ v(x) \end{pmatrix} = \begin{pmatrix} \omega_{11}(x) & \omega_{12}(x) \\ \omega_{21}(x) & \omega_{22}(x) \end{pmatrix}\begin{pmatrix} 1 + A \\ ik(1 - A) \end{pmatrix} \tag{4.4}$$

Использование краевых условий (1.2) позволяет получить следующую линейную алгебраическую систему относительно неизвестных A и B:

$$\begin{aligned} (\omega_{11}(a) - \omega_{12}(a)ik)A - e^{ika}B &= -ik\omega_{12}(a) - \omega_{11}(a) \\ (\omega_{21}(a) - \omega_{22}(a)ik)A - ike^{ika}B &= -ik\omega_{22}(a) - \omega_{21}(a) \end{aligned} \tag{4.5}$$

Нахождение значений компонентов матрицы Ω на правом конце $[0; a]$, таким образом, позволит получить искомые значения A и B.

Для поиска фундаментальной матрицы в точках $[0; a]$ предлагается использование следующего численного метода, основанного на приближении функции $n(x)$ кусочно-постоянной, то есть замене неоднородной среды на плоско-слоистую с большим числом однородных слоёв.

Рассматриваемый интервал разбивается на N интервалов точками $x_i = \frac{a}{N} i$, $i = 1 \dots N$, в середине каждого из которых берётся точка $\overline{x_i} = \frac{a}{N}(i - \frac{1}{2})$. В данной точке функция $n(x)$ аппроксимируется ступенчатой функцией $\overline{n}(x) = n(\overline{x_i})$, так что (4.1) обращается в систему с постоянной матрицей $A_i = \left(\begin{smallmatrix} 0 & 1 \\ -\overline{n}(x) & 0 \end{smallmatrix} \right)$. Тогда на каждом интервале $[x_i; x_{i+1}]$ решение (4.3) может быть найдено явно в виде матричной экпоненты:

$$\Omega_i(x) = e^{A_i(x - x_i)}$$

При этом, значения фундаментальной матрицы $\Omega_N(x_{i+1})$ на правом конце $[x_i; x_{i+1}]$ служат начальным условием задачи Коши на $[x_{i+1}; x_{i+2}]$, так что для каждой точки $\hat{x} \in [x_i; x_{i+1}]$ фундаментальная матрица (4.2) имеет вид:

$$\Omega_N(\hat{x}) = e^{A_i(\hat{x} - x_i)} e^{A_{i-1}(x_i - x_{i-1})} \cdots e^{A_0(x_1 - x_0)} \tag{4.6}$$

На правом конце интервала $[0; a]$ приближённая фундаментальная матрица записывается в виде:

$$\Omega_N(a) = e^{A_{N-1}(x_N - x_{N-1})} e^{A_{N-2}(x_{N-1} - x_{N-2})} \cdots e^{A_0(x_1 - x_0)} \tag{4.7}$$

Для нахождения матричной экспоненты используется формула Тейлора:

$$\begin{aligned} e^{A_i(x - x_i)} \approx E + \frac{1}{1!} A_i(x - x_i) + \frac{1}{2!} A_i^2 (x - x_i)^2 \\ + \frac{1}{3!} A_i^3 (x - x_i)^3 + \frac{1}{4!} A_i^4 (x - x_i)^4 \end{aligned} \tag{4.8}$$

После построения фундаментальной матрицы в a коэффициенты A и B находятся из (4.5) с использованием полученных компонент $\omega_{ij}^N(a)$.

Решение на каждом отрезке разбиения $[0; a]$ находится из (4.4) с использование найденного A и последовательности приближений фундаментальной матрицы на $[x_i; x_{i+1}]$ для $\forall i$.

4.2 Реализация

Описание основных процедур, необходимых для реализации метода, находится в файле `fundmatrix-solution.scm` (его полное содержание приведено на странице 24).

В исходном тексте программы активно используются последовательные определения функций в терминах друг друга.

Так, основной вычислительный процесс метода — построение приближений фундаментальной матрицы — выражен в функции `build-fundamentals`:

```
(define (build-fundamentals right-bound n matrix)
  (let ((a right-bound)
        (step (/ right-bound n)))
    (evolve-sequence
     (lambda (prev a)
       (matrix-*-matrix
        prev
        ((matrix-exp matrix 5)
         a
         (+ a (/ step 2))
         (- a (/ step 2)))))
     (identity-matrix 2)
     (split-interval 0 right-bound n))))
```

`build-fundamentals` представляет последовательность вычислений фундаментальной матрицы на интервалах $[x_i; x_{i+1}]$ в виде частного случая функции `evolve-sequence`, возвращающей последовательность $a_{s_1}, \ldots, a_{s_n}$, где $a_{s_k} = f(a_{s_{k-1}}, s_{k-1})$ (см. с. 19). Из определения `build-fundamentals` видно, что роль s_k выполняют точки $x_i = \frac{a}{N} i$ с $i = 1 \ldots N$, а $a_{s_{k-1}}$ — приближение фундаментальной матрицы на предыдущем отрезке (на первом шаге Ω_N приближается единичной матрицей). Комбинируются же эти значения при помощи матричного умножения, как следует из (4.6).

Функция `variable-matrix` возвращает матрицу A системы (4.1) для заданной по условию функции $n(x)$:

```
(define (variable-matrix n)
  (lambda (x)
    (matrix (row 0          1)
            (row (- (n x)) 0))))
```

Возвращённая матрица потом используется в `build-fundamentals`.

В `build-fundamentals` используется и функция `matrix-exp`, вычисляющая матричную экспоненту $f(x, y, z) = e^{A(x)(y-z)}$ путём разложения в ряд Тейлора (4.8) с использованием схемы Горнера:

```
(define (matrix-exp A n)
  (lambda (x y z)
    (let ((matrix (A x)))
```

```
    (general-horner-eval
     (matrix-*-number matrix (- y z))
     (exp-series-coefficients n)
     matrix-*-matrix
     (lambda (high-terms coeff)
       (add-matrices high-terms
                     (matrix-*-number
                      (identity-matrix (matrix-size matrix))
                      coeff)))
     (zero-matrix (matrix-size matrix))))))
```

В этой функции последовательность коэффициентов матричного многочлена $\frac{1}{0!}, \frac{1}{1!}, \frac{1}{2!}, \ldots$ генерируется функцией `exp-series-coefficients`, выраженной в терминах вышеупомянутой `evolve-sequence`.

Вычисление матричной экспоненты в каждой точке выражено через функцию высшего порядка `general-horner-eval`, представляющую собой обобщённую схему Горнера:

```
(define (general-horner-eval x coefficient-sequence
                             mult add zero)
  (fold-right
   (lambda (this-coeff higher-terms)
     (add (mult higher-terms x) this-coeff))
   zero
   coefficient-sequence))
```

`general-horner-eval`, в свою очередь, является частным случаем функции высшего порядка `fold-right`, реализующей свёртку последовательности справа налево.

После построения последовательности фундаментальных матриц осуществляется поиск коэффициентов A, B путём решения системы (4.5) с помощью метода Гаусса (его исходный код приведён на странице 22):

```
(define (find-A-B fundamentals k right-bound)
  (let ((fundamental (list-ref fundamentals
                               (- (length fundamentals) 1)))
        (a right-bound))
    (define (w i j)
      (get-item (get-row fundamental i) j))
    (solve-linear
     (matrix
      (row (- (w 1 1) (* (w 1 2) +i k))
           (- (exp (* +i k a))))
      (row (- (w 2 1) (* (w 2 2) +i k))
           (- (* (exp (* +i k a)) +i k))))
     (vector
      (- (- (* (w 1 2) +i k)) (w 1 1))
      (- (- (* (w 2 2) +i k)) (w 2 1))))))
```

С использованием найденного коэффициента A и приближений из `build-fundamentals`

в функции `approximate-solution` согласно (4.4) рассчитывается и приближённое решение исходного уравнения (1.1) внутри неоднородного слоя:

```
(define (approximate-solution fundamentals A k)
  (map
   (lambda (matrix)
     (caar
      (matrix-*-vector
       matrix
       (vector (+ 1 A)
                    (* +i k (- 1 A))))))
   fundamentals))
```

Функция высшего порядка `iterative-improve` выражает один из подходов к решению вычислительных задач: улучшать начальное решение при помощи заданной «функции улучшения» до тех пор, пока его не можно будет считать «хорошим» в смысле заданной функции оценки решения:

```
(define (iterative-improve good? improve)
  (define (solve x)
    (if (good? x)
        x
        (solve (improve x))))
  solve)
```

Так, в данном методе процедура построения фундаментальных матриц повторяется до тех пор, пока вычисляемые коэффициенты отражения и прохождения не будут удовлетворять (2.1):

```
(define (get-solution refraction right-bound subintervals eps)
  (let ((wave-number (get-wave-number refraction)))
    (define (improve solution)
      (let* ((fundamentals (build-fundamentals
                            right-bound
                            (* 2 (length (car solution)))
                            (variable-matrix refraction)))
             (coeffs (find-A-B
                      fundamentals
                      wave-number
                      right-bound))
             (A (car coeffs))
             (approx (approximate-solution
                      fundamentals
                      A wave-number)))
        (cons approx (cons A (cadr coeffs)))))
    (define (good? solution)
      (let* ((coeffs (cdr solution))
             (A (car coeffs))
             (B (cdr coeffs)))
        (energy-conserves? A B eps)))
    (let* ((initial-solution
```

```
            (cons (tabulate-function (lambda (x) 0)
                                     0 right-bound subintervals)
                  (cons 0 0))))
      ((iterative-improve good? improve) initial-solution))))
```

Интересно, что второй метод решения поставленной задачи, о котором рассказано далее, также определяет сходную функцию `make-solution` в виде частного случая `iterative-improve`.

Функцией улучшения является двукратное увеличение числа отрезков разбиения интервала $[0; a]$ с последующим построением фундаментальных матриц, решения и нахождения A, B, а проверка «качества» полученного решения осуществляется при помощи предиката `energy-conserves?`:

```
(define (energy-conserves? A B eps)
  (< (abs (- 1
             (+ (expt (magnitude A) 2)
                (expt (magnitude B) 2))))
     eps))
```

4.3 Исходные данные

Рассматривается отрезок $[0; 2]$.

Функция показателя преломления среды $n(x)$ задана следующим образом:

$$n(x) = \begin{cases} 35 + 3(x-1)^2 & 0 < x < 2 \\ 36 & x \leq 0,\ x \geq 2 \end{cases} \tag{4.9}$$

В терминах Scheme она задаётся следующим образом:

```
(define (f x)
  (if (and (> x 0) (< x 2))
      (+ 35 (* 3 (expt (- x 1) 2)))
      36))
```

4.4 Результаты

В результате использования метода были получены следующие результаты:

$$\begin{aligned} A &= -0.00478 - 0.00750i \\ B &= 0.99996 - 0.00104i \end{aligned} \tag{4.10}$$

Подтверждено, что значения A и B удовлетворяют условию 2.1:

$$\left| \left(|A|^2 + |B|^2 \right) - 1 \right| < 0.0001$$

Расчёт занял 1.320155 с.

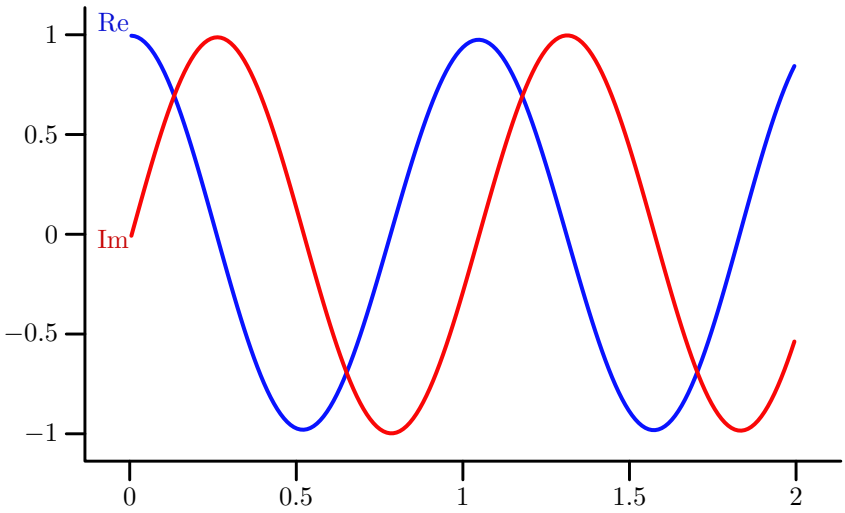

Рис. 1: График $u(x)$ для функции преломления (4.9)

4.4.1 Результат вычислений с альтернативным набором исходных данных

Метод построения фундаментальных матриц применим в задачах с более сложными выражениями для $n(x)$ (но требуется, чтобы $n(x)$ была непрерывной внутри $[0; a]$). Далее представлены результаты работы с иными, нежели в предыдущем разделе, данными.

Рассматривается отрезок $[0; 2]$.

Функция показателя преломления среды $n(x)$ задана следующим образом:

$$n(x) = \begin{cases} \frac{1}{x+0.1} + e^x x^5 & 0 < x < 2 \\ 100 & x \leq 0,\ x \geq 2 \end{cases} \tag{4.11}$$

В результате использования метода были получены следующие результаты:

$$\begin{aligned} A &= 0.70293 + 0.35912i \\ B &= 0.45555 + 0.41157i \end{aligned} \tag{4.12}$$

Подтверждено, что значения A и B удовлетворяют условию 2.1:

$$\left| \left(|A|^2 + |B|^2 \right) - 1 \right| < 0.000001$$

Расчёт занял 0.226783 с.

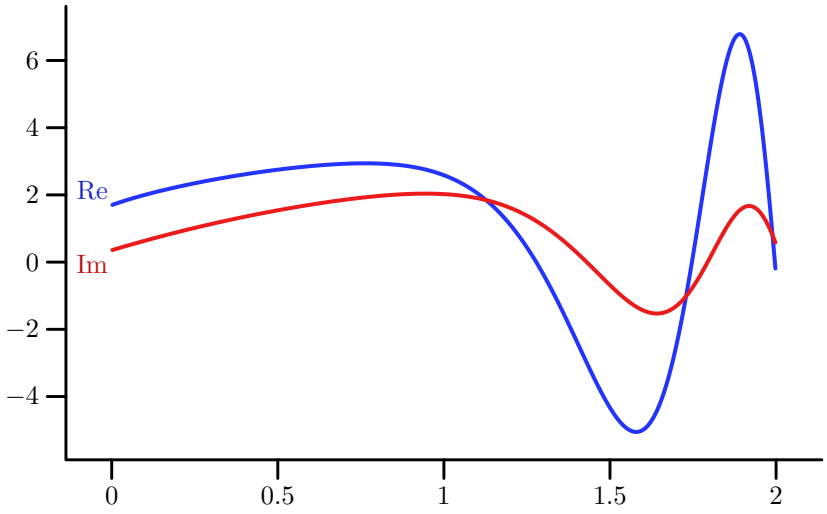

Рис. 2: График $u(x)$ для функции преломления (4.11)

5 Решение методом последовательных приближений

5.1 Описание

Уравнение (1.1) можно переписать в следующем виде:

$$\frac{d^2u}{dx^2} + k^2u(x) = (k^2 - n(x))u(x) \tag{5.1}$$

Где k^2 — значение функции вне интервала $[0; a]$.

Для такого уравнения известна функция Грина:

$$G(x,t) = \frac{e^{ik|x-t|}}{2ik}$$

Так что решение (5.1) выписывается в таком виде:

$$u(x) = \int\limits_{-\infty}^{+\infty} \frac{e^{ik|x-t|}}{2ik}(k^2 - n(t))u(t)dt$$

В силу свойств функции $n(x)$, при $x < 0$ и $x > a$ имеем $(k^2 - n(x)) \equiv 0$, так что несобственный интеграл можно заменить на интеграл в конечных пределах от 0 до a:

$$u(x) = \int\limits_{0}^{a} \frac{e^{ik|x-t|}}{2ik}(k^2 - n(t))u(t)dt$$

Кроме того, если раскрыть модуль в показателе e для $x < 0$ и $x > a$, то станет ясно, что выписанное решение содержит волны, уходящие в $+\infty$ и $-\infty$. Так как постановка задачи содержит волну e^{ikx}, приходящую из минус бесконечности, полное поле во всей области от $-\infty$ до $+\infty$ имеет следующий вид:

$$u(x) = \int\limits_{0}^{a} \frac{e^{ik|x-t|}}{2ik}(k^2 - n(t))u(t)dt + e^{ikx} \tag{5.2}$$

Сокращённо это интегральное уравнение Фредгольма второго рода записывается в таком виде:

$$u = \mathrm{A}u + f \tag{5.3}$$

Где A — интегральный оператор $\int_0^a \frac{e^{ik|x-t|}}{2ik}(k^2 - n(t))u(t)dt$, который действует на функцию $u(x)$.

При $x < 0$ решение имеет вид:

$$u(x) = \frac{e^{-ikx}}{2ik} \int\limits_0^a e^{ikt}(k^2 - n(t))u(t)dt + e^{ikx}$$

Откуда можно получить выражение для коэффициента A:

$$A = \frac{1}{2ik} \int\limits_0^a e^{ikt}(k^2 - n(t))u(t)dt \tag{5.4}$$

Аналогично получается следующее выражение и для коэффициента B:

$$B = \frac{1}{2ik} \int\limits_0^a e^{-ikt}(k^2 - n(t))u(t)dt + 1 \tag{5.5}$$

Построение решения $u(x)$ производится при помощи метода последовательных приближений. Решение (5.3) представим в таком виде:

$$u = u_0 + u_1 + u_2 + \cdots$$

Его подстановка в (5.3) даёт следующее:

$$u_0 + u_1 + u_2 + \cdots = \mathrm{A}u_0 + \mathrm{A}u_1 + \mathrm{A}u_2 + \cdots + f$$

Если положить $u_0 = f, u_1 = \mathrm{A}u_0, \ldots, u_{n+1} = \mathrm{A}u_n$, то таким выбором последовательных приближений это уравнение будет тождественно удовлетворено.

После нахождения $u(x)$ коэффициенты A и B вычисляются из (5.4) и (5.5), соответственно.

5.2 Реализация

Для обеспечения работы программы с широким диапазоном исходных данных — различных функций $n(x)$ — интегрирование (5.2) осуществлялось *численно* по формуле Симпсона.

Подинтегральная функция зависит от двух переменных x, t, а интегрирование производится по t. Используемая функция `integrate` возвращает $g(x) = \int_a^b f(x,t)dt$, применяя формулу Симпсона так:

$$\int_a^b f(x,t)dt = \frac{h}{3}(f(x,a) + 4(f(x,t_1) + f(x,t_3) + \cdots + f(x,t_{n-1})) + \\ +2(f(x,t_2) + f(x,t_4) + \cdots + f(x,t_{n-2})) + f(x,b)) \tag{5.6}$$

Здесь $n = 2k, k \in \mathbb{Z},\ h = (b-a)/n,\ t_k = a + kh$.

`integrate` осуществляет интегрирование функции, переданной ей в качестве параметра, по второму её аргументу:

```
(define (integrate f a b subintervals)
  (let ((h (/ (- b a) subintervals)))
    (lambda (x)
      (* (/ h 3)
         (+ (f x a)
            (* 4 (sum
                  (map (lambda (t) (f x t))
                       (split-interval a b
                                       (/ subintervals 2)))))
            (* 2 (sum
                  (map (lambda (t) (f x t))
                       (split-interval (+ a h) (- b h)
                                       (- (/ subintervals 2) 1)))))
            (f x b))))))
```

Функция `integrate` выражает идею интегрирования функции двух переменных в общем виде. Для вычисления (5.2) требуется использовать приближение функции $u(x)$ и данную по условию функцию $n(x)$ в выражении $f(x,t) = e^{ik|x-t|}(k^2 - n(t))u(t)$, которое и будет интегрироваться (постоянный множитель $1/2ik$ выносится из под интеграла). Требуемое сопоставление выполняет функция `green-transform`:

```
(define (green-transform u n)
  (green-subtransform u n
                      (lambda (x t) (abs (- x t)))))
```

Причём эта функция представлена в виде частного случая более общей процедуры `green-subtransform`, выполняющей преобразование своих трёх аргументов-функций $u(t), n(t), g(x,t)$ в функцию двух аргументов $f(x,t) = e^{ik\cdot g(x,t)}(k^2 - n(t))u(t)$:

```
(define (green-subtransform u n g)
  (let ((k (get-wave-number n)))
    (lambda (x t)
      (* (exp (* +i k (g x t)))
         (- (sqr k) (n t))
         (u t)))))
```

Тогда при $g(x,t) \equiv |x-t|$ получим подинтегральную функцию из (5.2), а при $g(x,t) \equiv -t$ — из выражения для коэффициента B (5.5).

Функция `green-integrate`, используя описанные `green-transform` и `integrate`, выполняет интегрирование (5.2), принимая функции $u(x), n(x)$ и правый край отрезка a в качестве аргументов. В сущности, `green-integrate` представляет в терминах Scheme оператор A из (5.3):

```
(define (green-integrate u refraction right-bound subintervals)
  (let ((k (get-wave-number refraction)))
    (lambda (x)
      (/ ((integrate (green-transform u refraction)
                     0 right-bound subintervals) x)
         (* 2 +i k)))))
```

В этой функции также осуществляется деление результата интегрирования на константу $2ik$.

Стоит заметить, что процедура `integrate` (и, следовательно, `green-integrate`) возвращает не число, а *функцию*. Вычисление значения этой функции в любой точке x_0 порождает свёртку интервала $[0;a]$ в эту точку. Таким образом, последовательное применение `green-integrate` порождает функцию, вычисление значения которой породит последовательность вложенных свёрток на интервале $[0;a]$.

5.2.1 Применимость метода

Сходимость ряда последовательных приближений $u = u_0 + \mathrm{A}u_0 + \mathrm{A}^2 u_0 \cdots$ обеспечивается наличием константы $1/2ik$ перед интегралом (5.2), если данные $a, n(x), k$ таковы, что для $M = \max(k^2 - n(x_0))$ выполняется:

$$\frac{Ma}{2k} < 1 \tag{5.7}$$

В случаях, когда исходные данные не удовлетворяют этому условию, ряд приближений разойдётся и получить решение не удастся, в то время как при помощи фундаментальных матриц за приемлемое время решаются уравнения в системах, где (5.7) не выполняется.

Исходные данные (функция $n(x)$), предлагаемые к использованию в настоящей курсовой работе, специально подобраны таким образом, чтобы (5.7) заведомо выполнялись.

5.2.2 Эффективность реализации

Высокий уровень применяемой абстракции и простота реализации метода негативным образом сказываются на эффективности: создаваемая последовательным действием `green-integrate` композиция свёрток — функция, вычисление которой в каждой точке требует большого количества операций.

Сложность `iterative-solve` по элементарным операциям `integrate` экспоненциально растёт с увеличением числа применений оператора A к начальному приближению.

При $u_k(x) = \underbrace{\mathrm{A} \circ \cdots \circ \mathrm{A}}_{k} \circ u_0(x)$ для вычисления значения $u_k(x_0)$ *в любой точке* x_0 потребуется выполнение m^k операций, где m — постоянное число элементарных операций сложения, умножения в процедуре `integrate`, зависящее от количества разбиений для интегрирования функции на $[0; a]$).

Учитывая построение *последовательных* приближений, то есть последовательное вычисление $\mathrm{A} \circ e^{ikx}$, $\mathrm{A} \circ \mathrm{A} \circ e^{ikx}$, $\mathrm{A} \circ \mathrm{A} \circ \mathrm{A} \circ e^{ikx}, \ldots$, количество выполняемых операций таково, что описываемый подход к реализации метода очевидно не может реально применяться на практике в расчётных задачах в силу низкой эффективности.

5.2.3 Другие подходы к реализации метода

Символьное интегрирование Интеграл (5.2) может быть вычислен по формуле Ньютона-Лейбница, если известна первообразная подинтегральной функции. Её поиск в явном виде составляет задачу символьного неопределённого интегрирования.

Интегрирование элементарных функций и их простых сочетний является решаемой алгоритмически задачей, которая, однако, не является тривиальной (по причинам, в большей степени относящейся к общим сложностям представления алгебраических выражений на компьютере). Сложные выражения под знаком интеграла приводят к непростым заменам, преобразованиям и приёмам. Интеграл может и не браться в явном виде.

Символьное интегрирование потенциально даёт значительно большие преимущества, чем численное, но сравнительно сложнее в реализации. Описанная же ранее численная реализация метода последовательных приближений представляет собой почти дословный перевод словесного описания в термины Лиспа.

Использование готового выражения первообразной Первообразная $e^{ik|x-t|}(k^2 - n(t))u(t)$ по t может быть получена сторонними средствами — в другой программе или вручную (что является единственным универсальным и наиболее гибким методом интегрирования). По затратам на количество операций, выполняемых на стороне описываемой в настоящей работе программы, этот подход наиболее эффективен, поскольку сводит задачу нахождения интеграла к совсем элементарным операциям.

5.3 Результаты

Исходные данные к задаче приведены в разделе 4.9 на странице 10.

В результате использования метода были получены следующие результаты:

$$\begin{aligned} A &= -0.00579 - 0.00907i \\ B &= 0.99994 - 0.00325i \end{aligned} \tag{5.8}$$

Подтверждено, что значения A и B удовлетворяют условию 2.1:

$$\left| \left(|A|^2 + |B|^2 \right) - 1 \right| < 0.0001$$

Расчёт занял 0.933066 с.

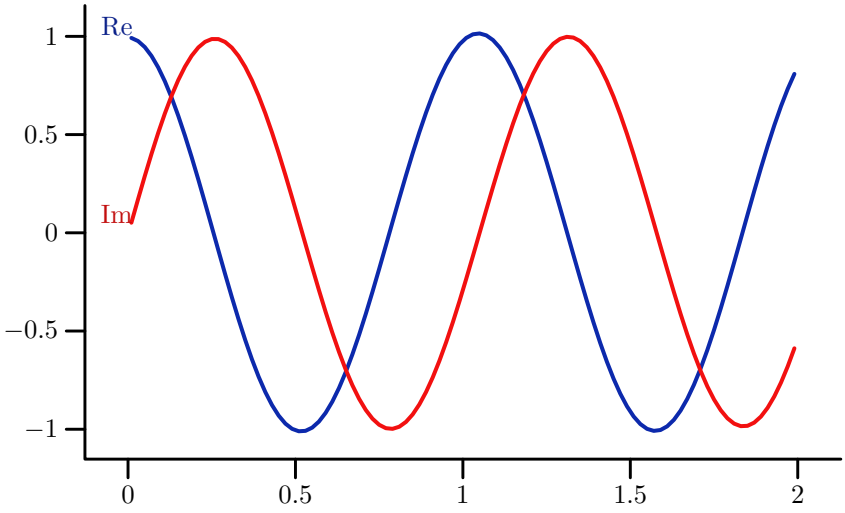

Рис. 3: График $u(x)$ для функции преломления (4.9)

6 Сопоставление результатов

Сравнение результатов вычислений для одних и тех же исходных данных (см. с. 10) при помощи различных методов — построением фундаментальной матрицы и решением интегрального уравнения последовательными приближениями — позволяет проверить корректность реализации обоих методов:

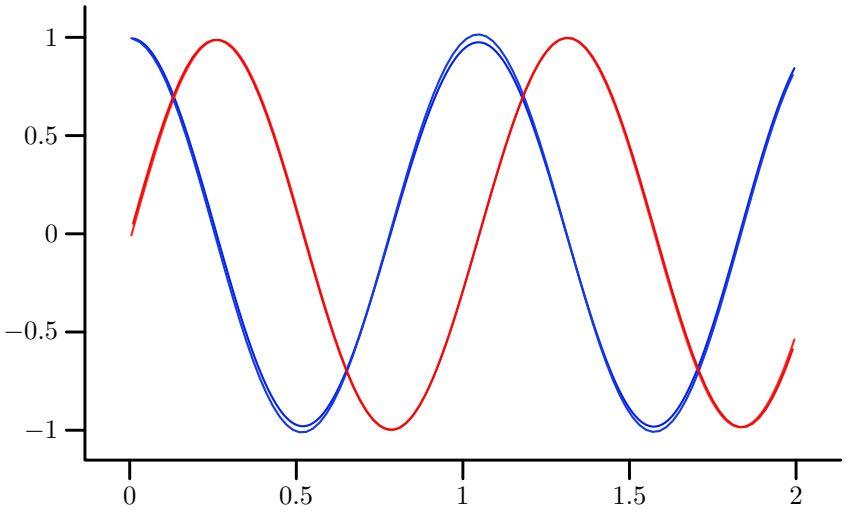

Рис. 4: Графики решения для (4.9), полученные двумя разными методами

Часть III
Исходные тексты

A Общие файлы

Содержимое файла `shared.scm`

```
(use-modules (srfi srfi-1))

;; ∑_n s_n
(define (sum sequence)
  (fold + 0 sequence))

;; x^2
(define (sqr x)
  (* x x))

;; (···((a_n · x + a_{n-1}) · x + a_{n-2}) · x ··· + a_1) · x + a_0)
(define (general-horner-eval x coefficient-sequence
                             mult add zero)
  (fold-right
   (lambda (this-coeff higher-terms)
     (add (mult higher-terms x) this-coeff))
   zero
   coefficient-sequence))

;; l, l+1, ..., h-1, h
(define (enumerate-interval low high)
  (define (iter low high acc)
    (if (> low high)
        acc
        (iter low (- high 1) (cons high acc))))
  (iter low high '()))

;; 1, 2, ..., n
(define (enumerate-n n)
  (enumerate-interval 1 n))

;; a_{s_1}, a_{s_2}, ..., a_{s_n}, a_{s_k} = f(a_{s_{k-1}}, s_k)
(define (evolve-sequence evolve initial index)
  (define (evolve-next index prev acc)
    (if (null? index)
        acc
        (let ((current (evolve prev (car index))))
          (evolve-next (cdr index)
                       current
                       (append acc (list current))))))
  (evolve-next index initial (list initial)))

;; a_1, a_2, ..., a_n, a_k = f(a_{k-1})
(define (evolve-series evolve initial n)
  (evolve-sequence evolve initial (enumerate-n n)))
```

```
;; x_1, x_2, ..., x_n, x_k = a + h(k - 1/2), h = (b - a)/n
(define (split-interval a b n)
  (let ((step (/ (- b a) n)))
    (evolve-series (lambda (x n) (+ x step))
                   (+ a (/ step 2))
                   (- n 1))))

;; 1/0!, 1/1!, 1/2!, ..., 1/(n - 1)!
(define (exp-series-coefficients n)
  (evolve-series (lambda (prev i) (/ prev i))
                 1
                 (- n 1)))

;; |A|^2 + |B|^2 = 1
(define (energy-conserves? A B eps)
  (< (abs (- 1
             (+ (expt (magnitude A) 2)
                (expt (magnitude B) 2))))
     eps))

;; f(x_1), f(x_2), ..., f(x_n), x_k = a + h(k - 1/2), h = (b - a)/n
(define (tabulate-function f a b subintervals)
  (map f (split-interval a b subintervals)))

;; e^{ikx}
(define (wave k)
  (lambda (x)
    (exp (* +i k x))))

(define (get-wave-number function)
  (sqrt (function -1)))

(define (iterative-improve good? improve)
  (define (solve x)
    (if (good? x)
        x
        (solve (improve x))))
  solve)
```

Содержимое файла `matrices.scm`

```
(use-modules (srfi srfi-1))

(load "shared.scm")

(define (vector . items)
  items)

(define (matrix . rows)
  rows)

(define row
  vector)

(define (get-item vector n)
  (list-ref vector (- n 1)))
```

```
(define get-row get-item)

(define (first-column matrix)
  (map car matrix))

(define (matrix-size m)
  (length m))

;; A = (a_ij)_{m×n} → A^T = (a_ji)_{n×m}
(define (transpose m)
  (array->list
   (transpose-array
    (list->array (list (length m)
                       (length (car m)))
                 m)
    1 0)))

;; M × N
(define (matrix-*-matrix m n)
  (let ((cols (transpose n)))
    (map (lambda (row)
           (map (lambda (col)
                  (sum (map * row col)))
                cols))
         m)))

;; A × v⃗
(define (matrix-*-vector matrix vector)
  (matrix-*-matrix matrix (map list vector)))

;; A = (a_ij) → A · c = (a_ij · c)
(define (matrix-*-number matrix n)
  (map
   (lambda (row)
     (map (lambda (x) (* x n)) row))
   matrix))

;; M + N = (m_ij + n_ij)
(define (add-matrices m n)
  (map
   (lambda (row1 row2)
     (map + row1 row2))
   m n))

;; A = (a_ij = 0)_{n×n}
(define (zero-matrix n)
  (map (lambda (row)
         (map (lambda (i) 0)
              (enumerate-n n)))
       (enumerate-n n)))

;; δ^i_j
(define (kronecker i j)
  (if (= i j) 1 0))

;; E = (δ^i_j)_{n×n}
(define (identity-matrix n)
```

```
  (map
   (lambda (i)
     (map
      (lambda (j)
        (kronecker i j))
      (enumerate-n n)))
   (enumerate-n n)))

;; f(x,y,z) = e^{A(x)(y-z)}
(define (matrix-exp A n)
  (lambda (x y z)
    (let ((matrix (A x)))
      (general-horner-eval
       (matrix-*-number matrix (- y z))
       (exp-series-coefficients n)
       matrix-*-matrix
       (lambda (high-terms coeff)
         (add-matrices high-terms
                       (matrix-*-number
                        (identity-matrix (matrix-size matrix))
                        coeff)))
       (zero-matrix (matrix-size matrix))))))
```

Содержимое файла `gauss.scm`

```
(use-modules (srfi srfi-1))

(load "shared.scm")
(load "matrices.scm")

(define (max-nonzero-index lst)
  (fold
   (lambda (i prev)
     (if (> (real-part (list-ref lst (- i 1)))
            (real-part (list-ref lst (- prev 1))))
         i
         prev))
   1
   (enumerate-n (length lst))))

;; l_1,...,l_i,l_j,...,l_n -> l_1,...,l_j,l_i,...,l_n
(define (swap-items i j lst)
  (map
   (lambda (index)
     (if (= index i)
         (list-ref lst (- j 1))
         (if (= index j)
             (list-ref lst (- i 1))
             (list-ref lst (- index 1)))))
   (enumerate-n (length lst))))

(define (solve-linear A v)
  (define (top-left equations)
    (caar equations))
  (define (top-right equations)
    (let ((first (car equations)))
      (get-item first (length first))))
```

```
(define (row-reduce equations)
  (map
   (lambda (equation)
     (map
      (lambda (first-eq coeff)
        (- coeff
         (* first-eq
            (/ (car equation)
               (top-left equations)))))
      (cdr (car equations))
      (cdr equation)))
   (cdr equations)))
(define (solve-equations equations)
  (if (= (matrix-size equations) 1)
      (vector (/ (top-right equations)
                 (top-left equations)))
      (let* ((next-row (max-nonzero-index
                        (first-column equations)))
             (equations (swap-items 1 next-row equations))
             (subsolution (solve-equations
                           (row-reduce equations))))
        (append
         (solve-equations (matrix
                           (row (top-left equations)
                                (- (top-right equations)
                                   (sum (map *
                                             (drop-right
                                              (cdr (car equations
                                                   )) 1)
                                             subsolution))))))
         subsolution))))
(let ((augmented (map (lambda (matrix-row vector-item)
                        (append matrix-row (list vector-item)))
                      A v)))
  (solve-equations augmented)))
```

B Реализации методов решения

B.1 Метод построения фундаментальной матрицы

Содержимое файла `fundmatrix-solution.scm`

```
(load "shared.scm")
(load "matrices.scm")
(load "gauss.scm")

(define (build-fundamentals right-bound n matrix)
  (let ((a right-bound)
        (step (/ right-bound n)))
    (evolve-sequence
     (lambda (prev a)
       (matrix-*-matrix
        prev
        ((matrix-exp matrix 5)
         a
         (+ a (/ step 2))
         (- a (/ step 2)))))
     (identity-matrix 2)
     (split-interval 0 right-bound n))))

(define (variable-matrix n)
  (lambda (x)
    (matrix (row 0          1)
            (row (- (n x)) 0))))

(define (find-A-B fundamentals k right-bound)
  (let ((fundamental (list-ref fundamentals
                               (- (length fundamentals) 1)))
        (a right-bound))
    (define (w i j)
      (get-item (get-row fundamental i) j))
    (solve-linear
     (matrix
      (row (- (w 1 1) (* (w 1 2) +i k))
           (- (exp (* +i k a))))
      (row (- (w 2 1) (* (w 2 2) +i k))
           (- (* (exp (* +i k a)) +i k))))
     (vector
      (- (- (* (w 1 2) +i k)) (w 1 1))
      (- (- (* (w 2 2) +i k)) (w 2 1))))))

(define (approximate-solution fundamentals A k)
  (map
   (lambda (matrix)
     (caar
      (matrix-*-vector
       matrix
       (vector (+ 1 A)
               (* +i k (- 1 A))))))
   fundamentals))

(define (get-solution refraction right-bound subintervals eps)
  (let ((wave-number (get-wave-number refraction)))
```

```
(define (improve solution)
  (let* ((fundamentals (build-fundamentals
                        right-bound
                        (* 2 (length (car solution)))
                        (variable-matrix refraction)))
         (coeffs (find-A-B
                  fundamentals
                  wave-number
                  right-bound))
         (A (car coeffs))
         (approx (approximate-solution
                  fundamentals
                  A wave-number)))
    (cons approx (cons A (cadr coeffs)))))
(define (good? solution)
  (let* ((coeffs (cdr solution))
         (A (car coeffs))
         (B (cdr coeffs)))
    (energy-conserves? A B eps)))
(let* ((initial-solution
        (cons (tabulate-function (lambda (x) 0)
                                 0 right-bound subintervals)
              (cons 0 0))))
  ((iterative-improve good? improve) initial-solution))))
```

B.2 Метод последовательных приближений

Содержимое файла `iterative-solution.scm`

```
(load "shared.scm")

;; φ(u(x), n(x), g(x,t)) = f(x,t) = e^{ik·g(x,t)}(k^2 − n(t))u(t)
(define (green-subtransform u n g)
  (let ((k (get-wave-number n)))
    (lambda (x t)
      (* (exp (* +i k (g x t)))
         (- (sqr k) (n t))
         (u t))))))

;; φ̂(u(x), n(x)) = f(x,t) = e^{ik|x−t|}(k^2 − n(t))u(t)
(define (green-transform u n)
  (green-subtransform u n
                      (lambda (x t) (abs (- x t)))))

;;  ∫_a^b f(x,t)dt
(define (integrate f a b subintervals)
  (let ((h (/ (- b a) subintervals)))
    (lambda (x)
      (* (/ h 3)
         (+ (f x a)
            (* 4 (sum
                  (map (lambda (t) (f x t))
                       (split-interval a b
                                       (/ subintervals 2)))))
            (* 2 (sum
                  (map (lambda (t) (f x t))
                       (split-interval (+ a h) (- b h)
                                       (- (/ subintervals 2) 1)))))
            (f x b))))))

;; A ∘ u(x): 1/(2ik) ∫_0^a e^{ik|x−t|}(k^2 − n(t))u(t)dt
(define (green-integrate u refraction right-bound subintervals)
  (let ((k (get-wave-number refraction)))
    (lambda (x)
      (/ ((integrate (green-transform u refraction)
                     0 right-bound subintervals) x)
         (* 2 +i k)))))

(define (find-A-B solution refraction right-bound subintervals)
  (let ((k (get-wave-number refraction)))
    (let (
          ;; A = 1/(2ik) ∫_0^a e^{ikt}(k^2 − n(t))u(t)dt
          (A ((green-integrate solution refraction
                               right-bound subintervals) 0))
          ;; B = 1/(2ik) ∫_0^a e^{−ikt}(k^2 − n(t))u(t)dt + 1
          (B (+ (/ ((integrate (green-subtransform
                                solution refraction
                                (lambda (x t) (- t)))
                               0 right-bound subintervals) 0)
                   (* 2 +i k))
                1)))
```

```
      (cons A B))))

(define (make-solution refraction right-bound subintervals eps)
  (let* ((k (get-wave-number refraction))
         (initial-solution (cons (wave k)
                                 (cons 0 0))))
    (define (improve solution)
      (let* ((u (lambda (x)
                  (+ ((car solution) x)
                     ((green-integrate
                       (car solution)
                       refraction
                       right-bound subintervals) x))))
             (coeffs (find-A-B
                      u refraction
                      right-bound subintervals)))
        (cons u coeffs)))
    (define (good? solution)
      (let* ((coeffs (cdr solution))
             (A (car coeffs))
             (B (cdr coeffs)))
        (energy-conserves? A B eps)))
    ((iterative-improve good? improve) initial-solution)))

(define (get-solution refraction right-bound subintervals test-
    epsilon)
  (let* ((wave-number (get-wave-number refraction))
         (initial (wave wave))
         (solution (make-solution
                    refraction
                    right-bound subintervals test-epsilon)))
    (cons (tabulate-function (car solution)
                             0 right-bound subintervals)
          (cdr solution))))
```

C Диспетчер

Содержимое файла `dispatcher.scm`

```
(use-modules ((srfi srfi-19) :renamer (symbol-prefix-proc 'srfi
    -19:)))
(use-modules (ice-9 getopt-long))
(use-modules (ice-9 format))

(load "shared.scm")

(define-macro (let-options opts . body)
  (let ,(map (lambda (opt-name)
               `(,opt-name (string->number
                            (option-ref options
                                        ',opt-name
                                        (number->string ,opt-name)
                                            ))))
             opts)
    ,@body))

(define (dispatch args)
  (define (get-seconds time)
    (+ (* 1.0 (/ (srfi-19:time-nanosecond time) (expt 10 9)))
       (srfi-19:time-second time)))
  (define option-spec
    ((method (single-char #\m) (value #t))
     (right-bound (single-char #\a) (value #t))
     (subintervals (single-char #\n) (value #t))
     (statement-file (single-char #\f) (value #t))
     (test-epsilon (single-char #\t) (value #t))))
  (let* ((options (getopt-long args option-spec))
         (statement-file (option-ref options 'statement-file "
             statement.scm")))
    (load-from-path statement-file)
    (let ((method (option-ref options 'method "fundmatrix")))
      (let-options (right-bound subintervals test-epsilon)
                   (load-from-path (string-concatenate
                                    (list method "-solution.scm")))
                   (let* ((start-time (srfi-19:current-time))
                          (solution (get-solution
                                     f right-bound
                                     subintervals
                                     test-epsilon))
                          (end-time (srfi-19:current-time))
                          (time-taken (srfi-19:time-difference end-
                              time start-time)))
                     (print-all-solution solution
                                         right-bound
                                         test-epsilon method
                                         (get-seconds time-taken)))
                                             ))))

(define (print-all-solution solution right-bound test-epsilon used-
    method time-taken)
  (let ((approx (car solution))
        (coeffs (cdr solution)))
```

```
    (print-approximate approx 0 right-bound)
    (display "%%\n")
    (print-A-B coeffs test-epsilon)
    (newline)
    (display (format "method: ~a\n" used-method))
    (display (format "time: ~a\n" time-taken))))

(define (print-approximate solution from to)
  (let ((step (/ (- to from)
                 (length solution))))
    (for-each
     (lambda (n)
       (let ((z (list-ref solution (- n 1))))
         (display (format "~f ~f ~f"
                          (+ from (* (- n 0.5) step))
                          (real-part z)
                          (imag-part z)))
         (newline)))
     (enumerate-n (length solution)))))

(define (print-A-B coeffs test-eps)
  (let ((A (car coeffs))
        (B (cdr coeffs)))
    (display (format "A: ~,5i\n" A))
    (display (format "B: ~,5i\n" B))
    (display (format "conserves: ~a\n"
                     (if (energy-conserves? A B test-eps)
                         "yes"
                         "no")))
    (display (format "eps: ~f" test-eps))))
```

Часть IV
Информация о документе

Данный документ был подготовлен с использованием LaTeX.

Для автоматизации сборки отчёта о курсовой работе применялась сборочная система `GNU Make`. Для извлечения определений процедур из исходного кода использовались сценарии командной оболочки и `GNU Emacs`. С их же помощью был автоматизирован процесс включения результата расчётов в курсовую работу. Графики решений были построены по данным, предоставленным расчётной программой, при помощи средств METAPOST.

Для автоматического определения зависимостей LaTeX-файла использовалась утилита `texdepend`.

Список литературы

[1] *Бахвалов, Н. С.* Численные методы / Н. С. Бахвалов, Н. П. Жидков, Г. М. Кобельков. — М.: Физматлит, 2001.

[2] *Полянин, А. Д.* Справочник по интегральным уравнениям / А. Д. Полянин, А. В. Манжиров. — М.: Физматлит, 2003. — 603 с.