БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
Факультет радиофизики и компьютерных технологий

Реферат

Теоретические основы численных методов реализованных в лабораторном практикуме

Преподаватель:
Мулярчик Степан Григорьевич

Подготовил:
Глеба Евгений Михайлович
2 курс, АРИСТ,
4 группа

Минск, 2022

Содержание

1 РЕШЕНИЕ СИСТЕМ ЛИНЕЙНЫХ АЛГЕБРАИЧЕСИХ УРАВНЕНИЙ

1.1 Метод Гаусса

Метод Гаусса – наиболее мощный и универсальный инструмент для нахождения решения любой системы линейных уравнений. Как мы помним, правило Крамера и матричный метод непригодны в тех случаях, когда система имеет бесконечно много решений или несовместна. А метод последовательного исключения неизвестных в любом случае приведет нас к ответу.

Из прямых методов популярным у вычислителей является этот метод. Поиск главного элемента позволяет, с одной стороны, ограничить рост коэффициентов на каждом шаге исключения и, следовательно, уменьшить влияние ошибок округления на точность решения, с другой, обеспечить для невырожденных систем выполнение условия $a_{kk} \neq 0$ (отсутствие аварийных остановов вследствие деления на нуль).

Пусть исходная матрица выглядит так:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1, \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2, \\ \cdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n = b_n. \end{cases} \tag{1}$$

Процесс её решения делится на два этапа, называемых соответственно прямым и обратным ходом.

На первом этапе система (1) путем последовательного исключения переменных $x_1, x_2, \ldots, x_n$ сводится к эквивалентной системе с верхней треугольной матрицей коэффициентов с единичной диагональю:

$$\begin{aligned} x_1 + u_{12}x_2 + u_{13}x_3 + \cdots + u_{1,n-1}x_{n-1} + u_{1n}x_n &= q_1, \\ x_2 + u_{23}x_3 + \cdots + u_{2,n-1}x_{n-1} + u_{2n}x_n &= q_2, \\ \cdots & \\ x_{n-1} + u_{n-1,n}x_n &= q_{n-1}, \\ x_n &= q_n. \end{aligned} \tag{2}$$

Исключение переменной x_k (k-ый шаг прямого хода Гаусса) включает вычисление k-ой строки треугольной матрицы:

$$u_{kj} = a_{kj}^{(k-1)} / a_{kk}^{(k-1)}, j = \overline{k+1, n}, \tag{3}$$

k-го свободного члена:

$$q_k = b_k^{(k-1)} / a_{kk}^{(k-1)}, \tag{4}$$

преобразование уравнений системы (1) с номерами $\mathrm{k}+1, \mathrm{k}+2, \ldots, \mathrm{n}$:

$$\begin{gathered} a_{ij}^{(k)} = a_{ij}^{(k-1)} - a_{ik}^{(k-1)} u_{kj}; b_i^{(k)} = b_i^{(k-1)} - a_{ik}^{(k-1)} q_k, \\ i = \overline{k+1, n};\ j = \overline{k+1, n}. \end{gathered} \tag{5}$$

На втором этапе (обратный ход) решают систему (2):

$$x_n = q_n; \quad x_k = q_k - \sum_{j=k+1}^{n} u_{kj} x_j; \quad k = \overline{n-1, 1}, \tag{6}$$

последовательно определяя неизвестные $x_n, x_{n-1,\ldots,}x_1$.

Метод Гаусса для плохо обусловленных матриц коэффициентов является вычислительно неустойчивым. Уменьшить вычислительную ошибку можно с помощью метода Гаусса с выделением главного элемента, который является условно устойчивым. Широкое применение метода Гаусса связано с тем, что плохо обусловленные матрицы встречаются на практике относительно редко.

1.1.1 Алгоритм

Алгоритм решения СЛАУ методом Гаусса подразделяется на два этапа:

- На первом этапе осуществляется так называемый прямой ход, когда путём элементарных преобразований над строками систему приводят к ступенчатой или треугольной форме, либо устанавливают, что система несовместна. Для этого среди элементов первого столбца матрицы выбирают ненулевой, перемещают содержащую его строку в крайнее верхнее положение, делая эту строку первой. Далее ненулевые элементы первого столбца всех нижележащих строк обнуляются путём вычитания из каждой строки первой строки, домноженной на отношение первого элемента этих строк к первому элементу первой строки. После того, как указанные преобразования были совершены, первую строку и первый столбец мысленно вычёркивают и продолжают, пока не останется матрица нулевого размера. Если на какой-то из итераций среди элементов первого столбца не нашёлся ненулевой, то переходят к следующему столбцу и проделывают аналогичную операцию.
- На втором этапе осуществляется так называемый обратный ход, суть которого заключается в том, чтобы выразить все получившиеся базисные переменные через небазисные и построить фундаментальную систему решений, либо, если все переменные являются базисными, то выразить в численном виде единственное решение системы линейных уравнений. Эта процедура начинается с последнего уравнения, из которого выражают соответствующую базисную переменную (а она там всего одна) и подставляют в предыдущие уравнения, и так далее, поднимаясь по «ступенькам» наверх. Каждой строчке соответствует ровно одна базисная переменная, поэтому на каждом шаге, кроме последнего (самого верхнего), ситуация в точности повторяет случай последней строки.

Метод Гаусса требует $O(n^3/3)$ арифметических операций.

1.1.2 Реализация алгоритма на языке C#

Листинг 1: Метод Гаусса на языке C#

```
namespace Gauss_Method
{
    class Maths
    {
        /// <summary>
        /// Gauss method
        /// </summary>
        /// <param name="Matrix">Initial matrix</param>
        /// <returns></returns>
        public static double[] Gauss(double[,] Matrix)
        {
            int n = Matrix.GetLength(0); // The dimensionality of
                the initial matrix (row)
            double[,] Matrix_Clone = new double[n, n + 1]; //
                Matrix Dupler
            for (int i = 0; i < n; i++)
                for (int j = 0; j < n + 1; j++)
                    Matrix_Clone[i, j] = Matrix[i, j];
```

```
            // Straight stroke (Lower left corner jamming)
            for (int k = 0; k < n; k++) // k-line number
            {
                for (int i = 0; i < n + 1; i++) // i-number of
                    column
                    Matrix_Clone[k, i] = Matrix_Clone[k, i] /
                        Matrix[k, k]; // Dividing the k-string by
                        the first term !=0 to convert it to one
                for (int i = k + 1; i < n; i++) // i-number of the
                    next line after k
                {
                    double K = Matrix_Clone[i, k] / Matrix_Clone[k
                        , k];
                    for (int j = 0; j < n + 1; j++) // j-number of
                        the column of the next row after k
                        Matrix_Clone[i, j] = Matrix_Clone[i, j] -
                            Matrix_Clone[k, j] * K; // Zoning of
                            matrix elements below the first term
                            converted to one
                }
                for (int i = 0; i < n; i++) // Updating, making
                    changes to the initial matrix
                    for (int j = 0; j < n + 1; j++)
                        Matrix[i, j] = Matrix_Clone[i, j];
            }

            // Reverse stroke (Backing up the upper right corner)
            for (int k = n - 1; k > -1; k--) // k-line number
            {
                for (int i = n; i > -1; i--) // i-number of column
                    Matrix_Clone[k, i] = Matrix_Clone[k, i] /
                        Matrix[k, k];
                for (int i = k - 1; i > -1; i--) // i-number of
                    the next line after k
                {
                    double K = Matrix_Clone[i, k] / Matrix_Clone[k
                        , k];
                    for (int j = n; j > -1; j--) // j-number of
                        the column of the next row after k
                        Matrix_Clone[i, j] = Matrix_Clone[i, j] -
                            Matrix_Clone[k, j] * K;
                }
            }

            // Separating the answers from the total matrix
            double[] Answer = new double[n];
            for (int i = 0; i < n; i++)
                Answer[i] = Matrix_Clone[i, n];

            return Answer;
        }
    }
}
```

1.1.3 Применение и модификации

Помимо аналитического решения СЛАУ, метод Гаусса также применяется для:

- нахождения матрицы, обратной к данной (к матрице справа приписывается единичная такого же размера, что и исходная: $[A|E]$, после чего A приводится к виду единичной матрицы методом Гаусса—Жордана; в результате на месте изначальной единичной матрицы справа оказывается обратная к исходной матрица: $[E|A^{-1}]$
- определения ранга матрицы (согласно следствию из теоремы Кронекера — Капелли ранг матрицы равен числу её главных переменных)
- численного решения СЛАУ в технических приложениях (для уменьшения погрешности вычислений используется Метод Гаусса с выделением главного элемента, суть которого заключена в том, чтобы на каждом шаге в качестве главной переменной выбирать ту, при которой среди оставшихся после вычёркивания очередных строк и столбцов стоит максимальный по модулю коэффициент).

1.1.4 Достоинства метода

- Для матриц ограниченного размера — менее трудоёмкий по сравнению с другими методами.
- Позволяет однозначно установить, совместна система или нет, и если совместна, найти её решение.
- Позволяет найти максимальное число линейно независимых уравнений — ранг матрицы системы.

1.1.5 Устойчивость метода Гаусса

Метод Гаусса для плохо обусловленных матриц коэффициентов является вычислительно неустойчивым. Например, для матриц Гильберта метод приводит к очень большим ошибкам даже при небольшой размерности этих матриц. Уменьшить вычислительную ошибку можно с помощью метода Гаусса с выделением главного элемента, который является условно устойчивым. Широкое применение метода Гаусса связано с тем, что плохо обусловленные матрицы встречаются на практике относительно редко.

1.1.6 Неоптимальность метода Гаусса

В 1969 году Штрассен доказал, что большие матрицы можно перемножить за время $O(n^{log_2 7}) = O(n^{2.81})$. Отсюда вытекает, что обращение матриц и решение СЛАУ можно осуществлять алгоритмами асимптотически более быстрыми по порядку, чем метод Гаусса. Таким образом, для больших СЛАУ метод Гаусса не оптимален по скорости.

1.2 Метод $\bar{L}D\bar{L}^T-$ факторизации

Иногда удобнее бывает рассматривать $\bar{L}D\bar{L}^T$ вариант симметричного треугольного разложения, в котором матрица L является нижней унитреугольной (т.е. имеет единицы на главной диагонали), а D - диагональная матрица с положительными элементами.

Здесь мы опишем методы, использующие специфику при решении задачи $Ax = b$. В случае, когда A — симметричная невырожденная матрица, т.е. $A = A^T$ и $\det(A) \neq 0$, существует разложение вида:

$$A = LDL^T, \tag{7}$$

где L — нижняя унитреугольная матрица, D — диагональная матрица. В связи с этим работа связанная с получением разложения :eq:$sles - ldl$, составляет половину от того, что требуется для исключения Гаусса. Когда разложение :eq:$sles - ldl$ получено, решение системы $Ax = b$ может быть найдено посредством решения систем $Ly = b$ (прямая подстановка), $Dz = y$ и $L^T x = z$.

1.2.1 LDL^T-разложение

Разложение (7) может быть найдено при помощи исключения Гаусса, вычисляющего $A = LU$, с последующим определением D из уравнения $U = DL^T$. Тем не менее можно использовать интересный альтернативный алгоритм непосредственного вычисления L и D.

Допустим, что мы знаем первые $j-1$ столбцов матрицы L, диагональные элементы $d_1, d_2, \ldots, d_{j-1}$ матрицы D для некоторого j, $1 \le j \le n$. Чтобы получить способ вычисления l_{ij}, $i = j+1, j+2, \ldots, n$, и d_j приравняем j-е столбцы в уравнении $A = LDL^T$. В частности,

$$A(1:j, j) = Lv, \tag{8}$$

где

$$v = DL^T e_j = \begin{bmatrix} d_1 l_{j1} \\ \vdots \\ d_{j-1} l_{jj-1} \\ d_j \end{bmatrix}$$

Следовательно, компоненты v_k, $k = 1, 2, \ldots, j-1$ вектора v могут быть получены простым масштабированием элементов j-й строки матрицы L. Формула для j-й компоненты вектора v получается из j-го уравнения системы $L(1:j, 1:j)v = A(1:j, j)$:

$$v_j = a_{jj} - \sum_{k=1}^{j-1} l_{ik} v_k,$$

Когда мы знаем v, мы вычисляем $d_j = v_j$. «Нижняя» половина формулы (8) дает уравнение

$$L(j+1:n, 1:j)v(1:j) = A(j+1:n, j),$$

откуда для вычисления j-го столбца матрицы L имеем:

$$L(j+1:n, j) = (A(j+1:n, j) - L(j+1:n, 1:j-1)v(1:j-1))/v_j.$$

1.2.2 Реализация на языке Python

Для получения LDL^T-разложения матрицы A можем написать функцию:

Листинг 2: LDL^T разложение на языке python

```
def ld(A):
    """
    For a symmetric matrix A calculates the lower triangular
    matrix L and the diagonal matrix D such that
    that A = LDL^T. The elements a_{ij} are replaced
    by l_{ij} if i > j, and by d_i if i = j
    """
    n = len(A)
LD = np.array(A, float)
 for j in range(n):
  v = np.zeros(j+1)
  v[:j] = LD[j, :j]*LD[range(j), range(j)]
  v[j] = LD[j, j] - np.dot(LD[j, :j], v[:j])
  LD[j, j] = v[j]
  LD[j+1:, j] = (LD[j+1:, j] - np.dot(LD[j+1:, :j], v[:j]))/v[j]

 return LD
```

В этой реализации мы использовали векторизованные вычисления. Разберем некоторые выражения. Строка

```
v[:j] = LD[j,:j]*LD[range(j),range(j)]
```

заменяется следующим циклом:

```
for i in range(j):
    v[i] = LD[j,i]*LD[i,i]
```

В нашей программе доступ к `j` диагональным элементам массива `A` осуществляется выражением `A[range(j),range(j)]`.

При вычислении `v[j]` использовалась функция `np.dot`, которая вычисляет скалярное произведение векторов.

тметим также строку

```
LD[j+1:,j] = (LD[j+1:,j] - np.dot(LD[j+1:,:j],v[:j]))/v[j]
```

в которой используется срез `L[j+1:,j]`, т.е. элементы с `j+1`-го до последнего в `j`-ом столбце.

Для решения системы $Ax = b$‘ с использованием LDL^T-разложения можно написать следующую функцию:

Листинг 3: Решение LDL^T системы на языке python

```
def ld_solve(A, b):
 """
 Solves the system Ax = b using LDL^T-decomposition
 """
 LD = ld(A)
 b = np.array(b, float)
 for i in range(1, len(b)):
  b[i] = b[i] - np.dot(LD[i, :i], b[:i])
 b[:] = b[:]/LD[range(len(b)), range(len(b))]
 for i in range(len(b)-1, -1, -1):
  b[i] = (b[i] - np.dot(LD[i+1:, i], b[i+1:]))
 return b
```

2 РЕШЕНИЕ СИСТЕМ НЕЛИНЕЙНЫХ АЛГЕБРАИЧЕСКИХ УРАВНЕНИЙ

Метод Ньютона, алгоритм Ньютона (также известный как метод касательных) — это итерационный численный метод нахождения корня (нуля) заданной функции. Метод был впервые предложен английским физиком, математиком и астрономом Исааком Ньютоном (1643—1727). Поиск решения осуществляется путём построения последовательных приближений и основан на принципах простой итерации. Метод обладает квадратичной сходимостью. Модификацией метода является метод хорд и касательных. Также метод Ньютона может быть использован для решения задач оптимизации, в которых требуется определить ноль первой производной либо градиента в случае многомерного пространства.

2.1 Описание метода

Чтобы численно решить уравнение $f(x) = 0$, его необходимо привести к эквивалентному уравнению: $x = \varphi(x)$, где φ - сжимающее отображение.

Для наилучшей сходимости метода в точке очередного приближения x^* должно выполняться условие $\varphi'(x^*) = 0$ Решение данного уравнения ищут в виде $\varphi(x) = x + \alpha(x)f(x)$ тогда:

$$\varphi'(x^*) = 1 + \alpha'(x^*)f(x^*) + \alpha(x^*)f'(x^*) = 0 \tag{9}$$

В предположении, что точка приближения «достаточно близка» к корню $\widetilde{x}$ и что заданная функция ($f(x^*) \approx f(\widetilde{x}) = 0$) окончательная формула для $\alpha(x)$ такова:

$$\alpha(x) = -\frac{1}{f'(x)} \tag{10}$$

С учётом этого функция $\varphi(x)$ определяется:

$$\varphi(x) = x - \frac{f(x)}{f'(x)} \tag{11}$$

При некоторых условиях эта функция в окрестности корня осуществляет сжимающее отображение.

2.1.1 Доказательство

Пусть дана вещественного переменного дважды непрерывно дифференцируемая в своей области определения, производная которой нигде не обращается в нуль:

$$f(x) : \mathbb{X} \to \mathbb{R},\ f(x) \in C^2(\mathbb{X}); \quad \forall x \in \mathbb{X}\ f'(x) \neq 0.$$

И необходимо доказать, что функция $\varphi(x) = x - \frac{f(x)}{f'(x)}$ осуществляет сжимающее отображение вблизи корня уравнения $f(x) = 0$.

В силу непрерывной дифференцируемости функции $f(x)$ и неравенства нулю её первой производной $\varphi(x)$ непрерывна.

Производная $\varphi'(x)$ равна:

$$\varphi'(x) = \frac{f(x)f''(x)}{(f'(x))^2}.$$

В условиях, наложенных на $f(x)$, она также непрерывна. Пусть $\widetilde{x}$ искомый корень уравнения: $f(\widetilde{x}) = 0$, следовательно в его окрестности $\varphi'(x) \approx 0$:

$$\forall \varepsilon : 0 < \varepsilon < 1,\ \exists \delta > 0\ \forall x \in \mathbb{X}\ |x - \widetilde{x}| < \delta : |\varphi'(x) - 0| < \varepsilon.$$

Тогда согласно теореме Лагранжа:

$$\forall x_1,\ x_2 \in U_\delta(\widetilde{x})\ \exists \xi \in U_\delta(\widetilde{x}) : |\varphi(x_1) - \varphi(x_2)| = |\varphi'(\xi)||x_1 - x_2| < \varepsilon |x_1 - x_2|.$$

В силу того, что $\varphi(\widetilde{x}) = \widetilde{x}$ в этой же дельта окрестности выполняется:

$$\forall x \in U_\delta(\widetilde{x}) :\ |\varphi(x) - \widetilde{x}| < \varepsilon |x - \widetilde{x}|.$$

Таким образом полученная функция $\varphi(x)$ в окрестности корня $U_\delta(\widetilde{x})$ осуществляет сжимающее отображение.

Поэтому алгоритм нахождения численного решения уравнения $f(x) = 0$ сводится к итерационной процедуре вычисления:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

По теореме Банаха последовательность приближений стремится к корню уравнения $f(x) = 0$.

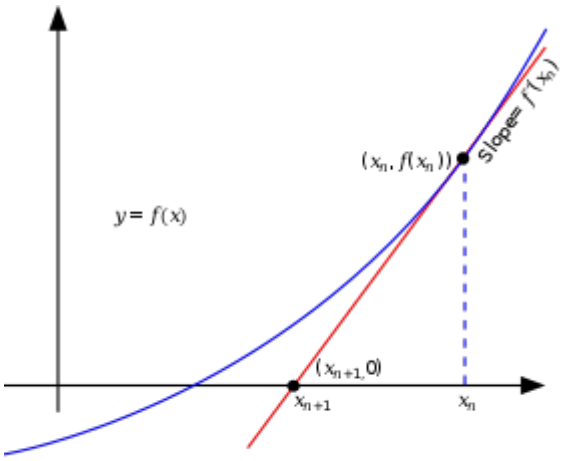

Рис. 1: *Иллюстрация метода Ньютона (синим изображена функция $f(x)$, ноль которой необходимо найти, красным - касательная в точке очередного приближения x_n Здесь мы можем увидеть, что последующее приближение x_{n+1} лучше предыдущего x_n.*

2.1.2 Ограничения

Пусть задано уравнение $f(x) = 0$, где $f(x) : \mathbb{X} \to \mathbb{R}$ и надо найти его решение.

Ниже приведена формулировка основной теоремы, которая позволяет дать чёткие условия применимости. Она носит имя советского и (—).

Теорема Канторовича.

Если существуют такие константы $A,\ B,\ C$, что:

1. $\frac{1}{|f'(x)|} < A$ *на* $[a,\ b]$, *то есть* $f'(x)$ *существует и не равна нулю;*
2. $\left|\frac{f(x)}{f'(x)}\right| < B$ *на* $[a,\ b]$, *то есть* $f(x)$ *ограничена;*
3. $\exists f''(x)$ *на* $[a,\ b]$, *и* $|f''(x)| \leqslant C \leqslant \frac{1}{2AB}$;

Причём длина рассматриваемого отрезка $|a-b| < \frac{1}{AB}\left(1 - \sqrt{1 - 2ABC}\right)$. *Тогда справедливы следующие утверждения:*

1. *на* $[a,\ b]$ *существует корень* x^* *уравнения* $f(x) = 0 : \exists x^* \in [a,\ b] : f(x^*) = 0$;

2. *если* $x_0 = \frac{a+b}{2}$, *то итерационная сходится к этому корню:* $\left\{x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}\right\} \to x^*$;

3. *погрешность может быть оценена по формуле* $|x^* - x_n| \leqslant \frac{B}{2^{n-1}}(2ABC)^{2^{n-1}}$.

Из последнего из утверждений теоремы в частности следует квадратичная метода: $|x^* - x_n| \leqslant \frac{B}{2^{n-1}}(2ABC)^{2^{n-1}} = \frac{1}{2}\frac{B}{2^{n-2}}\left((2ABC)^{2^{n-2}}\right)^2 = \alpha|x^* - x_{n-1}|^2$.
Тогда ограничения на исходную функцию $f(x)$ будут выглядеть так:

1. функция должна быть ограничена;
2. функция должна быть , дважды ;
3. её первая $f'(x)$ равномерно отделена от нуля;
4. её вторая производная $f''(x)$ должна быть равномерно ограничена.

2.2 Обобщения и модификации

2.2.1 Многомерный случай

Обобщим полученный результат на многомерный случай.
Пусть необходимо найти решение системы:

$$\begin{cases} f_1(x_1,\ x_2,\ \ldots,\ x_n) &= 0, \\ \cdots & \\ f_m(x_1,\ x_2,\ \ldots,\ x_n) &= 0. \end{cases}$$

Выбирая некоторое начальное значение $\vec{x}^{[0]}$, последовательные приближения $\vec{x}^{[j+1]}$ находят путём решения систем уравнений:

$$f_i + \sum_{k=1}^{n} \frac{\partial f_i}{\partial x_k}(x_k^{[j+1]} - x_k^{[j]}) = 0, \qquad i = 1,\ 2,\ \ldots,\ m,$$

где

$$\vec{x}^{[j]} = (x_1^{[j]},\ x_2^{[j]},\ \ldots,\ x_n^{[j]}), \quad j = 0,\ 1,\ 2,\ \ldots.$$

2.2.2 Метод секущих

Родственный является «приближённым» методом Ньютона и позволяет **не вычислять** производную. Значение производной в итерационной формуле заменяется её оценкой по двум предыдущим точкам итераций:

$$f'(x_n) \approx \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}$$

Таким образом, основная формула имеет вид

$$x_{n+1} = x_n - f(x_n) \cdot \frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}.$$

Этот метод схож с методом Ньютона, но имеет немного меньшую скорость сходимости. Порядок сходимости метода равен $\ — 1{,}618\ldots$

Замечания. 1) Для начала итерационного процесса требуются два различных значения x_0 и x_1 .
2) В отличие от «настоящего метода Ньютона» (метода касательных), требующего хранить только x_n (и в ходе вычислений — временно $f(x_n)$ и $f'(x_n)$), для метода секущих требуется сохранение x_{n-1} , x_n , $f(x_{n-1})$, $f(x_n)$.
3) Применяется, если вычисление $f'(x)$ затруднено (например, требует большого количества машинных ресурсов: времени и/или памяти).

2.2.3 Метод Ньютона - Рафсона

Метод Ньютона — Рафсона является улучшением метода Ньютона нахождения экстремума, описанного выше. Основное отличие заключается в том, что на очередной итерации каким-либо из выбирается оптимальный шаг:

$\vec{x}^{[j+1]} = \vec{x}^{[j]} - \lambda_j H^{-1}(\vec{x}^{[j]})\nabla f(\vec{x}^{[j]})$,

где $\lambda_j = \arg\min_{\lambda} f(\vec{x}^{[j]} - \lambda H^{-1}(\vec{x}^{[j]})\nabla f(\vec{x}^{[j]}))$. Для оптимизации вычислений применяют следующее улучшение: вместо того, чтобы на каждой итерации заново вычислять, ограничиваются начальным приближением $H(f(\vec{x}^{[0]}))$ и обновляют его лишь раз в m шагов, либо не обновляют вовсе.

2.3 Алгоритм

1. Задается начальное приближение x_0.
2. Пока не выполнено условие остановки, в качестве которого можно взять $|x_{n+1} - x_n| < \varepsilon$ или $|f(x_{n+1})| < \varepsilon$ (то есть погрешность в нужных пределах), вычисляют новое приближение: $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$.

2.3.1 Реализация на языке Haskell

Листинг 4: Метод Ньютона на языке *Haskell*

```
import Data.List ( iterate' )

main :: IO ()
main = print $ solve (\ x -> x * x - 17) ( * 2) 4

-- The solve function is universal for all real types whose values
    can be compared.
solve = esolve 0.000001

esolve epsilon func deriv x0 = fst . head $ dropWile pred pairs
  where
    pred (xn, xn1) = (abs $ xn - xn1) > epsilon -- The pred
       function determines whether the required precision is
       achieved.
    next xn = xn - func xn / deriv xn -- The next function
       calculates a new approximation.
    iters = iterate' next x0 -- An infinite list of iterations.
    pairs = zip iters (tail iters) -- An infinite list of
       iteration pairs of the form: [(x0, x1), (x1, x2) ...].
```

3 РЕШЕНИЕ СИСТЕМ ОБЫКНОВЕННЫХ ДИФФЕРЕНЦИАЛЬНЫХ УРАВНЕНИЙ

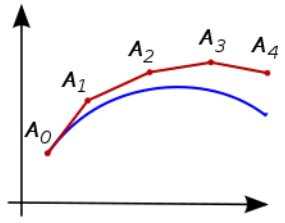

Рис. 2: *Ломаная Эйлера — приближённое решение в пяти узлах задачи Коши и точное решение этой задачи*

Метод Эйлера — простейший численный метод решения систем обыкновенных дифференциальных уравнений. Впервые описан Леонардом Эйлером в 1768 году в работе «Интегральное исчисление». Метод Эйлера является явным, одношаговым методом первого порядка точности. Он основан на аппроксимации интегральной кривой кусочно-линейной функцией, так называемой ломаной Эйлера.

3.1 Описание метода

Пусть дана для уравнения первого порядка:

$$\frac{dy}{dx} = f(x, y),\ y_{|x=x_0} = y_0,$$

где функция f определена на некоторой области $D \subset \mathbb{R}^2$. Решение ищется на полуинтервале $(x_0, b]$. На этом промежутке введём узлы: $x_0 < x_1 < \cdots < x_n \leq b$. Приближенное решение в узлах x_i, которое обозначим через y_i, определяется по формуле:

$$y_i = y_{i-1} + (x_i - x_{i-1}) f(x_{i-1}, y_{i-1}), \quad i = 1, 2, 3, \ldots, n.$$

Эти формулы непосредственно обобщаются на случай систем обыкновенных дифференциальных уравнений.

3.1.1 Оценка погрешности метода на шаге и в целом

Погрешность на шаге или локальная погрешность — это разность между численным решением после одного шага вычисления y_i и точным решением в точке $x_i = x_{i-1} + h$. Численное решение задаётся формулой

$$y_i = y_{i-1} + h f(x_{i-1}, y_{i-1}).$$

Точное решение можно разложить в :

$$y(x_{i-1} + h) = y(x_{i-1}) + h y'(x_{i-1}) + O(h^2).$$

Локальную ошибку L получаем, вычитая из второго равенства первое:

$$L = y(x_{i-1} + h) - y_i = O(h^2).$$

Это справедливо, если y имеет непрерывную вторую производную. Другим достаточным условием справедливости этой оценки, из которого вытекает предыдущее и которое обычно может быть легко проверено, является непрерывная дифференцируемость $f(x, y)$ по обоим аргументам.

Погрешность в целом, глобальная или накопленная погрешность — это погрешность в последней точке произвольного конечного отрезка интегрирования уравнения. Для вычисления решения в этой точке требуется S/h шагов, где S длина отрезка. Поэтому глобальная погрешность метода $G = O(h^2 S/h) = O(h)$.

Таким образом, метод Эйлера является методом первого порядка — имеет погрешность на шаге $O(h^2)$ и погрешность в целом $O(h)$.

3.2 Явный метод Эйлера

Формула интегрирования явного метода Эйлера имеет вид:

$$y_{k+1} = y_k + \tau_k f\left(y_k, t_k\right), \quad k = 0, 1, 2, \ldots$$

где τ_{k} - шаг интегрирования. Его локальная погрешность

$$\varepsilon_k = \frac{\tau_k^2}{2} u''(\tilde{t}) \quad \tilde{t} \in [t_k; t_{k+1}]$$

пропорциональна τ_k^2, т. е. явный метод Эйлера имеет первый порядок точности. Приведем геометрическую интерпретацию явного метода Эйлера для задачи Коши: приращение на шаре интегрирования метода - катет значению производной в предыдущий момент времени. Вторым катетом этого треугольника является текущий шаг интегрирования.

Явный метод Эйлера для СОДУ является условно устойчивым с условием:

$$|1 + \tau\lambda_i| < 1, i = \overline{1, n}$$

где λ_i- собственные числа матрицы J = df/dx. Поэтому выбор величины шага интегрирования в явном методе Эйлера необходимо делать, исходя из сохранения устойчивости и точности вычислений.

Шаг мы вычисляем по формулам:

$$\tau_k^i \leq \varepsilon_{i,0n}^i / \left[|f^i(\mathbf{y}_k, t_k)| + \varepsilon_{n,0n}^i / \tau_{\max}\right]$$

Из (3) вытекает:

$$\tau_k = \min_{i=1,n} \tau_k^i$$

3.2.1 Реализация на языке Python

Листинг 5: Явный метод Эйлера на языке Python

```
def ForwardEuler(F, u0, tau, T):
    """
    > The function 'ForwardEuler' takes as input a function 'F',
       an initial condition 'u0', a time step 'tau', and a final
    time 'T', and returns the numerical solution 'u' and the time
       points 't'

    :param F: the function that defines the differential equation
    :param u0: initial condition
    :param tau: time step
    :param T: the total time of the simulation
    :return: the solution u and the time steps t.
    """
    N_t = int(round(T / tau))
    F_ = lambda t, u: np.asarray(F(t, u))
    t = np.linspace(0, N_t * tau, N_t + 1)
    u = np.zeros((N_t + 1, len(u0)))
    u[0] = np.array(u0)
    for n in range(N_t):
        u[n + 1] = u[n] + tau * F_(t[n], u[n])

    return u, t
```

3.3 Неявный метод Эйлера

Разностная схема неявного метода Эйлера выглядит следующим образом:

$$\mathbf{y}_{k+1} = \mathbf{y}_k + \tau_k \mathbf{f}\left(\mathbf{y}_{k+1}, t_{k+1}\right), \quad k = 0, 1, 2, \ldots$$

Она требует на каждом временном шаге решения следующей алгебраической задачи:

$$\mathbf{F}\left(\mathbf{y}_{k+1}\right) = \mathbf{y}_{k+1} - \mathbf{y}_k - \tau_k \mathbf{f}\left(\mathbf{y}_{k+1}, t_{k+1}\right) = 0$$

которую можно решить, например, методом Ньютона.

Локальная погрешность неявного метода Эйлера

$$\varepsilon_k = -\frac{\tau_k^2}{2} u''(\tilde{t}) \quad \tilde{t} \in [t_k; t_{k+1}]$$

по порядку величины τk является такой же, как и локальная погрешность явного метода Эйлера, и противоположна по знаку. После выполнения шага интегрирования τ_k вычисляются ε_k^i по формуле ε_k для всех переменных u^i. Если хотя бы одно $|\varepsilon_k^i| > \varepsilon_{\text{доп}}^i$ то шаг τ_k уменьшается вдвое, и вычисления повторяются.

При выборе шага интегрирования можно использовать также стратегию трёх зон:

В обоих стратегиях окончательный шаг интегрирования выбирается минимальным среди всех значений τ_{k+1}^i.

Неявный метод Эйлера является абсолютно устойчивым по отношению к шагу интегрирования. При решении этим методом жёстких систем дифференциальных уравнений шаг интегрирования выбирается только из соображений допустимой локальной погрешности.

3.3.1 Реализация на языке Python

Листинг 6: Неявный метод Эйлера на языке Python

```
def BackwardEuler(F, u0, tau, T):
    """
    > The function 'BackwardEuler' takes as input a function 'F',
       an initial condition 'u0', a time step 'tau', and a final
    time 'T', and returns the numerical solution 'u' and the time
       points 't'

    :param F: the function that defines the ODE
    :param u0: initial condition
    :param tau: time step
    :param T: the total time of the simulation
    :return: the solution u and the time points t.
    """
    N_t = int(round(T / tau))
    F_ = lambda t, u: np.asarray(F(t, u))
    t = np.linspace(0, N_t * tau, N_t + 1)
    u = np.zeros((N_t + 1, len(u0)))
    u[0] = np.array(u0)

    def Phi(z, t, v):
        return z - tau * F_(t, z) - v

    for n in range(N_t):
        u[n + 1] = optimize.fsolve(Phi, u[n], args=(t[n], u[n]))

    return u, t
```

3.4 Модификации и обобщения

3.4.1 Модифицированный метод Эйлера с пересчетом

Повысить точность и устойчивость вычисления решения можно с помощью явного метода Эйлера следующего вида.

Прогноз:

$$\widetilde{y}_i = y_{i-1} + (x_i - x_{i-1}) f(x_{i-1}, y_{i-1}).$$

Коррекция:

$$y_i = y_{i-1} + (x_i - x_{i-1}) \frac{f(x_{i-1}, y_{i-1}) + f(x_i, \widetilde{y}_i)}{2}.$$

Для повышения точности корректирующую итерацию можно повторить, подставляя $\widetilde{y}_i = y_i$.

Модифицированный метод Эйлера с пересчетом имеет второй порядок точности, однако для его реализации необходимо как минимум дважды вычислять $f(x, y)$. Метод Эйлера с пересчетом представляет собой разновидность (предиктор-корректор).

3.4.2 Двухшаговый метод Адамса — Башфорта

Другой способ повысить точность метода заключается в использовании не одного, а нескольких вычисленных ранее значений функции:

$$y_{i+1} = y_i + \tfrac{3}{2} h f(x_i, y_i) - \tfrac{1}{2} h f(x_{i-1}, y_{i-1}).$$

Это линейный многошаговый метод.

4 ПРИБЛИЖЕНИЕ ФУНКЦИИ МЕТОДОМ НАИМЕНЬШИХ КВАДРАТОВ

Метод наименьших квадратов (МНК) — математический метод, применяемый для решения различных задач, основанный на минимизации суммы квадратов отклонений некоторых функций от экспериментальных входных данных. Он может использоваться для «решения» переопределенных систем уравнений (когда количество уравнений превышает количество неизвестных), для поиска решения в случае обычных (не переопределенных) нелинейных систем уравнений, для аппроксимации точечных значений некоторой функции. МНК является одним из базовых методов регрессионного анализа для оценки неизвестных параметров регрессионных моделей по выборочным данным.

4.1 Суть метода наименьших квадратов

Пусть y_i, $i = 1, ...n$ набор скалярных экспериментальных данных, x_i, $i = 1, ...n$ набор векторных экспериментальных данных и предполагается, что y зависит от x.

Вводится некоторая (в простейшем случае линейная) скалярная функция $f(x, \beta)$, которая определяется вектором неизвестных параметров β.

Ставится задача найти вектор β такой, чтобы совокупность погрешностей $r_i = y_i - f(x_i, \beta)$ была в некотором смысле минимальной.

Согласно методу наименьших квадратов решением этой задачи является вектор β, который минимизирует функцию

$$S(\beta) = \sum_{i=1}^{n} (y_i - f(x_i, \beta))^2.$$

В простейшем случае $f(x) = \beta$, и тогда результатом МНК будет входных данных.

Преимущество МНК перед минимизацией других видов ошибок состоит в том, что если $f(x, \beta)$ дифференцируема по β, то $S(\beta)$ тоже дифференцируема. Приравнивание частных производных к нулю сводит задачу к решению системы уравнений, причём если $f(x, \beta)$ зависит от β линейно, то и система уравнений будет линейной.

4.2 МНК в регрессионном анализе (аппроксимация данных)

Пусть имеется n значений некоторой переменной y (это могут быть результаты наблюдений, экспериментов и т. д.) и соответствующих переменных x. Задача заключается в том, чтобы взаимосвязь между y и x аппроксимировать некоторой функцией $f(x, b)$, известной с точностью до некоторых неизвестных параметров b, то есть фактически найти наилучшие значения параметров b, максимально приближающие значения $f(x, b)$ к фактическим значениям y. Фактически это сводится к случаю «решения» переопределенной системы уравнений относительно b:

$$f(x_t, b) = y_t, t = 1, \ldots, n.$$

В регрессионном анализе и в частности в эконометрике используются вероятностные модели зависимости между переменными

$$y_t = f(x_t, b) + \varepsilon_t,$$

где ε_t — так называемые *случайные ошибки* модели.

Соответственно, отклонения наблюдаемых значений y от модельных $f(x, b)$ предполагается уже в самой модели. Сущность МНК (обычного, классического) заключается в том, чтобы найти такие параметры b, при которых сумма квадратов отклонений (ошибок, для регрессионных моделей их часто называют *остатками регрессии*) e_t будет минимальной:

$$\hat{b}_{OLS} = \arg\min_b RSS(b),$$

где RSS — Residual Sum of Squares определяется как:

$$RSS(b) = e^T e = \sum_{t=1}^{n} e_t^2 = \sum_{t=1}^{n} (y_t - f(x_t, b))^2.$$

В общем случае решение этой задачи может осуществляться численными методами оптимизации (минимизации). В этом случае говорят о *нелинейном МНК* (NLS или NLLS — Non-Linear Least Squares). Во многих случаях можно получить аналитическое решение. Для решения задачи минимизации необходимо найти стационарные точки функции $RSS(b)$, продифференцировав её по неизвестным параметрам b, приравняв производные к нулю и решив полученную систему уравнений:

$$\sum_{t=1}^{n} (y_t - f(x_t, b)) \tfrac{\partial f(x_t, b)}{\partial b} = 0.$$

4.2.1 МНК в случае линейной регрессии

Пусть регрессионная зависимость является линейной:

$$y_t = \sum_{j=1}^{k} b_j x_{tj} + \varepsilon = x_t^T b + \varepsilon_t.$$

Пусть y — вектор-столбец наблюдений объясняемой переменной, а X — это $(n \times k)$-матрица наблюдений факторов (строки матрицы — векторы значений факторов в данном наблюдении, по столбцам — вектор значений данного фактора во всех наблюдениях). Матричное представление линейной модели имеет вид:

$$y = Xb + \varepsilon.$$

Тогда вектор оценок объясняемой переменной и вектор остатков регрессии будут равны

$$\hat{y} = Xb, \quad e = y - \hat{y} = y - Xb.$$

соответственно сумма квадратов остатков регрессии будет равна

$$RSS = e^T e = (y - Xb)^T (y - Xb).$$

Дифференцируя эту функцию по вектору параметров b и приравняв производные к нулю, получим систему уравнений (в матричной форме):

$$(X^T X) b = X^T y.$$

В расшифрованной матричной форме эта система уравнений выглядит следующим образом:

$$\begin{pmatrix} \sum x_{t1}^2 & \sum x_{t1}x_{t2} & \sum x_{t1}x_{t3} & \dots & \sum x_{t1}x_{tk} \\ \sum x_{t2}x_{t1} & \sum x_{t2}^2 & \sum x_{t2}x_{t3} & \dots & \sum x_{t2}x_{tk} \\ \sum x_{t3}x_{t1} & \sum x_{t3}x_{t2} & \sum x_{t3}^2 & \dots & \sum x_{t3}x_{tk} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \sum x_{tk}x_{t1} & \sum x_{tk}x_{t2} & \sum x_{tk}x_{t3} & \dots & \sum x_{tk}^2 \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_k \end{pmatrix} = \begin{pmatrix} \sum x_{t1}y_t \\ \sum x_{t2}y_t \\ \sum x_{t3}y_t \\ \vdots \\ \sum x_{tk}y_t \end{pmatrix},$$

где все суммы берутся по всем допустимым значениям t.

Если в модель включена константа (как обычно), то $x_{t1} = 1$ при всех t, поэтому в левом верхнем углу матрицы системы уравнений находится количество наблюдений n, а в остальных элементах первой строки и первого столбца — просто суммы значений переменных: $\sum x_{tj}$ и первый элемент правой части системы — $\sum y_t$.

Решение этой системы уравнений и дает общую формулу МНК-оценок для линейной модели:

$$\hat{b}_{OLS} = (X^T X)^{-1} X^T y = \left(\tfrac{1}{n} X^T X\right)^{-1} \tfrac{1}{n} X^T y = V_x^{-1} C_{xy}.$$

Для аналитических целей оказывается полезным последнее представление этой формулы (в системе уравнений при делении на n вместо сумм фигурируют средние арифметические). Если в регрессионной модели данные *центрированы*, то в этом представлении первая матрица имеет смысл выборочной ковариационной матрицы факторов, а вторая — вектор ковариаций факторов с зависимой переменной. Если кроме того данные ещё и *нормированы* на СКО (то есть в конечном итоге *стандартизированы*), то первая матрица имеет смысл выборочной корреляционной матрицы факторов, второй вектор — вектора выборочных корреляций факторов с зависимой переменной.

Немаловажное свойство МНК-оценок для моделей *с константой* — линия построенной регрессии проходит через центр тяжести выборочных данных, то есть выполняется равенство:

$$\overline{y} = \hat{b_1} + \sum_{j=2}^{k} \hat{b}_j \overline{x}_j.$$

В частности, в крайнем случае, когда единственным регрессором является константа, получаем, что МНК-оценка единственного параметра (собственно константы) равна среднему значению объясняемой переменной. То есть среднее арифметическое, известное своими хорошими свойствами из законов больших чисел, также является МНК-оценкой — удовлетворяет критерию минимума суммы квадратов отклонений от неё.

4.2.2 Простейшие частные случаи

В случае парной линейной регрессии $y_t = a + bx_t + \varepsilon_t$, когда оценивается линейная зависимость одной переменной от другой, формулы расчёта упрощаются (можно обойтись без матричной алгебры). Система уравнений имеет вид:

$$\begin{pmatrix} n & \sum\limits_{t=1}^{n} x_t \\ \sum\limits_{t=1}^{n} x_t & \sum\limits_{t=1}^{n} x_t^2 \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} \sum\limits_{t=1}^{n} y_t \\ \sum\limits_{t=1}^{n} x_t y_t \end{pmatrix}.$$

Отсюда несложно найти оценки коэффициентов:

$$\begin{cases} \hat{b} = \dfrac{n \sum\limits_{t=1}^{n} x_t y_t - \left(\sum\limits_{t=1}^{n} x_t\right)\left(\sum\limits_{t=1}^{n} y_t\right)}{n \sum\limits_{t=1}^{n} x_t^2 - \left(\sum\limits_{t=1}^{n} x_t\right)^2}, \\ \hat{a} = \dfrac{\sum\limits_{t=1}^{n} y_t - \hat{b} \sum\limits_{t=1}^{n} x_t}{n}. \end{cases}$$

Несмотря на то, что в общем случае модели с константой предпочтительней, в некоторых случаях из теоретических соображений известно, что константа a должна быть равна нулю. Например, в физике зависимость между напряжением и силой тока имеет вид $U = I \cdot R$; замеряя напряжение и силу тока, необходимо оценить сопротивление. В таком случае речь идёт о модели $y = bx$. В этом случае вместо системы уравнений имеем единственное уравнение

$$\left(\sum x_t^2\right) b = \sum x_t y_t.$$

Следовательно, формула оценки единственного коэффициента имеет вид

$$\hat{b} = \frac{\sum\limits_{t=1}^{n} x_t y_t}{\sum\limits_{t=1}^{n} x_t^2}.$$

4.3 Реализация на языке Python

Листинг 7: МНК на языке Python

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

def lstsq(a, b):
    a, _ = _makearray(a)
    b, wrap = _makearray(b)
    is_1d = b.ndim == 1
    if is_1d:
        b = b[:, newaxis]
    _assert_2d(a, b)
    m, n = a.shape[-2:]
    m2, n_rhs = b.shape[-2:]
    if m != m2:
        raise LinAlgError('Incompatible dimensions')

    t, result_t = _commonType(a, b)
    result_real_t = _realType(result_t)

    if m <= n:
        gufunc = _umath_linalg.lstsq_m
    else:
        gufunc = _umath_linalg.lstsq_n

    signature = 'DDd->Ddid' if isComplexType(t) else 'ddd->ddid'
    extobj = get_linalg_error_extobj(_raise_linalgerror_lstsq)
    if n_rhs == 0:
        # lapack can't handle n_rhs = 0 - so allocate the array
            one larger in that axis
        b = zeros(b.shape[:-2] + (m, n_rhs + 1), dtype=b.dtype)
    x, resids, rank, s = gufunc(a, b, signature=signature, extobj=
        extobj)
    if m == 0:
        x[...] = 0
    if n_rhs == 0:
        # remove the item we added
        x = x[..., :n_rhs]
        resids = resids[..., :n_rhs]

    # remove the axis we added
    if is_1d:
        x = x.squeeze(axis=-1)
        # we probably should squeeze resids too, but we can't
        # without breaking compatibility.

    # as documented
    if rank != n or m <= n:
        resids = array([], result_real_t)

    # coerce output arrays
    s = s.astype(result_real_t, copy=False)
    resids = resids.astype(result_real_t, copy=False)
    x = x.astype(result_t, copy=True)  # Copying lets the memory
        in r_parts be freed
```

```
    return wrap(x), wrap(resids), rank, s

def generate_task_data():
    x = np.arange(60, 130, 10)
    y = np.array([0.0148, 0.0124, 0.0102, 0.0085, 0.0071, 0.0059,
       0.0051])

    return x, y


def process():
    x, y = generate_task_data()

    # Let's fit the data after we applied the log trick.
    A = np.vstack([x, np.ones(len(x))]).T
    beta, log_alpha = lstsq(A, np.log(y), rcond=None)[0]
    alpha = np.exp(log_alpha)
    print(f'alpha={alpha}, beta={beta}')

    # Let's have a look of the data
    plt.figure(figsize=(10, 8))
    plt.plot(x, y, 'b.')
    plt.plot(x, alpha * np.exp(beta * x), 'r')
    plt.xlabel('t')
    plt.ylabel('u')
    plt.show()


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

5 ВЫЧИСЛЕНИЕ ОПРЕДЕЛЁННЫХ ИНТЕГРАЛОВ

Численное интегрирование (историческое название: (численная) квадратура) — вычисление значения определённого интеграла (как правило, приближённое). Под численным интегрированием понимают набор численных методов для нахождения значения определённого интеграла.

Численное интегрирование применяется, когда:

1. Сама подынтегральная функция не задана аналитически. Например, она представлена в виде таблицы (массива) значений в узлах некоторой расчётной сетки.
2. Аналитическое представление подынтегральной функции известно, но её первообразная не выражается через аналитические функции. Например, $f(x) = \exp(-x^2)$.

В этих двух случаях невозможно вычисление интеграла по формуле Ньютона — Лейбница. Также возможна ситуация, когда вид первообразной настолько сложен, что быстрее вычислить значение интеграла численным методом.

Основная идея большинства методов численного интегрирования состоит в замене подынтегральной функции на более простую, интеграл от которой легко вычисляется аналитически. При этом для оценки значения интеграла получаются формулы вида

$$I \approx \sum_{i=1}^{n} w_i \, f(x_i),$$

где n — число точек, в которых вычисляется значение подынтегральной функции. Точки x_i называются узлами метода, числа w_i — весами узлов. При замене подынтегральной функции на полином нулевой, первой и второй степени получаются соответственно методы , и (Симпсона). Часто формулы для оценки значения интеграла называют квадратурными формулами.

Частным случаем является метод построения интегральных квадратурных формул для равномерных сеток, известный как **формулы Котеса**. Метод назван в честь . Основной идеей метода является замена подынтегральной функции каким-либо . После взятия интеграла можно написать

$$\int_a^b f(x)\,dx = \sum_{i=0}^{n} H_i \, f(x_i) + r_n(f),$$

где числа H_i называются *коэффициентами Котеса* и вычисляются как интегралы от соответствующих многочленов, стоящих в исходном интерполяционном многочлене для подынтегральной функции при значении функции в узле $x_i = a + ih$ ($h = (b-a)/n$ — шаг сетки; n — число узлов сетки, а индекс узлов $i = 0 \ldots n$). Слагаемое $r_n(f)$ — погрешность метода, которая может быть найдена разными способами. Для нечетных $n \geqslant 1$ погрешность может быть найдена интегрированием погрешности интерполяционного полинома подынтегральной функции.

Частными случаями формул Котеса являются: формулы прямоугольников ($n = 0$), формулы трапеций ($n = 1$), формула Симпсона ($n = 2$), формула Ньютона ($n = 3$) и т. д.

5.1 Метод трапеций

Метод трапеций — метод численного интегрирования функции одной переменной, заключающийся в замене на каждом элементарном отрезке подынтегральной функции на многочлен первой степени, то есть линейную функцию. Площадь под графиком функции аппроксимируется прямоугольными трапециями. Алгебраический порядок точности равен 1.

Если отрезок $[a, b]$ является элементарным и не подвергается дальнейшему разбиению, значение интеграла можно найти по формуле

$$\int_a^b f(x)\,dx = \tfrac{f(a)+f(b)}{2}(b-a) + E(f), \qquad E(f) = -\tfrac{f''(\xi)}{12}(b-a)^3.$$

Это простое применение формулы для площади трапеции — произведение полусуммы оснований, которыми в данном случае являются значения функции в крайних точках отрезка, на высоту (длину отрезка интегрирования). Погрешность аппроксимации для элементарного отрезка можно оценить через максимум второй производной

$$|E(f)| \leqslant \tfrac{(b-a)^3}{12} \max_{x\in[a,b]} \left|f''(x)\right|$$

5.1.1 Применение составной формулы трапеций

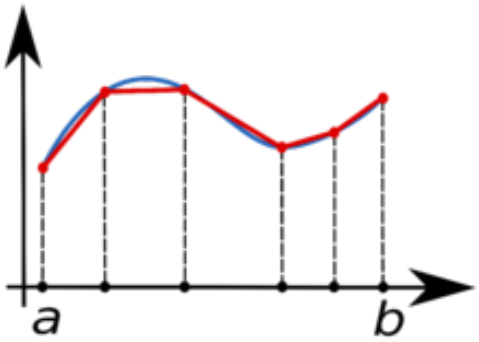

Рис. 3: Применение составной формулы трапеций

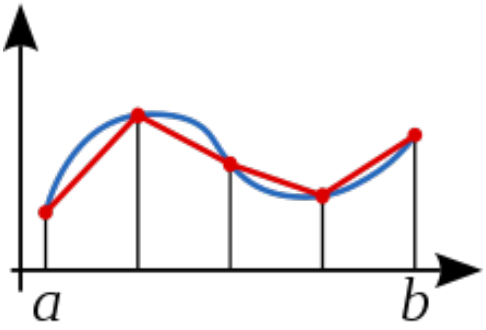

Рис. 4: Применение формулы трапеций для равномерной сетки

Если отрезок $[a, b]$ разбивается узлами интегрирования x_i, $i = 0, 1, \ldots, n$, так что $x_0 = a$ и $x_n = b$, и на каждом из элементарных отрезков $[x_i, x_{i+1}]$ применяется формула трапеций, то суммирование даст *составную формулу трапеций*

$$\int_a^b f(x)\,dx \approx \sum_{i=0}^{n-1} \tfrac{f(x_i)+f(x_{i+1})}{2}(x_{i+1}-x_i) =$$
$$= \tfrac{f(a)}{2}(x_1-a) + \tfrac{1}{2}\sum_{i=1}^{n-1} f(x_i)(x_{i+1}-x_{i-1}) + \tfrac{f(b)}{2}(b-x_{n-1}).$$

5.1.2 Формула Котеса

Применение формулы трапеций для равномерной сетки

В случае равномерной сетки $x_j = a + jh$, где $h = (b-a)/n$ — шаг сетки, составная формула трапеций упрощается:

$$\int_a^b f(x)\,dx = h\left(\tfrac{f_0+f_n}{2} + \sum_{i=1}^{n-1} f_i\right) + E_n(f),$$

причём для погрешности справедлива оценка

$$E_n(f) = -\tfrac{f''(\xi)}{12}(b-a)h^2.$$

5.1.3 Свойства

- Метод трапеций быстро сходится для осциллирующих функций, поскольку погрешность за период аннулируется.
- Метод может быть получен путём вычисления среднего арифметического между результатами применения формул правых и левых прямоугольников.

5.1.4 Реализация на языке Python

Листинг 8: Метод трапеций на языке Python

```
import math, random
from numpy import arange

def get_i():
    return math.e ** 1 - math.e ** 0

def trapezium_method(func, min_lim, max_lim, delta):
    def integrate(func, min_lim, max_lim, n):
        integral = 0.0
        step = (max_lim - min_lim) / n
        for x in arange(min_lim, max_lim-step, step):
            integral += step*(func(x) + func(x + step)) / 2
        return integral

    d, n = 1, 1
    while abs(d) > delta:
        d = (integrate(func, min_lim, max_lim, n * 2) - integrate(
            func, min_lim, max_lim, n)) / 3
        n *= 2

    a = abs(integrate(func, min_lim, max_lim, n))
    b = abs(integrate(func, min_lim, max_lim, n)) + d
    if a > b:
        a, b = b, a
    print('Trapezium:')
    print('\t%s\t%s\t%s' % (n, a, b))

trapezium_method(lambda x: math.e ** x, 0.0, 1.0, 0.001)
print('True value:\n\t%s' % get_i())
```

5.2 Метод Симпсона

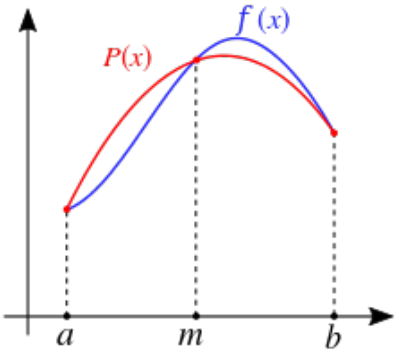

Рис. 5: *Суть метода — аппроксимация функции $f(x)$ квадратичным полиномом $P(x)$*

Формула Симпсона (также Ньютона-Симпсона) относится к приёмам численного интегрирования. Получила название в честь британского математика Томаса Симпсона (1710—1761).

Суть метода заключается в приближении подынтегральной функции на отрезке $[a, b]$ интерполяционным многочленом второй степени $p_2(x)$, то есть приближение графика функции на отрезке параболой. Метод Симпсона имеет порядок погрешности 4 и алгебраический порядок точности 3.

5.2.1 Формула

Формулой Симпсона называется интеграл от интерполяционного многочлена второй степени на отрезке $[a, b]$:

$$\int_a^b f(x)dx \approx \int_a^b p_2(x)dx = \tfrac{b-a}{6}\left(f(a) + 4f\left(\tfrac{a+b}{2}\right) + f(b)\right),$$

где $f(a)$, $f((a+b)/2)$ и $f(b)$ — значения функции в соответствующих точках (на концах отрезка и в его середине).

5.2.2 Погрешность

При условии, что у функции $f(x)$ на отрезке $[a, b]$ существует четвёртая , погрешность $E(f)$, согласно найденной формуле, равна:

$$E(f) = -\tfrac{(b-a)^5}{2880} f^{(4)}(\zeta), \quad \zeta \in [a, b].$$

В связи с тем, что значение ζ зачастую неизвестно, для оценки погрешности используется следующее неравенство:

$$|E(f)| \leqslant \tfrac{(b-a)^5}{2880} \max_{x\in[a,b]} \left|f^{(4)}(x)\right|.$$

5.2.3 Кубаторная формула Симпсона

Вычисление двойных интегралов также можно свести к формуле Симпсона:

$$I \approx \frac{h_x h_y}{9} \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} [f_{2i,2j} + 4f_{2i+1,2j} + f_{2i+2,2j} + 4f_{2i,2j+1} + 16f_{2i+1,2j+1} + \\ +4f_{2i+2,2j+1} + f_{2i,2j+2} + 4f_{2i+1,2j+2} + f_{2i+2,2j+2}],$$

где $\mathrm{h_X} = (\mathrm{b} - \mathrm{a})/2$ N, $\mathrm{hy} = (\mathrm{b} - \mathrm{a})/2$M.

Если S - криволинейная область интегрирования, то для применения формулы Симпсона область S заключают в прямоугольник $Sn(a \leq x \leq b, c \leq y \leq d\ )$ и пользуются вспомогательной функцией

$$g(x, y) = \begin{pmatrix} f(x, y), & \mathrm{emn(x, y)} \in \S, \\ \mathrm{emn} & (\mathrm{x, y}) \notin \S \cdot \S \end{pmatrix}.$$

Тогда

$$\iint_{(S)} f(x, y)dxdy = \iint_{(S_n)} g(x, y)dxdy$$

и для вычисления последнего интеграла привлекают метод Симпсона.

5.2.4 Реализация на языке Python

Листинг 9: Метод Симпсона на языке Python

```
import math, random
from numpy import arange

def get_i():
    return math.e ** 1 - math.e ** 0

def simpson_method(func, min_lim, max_lim, delta):
    def integrate(func, min_lim, max_lim, n):
        integral = 0.0
        step = (max_lim - min_lim) / n
        for x in arange(min_lim + step / 2, max_lim - step / 2,
            step):
            integral += step / 6 * (func(x - step / 2) + 4 * func(
                x) + func(x + step / 2))
        return integral

    d, n = 1, 1
    while abs(d) > delta:
        d = (integrate(func, min_lim, max_lim, n * 2) - integrate(
            func, min_lim, max_lim, n)) / 15
        n *= 2

    a = abs(integrate(func, min_lim, max_lim, n))
    b = abs(integrate(func, min_lim, max_lim, n)) + d
    if a > b:
        a, b = b, a
    print('Simpson:')
    print('\t%s\t%s\t%s' % (n, a, b))

simpson_method(lambda x: math.e ** x, 0.0, 1.0, 0.001)
print('True value:\n\t%s' % get_i())
```

5.3 Метод Монте-Карло

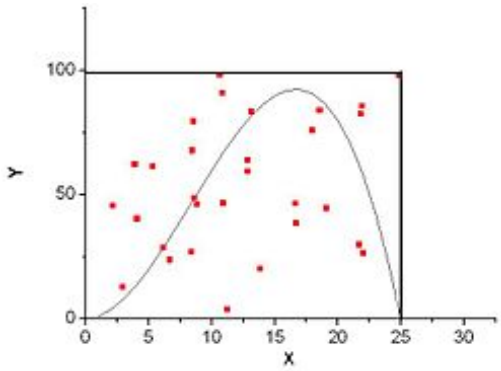

Рис. 6: *Численное интегрирование функции методом Монте-Карло*

Предположим, необходимо взять интеграл от некоторой функции. Воспользуемся неформальным геометрическим описанием интеграла и будем понимать его как площадь под графиком этой функции.

Для определения этой площади можно воспользоваться одним из обычных численных методов интегрирования: разбить отрезок на подотрезки, подсчитать площадь под графиком функции на каждом из них и сложить. Предположим, что для функции, представленной на рисунке 6, достаточно разбиения на 25 отрезков и, следовательно, вычисления 25 значений функции. Представим теперь, мы имеем дело с n-мерной функцией. Тогда нам необходимо 25^n отрезков и столько же вычислений значения функции. При размерности функции больше 10 задача становится огромной. Поскольку пространства большой размерности встречаются, в частности, в задачах теории струн, а также многих других физических задачах, где имеются системы со многими степенями свободы, необходимо иметь метод решения, вычислительная сложность которого бы не столь сильно зависела от размерности. Именно таким свойством обладает метод Монте-Карло.

5.3.1 Алгоритм

Для определения площади под графиком функции можно использовать следующий стохастический алгоритм:

- ограничим функцию прямоугольником (n-мерным параллелепипедом в случае многих измерений), площадь которого S_{par} можно легко вычислить; *любая сторона прямоугольника содержит хотя бы 1 точку графика функции, но не пересекает его;*
- «набросаем» в этот прямоугольник (параллелепипед) некоторое количество точек (N штук), координаты которых будем выбирать случайным образом;
- определим число точек (K штук), которые попадут под график функции;
- площадь области, ограниченной функцией и осями координат, S даётся выражением $S = S_{par}\frac{K}{N}$

Для малого числа измерений интегрируемой функции производительность Монте-Карло интегрирования гораздо ниже, чем производительность детерминированных методов. Тем не менее, в некоторых случаях, когда функция задана неявно, а необходимо определить область, заданную в виде сложных неравенств, стохастический метод может оказаться более предпочтительным.

5.3.2 Реализация на языке Python

Листинг 10: Метод Монте-Карло на языке Python

```
import math, random
from numpy import arange

def get_i():
    return math.e ** 1 - math.e ** 0
```

```
def monte_karlo_method(func, n):
    in_d, out_d = 0., 0.
    for i in arange(n):
        x, y = random.uniform(0, 1), random.uniform(0, 3)
        if y < func(x): in_d += 1

    print('M-K:')
    print('\t%s\t%s' % (n, abs(in_d/n * 3)))

monte_karlo_method(lambda x: math.e ** x, 100)
print('True value:\n\t%s' % get_i())
```

5.4 Увеличение точности

Приближение функции одним полиномом на всем отрезке интегрирования, как правило, приводит к большой ошибке в оценке значения интеграла.

Для уменьшения погрешности отрезок интегрирования разбивают на части и применяют численный метод для оценки интеграла на каждой из них.

При стремлении количества разбиений к бесконечности оценка интеграла стремится к его истинному значению для аналитических функций для любого численного метода.

Приведённые выше методы допускают простую процедуру уменьшения шага в два раза, при этом на каждом шаге требуется вычислять значения функции только во вновь добавленных узлах. Для оценки погрешности вычислений используется правило Рунге.

5.5 Многомерный случай

В небольших размерностях можно так же применять квадратурные формулы, основанные на интерполяционных многочленах. Интегрирование производится аналогично одномерному интегрированию. Для больших размерностей эти методы становятся неприемлемыми из-за быстрого возрастания числа точек сетки и/или сложной границы области. В этом случае применяется метод Монте-Карло. Генерируются случайные точки в нашей области и усредняются значения функции в них. Так же можно использовать смешанный подход — разбить область на несколько частей, в каждой из которых (или только в тех, где интеграл посчитать не удаётся из-за сложной границы) применить метод Монте-Карло.

Список использованной литературы

1. Мулярчик С. Г. Численные методы: Конспект лекций. Мн., БГУ, 2001. 127 с.;
2. И. М. Виноградов. Гаусса метод // Математическая энциклопедия. — М.: Советская энциклопедия. — 1977—1985.
3. Ильин В. А., Позняк Э. Г. Линейная алгебра: Учебник для вузов. — 6-е изд., стер. — М.: ФИЗМАТЛИТ, 2004. — 280 с.
4. Волков Е. А. Численные методы. — М.: Физматлит, 2003.
5. Тихонов А. Н., Самарский А. А. Уравнения математической физики. — гл. III
6. Сальвадори М. Дж. Численные методы в технике. - М., Вузовская книга, 2007.