Библиотека численных методов NumLibABC
для PascalABC.Net 3.5

Справочное пособие

Ростов-на-Дону
2019

Описывается состав и даются рекомендации по использованию библиотеки численных методов NumLibABC, реализованной в среде программирования PascalABC.NET 3.5.

Для школьников старших классов, учащихся колледжей и студентов младших курсов вузов.

Оглавление

1. Введение

NumLibABC – свободно распространяемая библиотека (далее также используется термин «пакет») численных методов, реализованная в системе программирования PascalABC.NET 3.5 и поставляющаяся вместе с ней, в том числе, в исходном коде (файл NumLibABC.pas).

В пакете находятся программы, реализующие различные численные методы посредством классов, а также вспомогательные программы и сопутствующие типы данных.

С помощью пакета NumLibABC можно решать задачи из следующих областей:

— нахождение корней нелинейных уравнений;
— операции с простыми дробями;
— операции с полиномами;
— интерполяция, дифференцирование и аппроксимация данных, заданных в табличном виде;
— операции с векторами и матрицами, решение систем линейных уравнений;
— решение систем дифференциальных уравнений;
— вычисление определенных интегралов;
— задачи оптимизации.

Часть программ переведена в паскаль на уровне исходного текста из существующих пакетов прикладных программ, таких как SSPLIB (на языке Фортран) или опубликованных в литературе. В

этом случае подробная ссылка на источник приведена в тексте программы. Другая часть написана автором на основе алгоритмов, приведенных в различных источниках и в этом случае ссылка на источник также дается в тексте программы.

1.1. Лицензия

2. Система тестирования NumLibABCTest

После установки или обновления PascalABC.NET рекомендуется выполнить тестирование пакета при помощи модуля NumLibABCTest.pas, который в исходном виде находится в поддиректории \PABCWork.NET\Samples\NumLibABC. Там же находятся файлы с примерами программ, приведенных в данном пособии.

Тестирование заключается в решении ряда контрольных заданий и сличении полученных результатов с эталонами. Проведение тестирования является хорошим подтверждением работоспособности установленной версии.

Каждый модуль пакета тестируется на наборе тестовых примеров и при непрохождении теста с помощью Assert выдается сообщение с указанием полученных и ожидаемых результатов, позволяющее локализовать место ошибки. Несмотря на то, что весь пакет тщательно тестируется, допускается возможность непрохождения тестов в программах, использующих случайные числа. В таких случаях полезно попытаться выполнить тестирование несколько раз, чтобы убедиться в наличии четкой ошибки.

В ходе тестирования по мере прохождения тестов программных единиц на монитор выводится протокол.

Набор тестовых заданий содержит достаточное количество примеров, ознакомление с которыми может оказаться полезным для лучшего понимания работы с пакетом.

3. Общая информация

3.1. Точность представления чисел типа real

В PascalABC.NET тип real базируется на представлении данных System.Double платформы Microsoft .NET. Значение типа real занимает 8 байт при длине мантиссы 52 разряда, что обеспечивает точность не более 17 десятичных знаков.

PascalABC.NET предоставляет несколько констант платформы .NET, связанных с типом real:

— real.MinValue - минимальное значение, примерно ное -1.7976931348623157E+308;

— real.MaxValue - максимальное значение, примерно равное 1.7976931348623157E+308;

— real.Epsilon - минимальное положительное число, отличное от нуля («машинная точность»), которое выводится с несколько странным значением 4.94065645841247E-324 (к этому мы еще вернемся);

— NaN – «Not a Number» («не число»), возникает при делении 0/0, вычислении функций с недопустимыми аргументами и т.п. Возникнув, имеет тенденцию распространяться на всю правую часть оператора присваивания, в связи с чем при программировании рекомендуется принимать меры к изоляции NaN при помощи проверки IsNaN(x), возвращающей true для x=NaN;

— real.NegativeInfinity – «отрицательная бесконечность», возникающая при делении отрицательной величины на ноль. Проверяется при помощи IsNegativeInfinity(x);
— real.PositiveInfinity - «положительная бесконечность», возникающая при делении положительной величины на ноль. Проверяется при помощи IsPositiveInfinity(x).

Когда знак бесконечности не имеет значения, можно воспользоваться проверкой IsInfinity(x).

Вернемся к рассмотрению машинной точности. Практическое использование константы real.Epsilon приводит к «удивительным» (в нехорошем смысле этого слова) результатам, чем и объясняется решение отказаться использования этой константы при написании пакета NumLibABC.

В [1] предлагается считать точностью (машинным эпсилон) такую минимальную величину ε, для которой 1 + ε > 1

При попытке воспользоваться real.Epsilon были получены совершенно неудовлетворительные результаты, в частности,

*1.0 + real.Epsilon = 1.0 + real.Epsilon*1e100*

Понятно, что это делает невозможными сравнения точности с величинами ε, 2ε, … 1000ε … , поэтому в программах машинная точность определяется следующим образом:

```
var eps:=1.0;
while eps+1.0>1.0 do eps*=0.5;
```

В этом случае найденная машинная точность оказалась равной 1.11022302462516e-16 ($7FFFFFFFFFFFFFFF_{16}$).

3.2. Работа с пакетом

Для подключения пакета следует использовать секцию раздела uses:

uses *NumLibABC;*

Далее делаются необходимые определения функций, переменных массивов и т.п, а затем создается объект нужного класса по образцу

var *имя_переменной :=* ***new*** *имя_класса(параметры);*

В некоторых случаях этого уже достаточно чтобы воспользоваться результатами, в остальных - вызывается какой-либо метод класса, возвращающий результаты.

Исходный текст библиотеки находится в файле
\Program Files\PascalABC.NET\LibSource\NumLibABC.pas

Данный документ стандартно находится в файле
\Program Files\PascalABC.NET\Doc\NumLibABC.pdf

4. Описание программ

4.1. Нахождение корней нелинейных уравнений

Пусть задана некоторая функция y=F(x). Требуется отыскать одно или более значений x, для которых F(x)=0. В этом случае все такие x будут называться корнями уравнения F(x)=0.

Теория утверждает, что если $x \in [a;b]$ и функция F(x) на интервале [a;b] меняет знак ровно один раз, то внутри этого интервала найдется отрезок $[\alpha;\beta]$ длины, не превышающей некоторого значения ε, на котором F(x) также меняет знак и с точностью ε этот интервал можно считать корнем уравнения F(x)=0. Отрезок [a;b] называется интервалом изоляции корня и далее предполагается, что F(a) и F(b) имеют разные знаки, либо F(a)=0, $F(b)\neq 0$, либо $F(a)\neq 0$, F(b)=0.

Почему вместо точного значения корня уравнения мы говорим о некотором интервале $[\alpha;\beta]$ с длиной, не превышающей заданную точность ε? Все дело в дискретности (и вытекающей из этого точности) представления чисел типа real. Это интервал далее называется интервалом неопределенности.

Решение задачи на практике состоит из нескольких шагов. На первом шаге определяются интервалы изоляции корней уравнения, а на последующих для каждого интервала изоляции с заданной точностью вычисляется очередной корень.

Одним из самых простых и надежных приемов отделения корня является табличный метод. Для совокупности равноотстоящих точек на оси x вычисляются значения y(x) до тех пор, пока не будет выявлен интервал изоляции.

Другой способ отделения корней основан на методе Монте-Карло. На некотором «разумном» интервале случайным образом заданных значений x вычисляются значения функции y(x) и запоминаются те точки x_0, в которых значение $y(x_0)$ наиболее близко к нулю. Затем делается шаг, например, в сторону уменьшения x, т.е. полагаем $x_1=x_0-h$, и если значение функции $y(x_1)$ также уменьшается, делается следующий шаг в этом же направлении, пока функция не изменит знак. Если при первоначальном шаге функция увеличила значение, то делаем шаг удвоенной длины в обратном направлении, приходя в точку $x_1=x_0+h$ и ведем поиск в новом направлении. Это способ, несмотря на большую, чем предыдущий сложность, в некоторых случаях может быстрее приводить к нужному результату.

4.1.1. Изоляция корней уравнения y(x)=0 на заданном интервале табличным методом (RootsIsolation)

Может применяться для получения интервалов изоляции произвольного количества корней произвольной функции одной переменной. Заданный интервал просмотра [a;b] последовательно просматривается с шагом h. Если корень уравнения попадает на

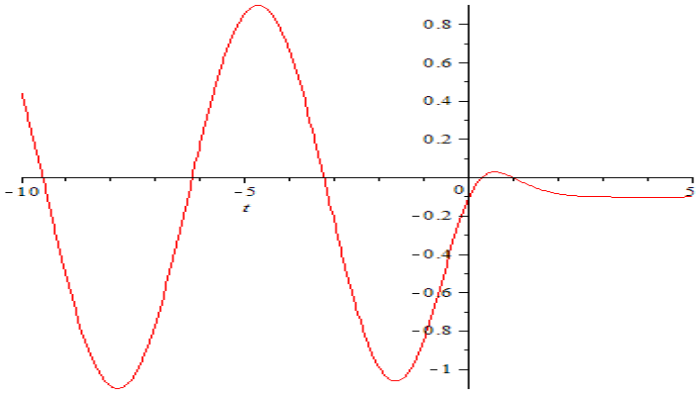
границу интервала b, интервал изоляции выбирается равным [b-h/2;b+h/2]. Правильный выбор шага h имеет важное значение.

Рассмотрим пример нахождения интервалов изоляции уравнения, график которого представлен далее.

$$\frac{\sin(t)}{1+\left(e^{t}\right)^{2}}-1=0$$

Конечно, если имеется график, интервалы изоляции можно оценить по нему. Но в данном случае график представлен чтобы показать, как выбор слишком большого шага h приводит к ошибкам в решении.

Как видно из графика, при отрицательных значениях аргумента корни отстоят друг от друга примерно на 3, а при положительных - меньше чем на 1. Следовательно, разумно будет выбрать шаг h<1, например, 0.5.



Пусть интервал поиска корней составит [-10;5].

```
var f:real->real:=t->sin(t)/(1+Sqr(Exp(t)))-0.1;
var (a,b,h):=(-10,5,0.5);
var oS:=new RootsIsolation(f,a,b,h); // создали объект
Writeln(oS.Value); // использовали его метод Value
```

Получаем решение:

[(-10,-9.5),(-6.5,-6),(-3.5,-3),(0,0.5),(1,1.5)]

Это – правильный результат, потому что корни уравнения приблизительно равны -9.52495, -6.18307, -3.24191, 0.27789, 1.00272 (см.4.1.2).

Попробуем задать шаг, который будет в данных условиях неприемлемым, например h=2.

Получаем решение:

[(-10,-8),(-8,-6),(-4,-2)] – потеряли два интервала изоляции.

Наконец, совсем большой шаг h=6 приводит к еще более катастрофическим результатам: [(-4,2)] – мы теряем четыре из пяти интервалов изоляции.

4.1.2. Нахождение действительного нуля функции на интервале изоляции (Zeroin)

Zeroin – один из лучших имеющихся машинных алгоритмов, сочетающих безотказность бисекции с асимптотической скоростью метода секущих для случая гладких функций [1]. В пакет включен класс Zeroin, портированный с фортрана в систему программирования PascalABC.NET.

Предполагается без проверки, что интервал изоляции корня [a;b] определен, в противном случае пользоваться Zeroin некорректно. Необходимо также задать величину максимально допустимого интервала неопределенности решения tol, т.е длину отрезка, содержащего корень уравнения, что можно рассматривать как некий аналог точности решения.

Найдем корни на интервалах, определенных в 4.1.1. Полная программа может выглядеть так:

```
uses NumLibABC;
begin
  var f:real->real := t->sin(t)/(1+Sqr(Exp(t)))-0.1;
  var oS := new Zeroin(f,1e-12); // "точность" 10^(-12)
  Println(oS.Value(-10,-9.5), oS.Value(-6.5,-6), os.Value(-3.5,-3),
    os.Value(0,0.5),os.Value(1,1.5))
end
```

Результаты

```
-9.52494538246664   -6.18301745778349   -3.24191364084812
0.277894592306507   1.00272135335711
```

Понятно, что доверять можно только 11-12 знакам после запятой в соответствии с запрошенным интервалом неопределенности.

4.2. Обыкновенные дроби

Выше (3.1) уже частично упоминались проблемы, связанные с невозможностью точного представления нецелых чисел в машинной арифметике. Следующая часть проблем состоит в том, что большинство отношений двух целых чисел нельзя точно представить десятичной дробью, например, 1/3. Обо всем этом понятно и подробно сказано, например, в [1] (глава 2).

Часть проблем удается устранить, используя обыкновенные дроби взамен десятичных. Беда лишь в том, что обыкновенных дробей нет ни в машинной арифметике, ни в распространенных универсальных языках программирования. Частично компенсировать их отсутствие может включенный в состав пакета класс Fraction.

4.2.1. Описание класса Fraction

Структурно дробь состоит из пары целых чисел, представляющих числитель (numerator) и знаменатель (denominator). Поскольку целые числа в машинной арифметике имеют достаточно ограниченную разрядность, при разработке класса было принято решение использовать класс BigInteger платформы .NET, реализующий практически неограниченную разрядность целых чисел.

Дробь создается конструктором класса, принимающим в качестве параметров значение числителя и знаменателя, например:

```
var a:=new Fraction(24243,745347);
```

Поскольку параметрами при вызове конструктора должны быть значения типа BigInteger, а константы такого типа в PascalABC.Net не предусмотрены, для чисел, в записи которых используется более 19 цифр приходится использовать строковое представление с последующим преобразованием

var b:=new Fraction(1,'12345678901234567890123'.ToBigInteger)

Некоторое дополнительное удобство при записи обыкновенных дробей - констант может предоставить функция Frc, которая имеет три модификации.

function Frc(a:BigInteger):fraction; - дробь вида a/1
function Frc(a,b:BigInteger):fraction; - дробь вида a/b
function Frc(a,b,c:BigInteger):fraction; - дробь вида a+b/c

Фактически это «обёртки» для вызова конструктора класса, позволяющее более комфортно записывать выражения, содержащие несколько обыкновенных дробей.

Например, выражение вида

$$\frac{34}{197}+6\frac{9}{91}-\frac{351}{95113}\times\left(\frac{-1}{7}\right)$$

запишется *Frc(34,197)+Frc(6,9,91)-Frc(351,95113)*Frc(-1,7);*

Если возник вопрос по поводу операций с дробями – все просто: они для класса Fraction перегружены.

Объект класса Fraction имеет два свойства – numerator и denominator, представляющие собой значения числителя и знаменателя. Дробь по возможности сокращена, т.е. числитель и знаменатель всегда разделены на их наибольший общий делитель (НОД).

Арифметические операции +, -, *, /, а также операторы +=, -=, *= и /= перегружены и могут использоваться также в случаях, когда один из операндов имеет тип, приводящийся к BigInteger. Это позволяет записывать операции по типу Frc(8/37)*12; что еще больше упрощает программирование выражений.

Также перегружены операции отношения =, <>, >, >=, <, ,<=.

Кроме Frc, класс предоставляет следующие методы:

- Abs – абсолютная величина дроби;
- Inv – обратная величина дроби;
- Print – вывод значения дроби в виде a/b;
- ToReal – преобразование дроби к типу real;
- ToString – преобразование дроби к строке вида a/b.

Пример программирования выражения

$$\left[\left(5\frac{5}{9}-\frac{7}{18}\right):35+\left(\frac{40}{63}-\frac{8}{21}\right):20+\left(\frac{83}{90}-\frac{41}{50}\right):2\right]\times 50$$

var r:=((Frc(5,5,9)-Frc(7,18))/35+(Frc(40,63)-Frc(8,21))/20+
*(Frc(83,90)-Frc(41,50))/2)*50;*

Класс может быть использован для достижения абсолютной точности в любых алгоритмах, базирующихся на действиях арифметики, порождающих только рациональные числа. Например, с его помощью можно точно найти обратную матрицу в условиях, когда исходная матрица почти вырождена. Достаточно лишь взять процедуру обращения матрицы и описать в ней переменные класса Fraction.

4.3. Полиномы с действительными коэффициентами

4.3.1. Описание класса Polynom

Обычно полином u(x) принято записывать в порядке убывания степеней, т.е. в виде $P(x) = a_n x^n + \ldots + a_1 x + a_0$

В то же время, большинство алгоритмов для работы с полиномами используют обратный порядок записи – в порядке возрастания степеней: $a(x) = a_0 + a_1 x + \ldots + a_n x^n$

Именно такой порядок следования коэффициентов полинома используется, например, в пакетах программ SSPLIB фирмы IBM [2] и IMSL® фирмы Rogue Software [3]. Исходя из изложенного выше, было принято использовать такой же порядок следования и в пакете *NumLibABC*.

Полиномы реализованы в виде класса Polynom, имеющего два свойства:

a – динамический массив типа real с коэффициентами полинома, расположенными в порядке **возрастания** степени;

n – количество членов полинома;

eps – оценка точности при экономизации полинома.

Для создания полинома используются две формы вызова конструктора класса.

Вызов вида new Polynom(k) создает полином с k членами (т.е. степени k-1) и все коэффициенты полинома устанавливает в значение 0.0.

Вызов new Polinom($a_0, a_1, \ldots a_m$) создает полином с указанными значениями коэффициентов. Вместо списка коэффициентов можно указать динамический массив типа real (см. примеры в 4.3.2).

Перечислим методы класса *Polynom*:

Value(x) – возвращает значение полинома для аргумента x;

Copy – создает копию объекта класса *Polynom*;

EconomSym(h,limit:real) – проводит экономизацию полинома (4.4.3) на симметричном интервале [-h;h] с ошибкой, не превышающей limit;

EconomUnsym(h,limit:real) – проводит экономизацию полинома (4.4.3) на несимметричном интервале [0;h] с ошибкой, не превышающей limit;

PDif – возвращает коэффициенты полинома, полученного дифференцированием исходного;

PInt – возвращает коэффициенты полинома, полученного при взятии неопределенного интеграла от исходного;

Polyx – возвращает полином при замене аргумента x на at+b (на основе алгоритма 29б [4]) ;

Print, Println – выводят на монитор коэффициенты полинома в строку через пробел. Println затем обеспечивает перевод вывода на новую строку;

Print(c), Println(c) вместо пробела используют разделитель c;

PrintlnBeauty – выводит полином в более привычном виде.

Примеры работы с классом *Polynom*

1. $p(x) = 4x^3 + 6.5x^2 - 18$

```
var p:=new Polynom(-18, 0, 6.5, 4);
```

2. Вычислить для x=-7.16 значение $u(x) = x^5 + 3.8x^2 - 6x + 2$

```
var u:=(new Polynom(2, -6, 0, 3.8, 0, 1)).Value(-7.16);
```

3. Поместить в p(x) полином q(x)

```
var p:=q.Copy;
```

4. Для $t(x) = -2x^4 - 3x^3 + 12x^2 - 7x + 1$ получить

$$p(x) = \int t(x)\,dx, \qquad q(x) = t'(x)$$

```
var t:=new Polynom(1, -7, 12, -3, -2);
var (p,q):=(t.PInt, t.PDif);
p.PrintlnBeauty; q.PrintlnBeauty;
```

Результаты

```
-0.4x^5-0.75x^4+4x^3-3.5x^2+x
-8x^3-9x^2+24x-7
```

Запишем результаты в стандартной форме:

$$p(x) = -0.4x^5 - 0.75x^4 + 4x^3 - 3.5x^2 + x + C,$$
$$q(x) = -8x^3 - 9x^2 + 24x - 7$$

5. Для $P(x) = 5x^4 - 4x^2 - 3x + 2$ сделать замену x=2t-3

```
var p:=new Polynom(2,-3,-4,0,5);
p.Polyx(2,-3).PrintlnBeauty('t');
```

Результаты

```
80t^4-480t^3+1064t^2-1038t+380
```

Безусловно, коэффициенты могут быть и нецелыми числами.

4.3.2. Полиномиальная арифметика

К полиномиальной арифметике отнесены операции сложения, вычитания, умножения и деления полиномов.

Для сложения, вычитания и умножения полиномов, а также для случая, когда один из операндов является константой, соответствующие арифметические операции для класса *Polynom* перегружены, что позволяет записывать арифметические выражения в привычном виде. Перегружены также унарный минус и операция «=».

Пример:

```
var a:=new Polynom(6.5,-4,2.12,1);
var b:=new Polynom(3,0,-3.8);
var c:=new Polynom(ArrGen(5,i->i*i+1.0));
(-c +(a-2*b)*a+11.5*(1-b)).Println;
```

Деление полиномов выполняется посредством перегруженной операции /, возвращающей кортеж из двух полиномов – частного и остатка. Вычислим следующее выражение:

$$\frac{2x^6 - x^5 + 12x^3 - 72x^2 + 3}{x^3 + 2x^2 - 1}$$

```
var p:=new Polynom(3,0,-72,12,0,-1,2);
var q:=new Polynom(-1,0,2,1);
var (a,b):=p/q;
a.PrintlnBeauty; b.PrintlnBeauty;
```

Результаты

`2x^3-5x^2+10x-6` – частное (полином a)
`-65x^2+10x-3` – остаток (полином b)

Решение выглядит следующим образом

$$\frac{2x^6 - x^5 + 12x^3 - 72x^2 + 3}{x^3 + 2x^2 - 1} =$$
$$2x^3 - 5x^2 + 10x - 6 + \frac{-65x^2 + 10x - 3}{x^3 + 2x^2 - 1}$$

Также разрешается делить скалярную величину на многочлен и многочлен делить на скалярную величину. В первом случае возвращается кортеж типа (real, Polynom), где первый элемент – это частное (собственно, исходная скалярная величина), а второй – остаток, т.е. фактически многочлен-делитель. Во втором случае возвращается многочлен

```
var p:=new Polynom(3,0,-72,12,0,-6,12);
var (a,b):=7/p;
Write(a,' '); b.PrintlnBeauty;
b:=p/3; b.PrintlnBeauty;
```

Результаты

```
7, 12x^6-6x^5+12x^3-72x^2+3
4x^6-2x^5+4x^3-24x^2+1
```

Возведение полинома в натуральную степень выполняется умножением:

```
var p:=new Polynom(-4.2,3.7,6,2.1);
(p*p*p).PrintlnBeauty;
```

Результаты

```
9.261x^9+79.38x^8+275.751x^7+440.154x^6+168.327x^5-
402.984x^4-397.655x^3+145.026x^2+195.804x-74.088
```

4.3.3. Корни полиномов. Операции с полиномами

Из основной теоремы алгебры следует, что если полином P(x) имеет степень n, то уравнение P(x)=0 имеет ровно n корней, среди которых могут быть как действительные корни, так и пары комплексно-сопряженных. Существует достаточно обширное количество алгоритмов нахождения корней полиномов. Одним из реализаций таких алгоритмов служит метод Ньютона-Рафсона.

4.3.3.1. Нахождение всех корней полинома с действительными коэффициентами методом Ньютона —Рафсона (PolRt)

Класс PolRt выполнен на основе подпрограммы DPOLRT [2], написанной на языке Fortran. Нельзя использовать его при степени полинома n>36 из-за ошибок округления, возникающих в машинной арифметике с плавающей точкой.

Метод Ньютона-Рафсона – это другое название итерационного метода, известного как «метод касательных».

Типовой вызов:

var *p :=* ***new*** *Polynom(коэффициенты по возрастанию степени)*

var *oL :=* ***new*** *PolRt(p);*

Далее следует проверить свойство oL.ier для оценки результатов решения:

0 – ошибок не найдено;

1 – недостаточно элементов для построения полинома;

2 – степень полинома превышает 36;

3 – не удалось достигнуть приемлемой точности за 500 шагов;

4 – коэффициент при старшей степени полинома нулевой.

При помощи метода Value получаем динамический массив типа complex, содержащий искомые корни:

```
var r:=oL.Value
```

Пример

$x^5 - 5x^4 - 38x^3 + 294x^2 - 283x - 609 = 0$

```
var p := new Polynom(-609, -283 ,294, -38, -5,1);
var oL := new PolRt(p);
if oL.ier=0 then oL.Value.Println
else Writeln('Ошибка: ier=',oL.ier);
```

Результаты

(-1,0) (3,0) (-7,0) (5,-2) (5,2) – найдены все пять корней уравнения: -1, 3, -7, 5-2i, 5+2i.

Класс PolRt можно также использовать для разложения полинома на множители, приравняв его нулю. В данном случае полином может быть представлен в виде

$(x + 1)(x - 3)(x + 7)(x - 5 - 2i)(x - 5 + 2i)$

Если комплексные числа в представлении недопустимы, достаточно попарно перемножить элементы, содержащие комплексно- сопряженные корни и не забывая, что i^2=-1:

$(x + 1)(x - 3)(x + 7)(x^2 - 10x + 29)$

Для разложения полинома на рациональные линейные множители, удобнее воспользоваться классом Factors.

4.3.3.2. Разложение полинома с целочисленными коэффициентами на рациональные линейные множители (Factors)

Класс Factors выполнен на основе процедуры factors [5], опубликованной на языке Algol-60. Он позволяет находить в разложении полинома множители вида px-q, где p,q – целые числа. Алгоритм базируется на схеме Горнера.

Пусть имеется некоторый полином с целочисленными коэффициентами $P(x) = a_0x^n + a_1x^{n-1} + \ldots + a_{n-1}x + a_n$

Известно, что если некоторое целое число p служит корнем полинома P(x), то p будет делителем свободного члена этого полинома. Алгоритм находит все делители p свободного члена a_n и все делители q коэффициента при старшей степени a_0.

Из каждой пары составляется моном (px-q) и проверяется, не является ли он множителем полинома P(x). Если проверка показывает положительный результат, выполняется деление P(x) на (px-q) и алгоритм применяется к полученному частному.

При создании класса Factors коэффициенты полинома должны задаваться в порядке возрастания степени. Метод Factorize для полинома степени n возвращает массив размера [m+1,2]. Его первая строка служебная и содержит в элементе [0,0] количество найденных линейных множителей m, а в элементе [0,1] - максимальный наибольший общий делитель коэффициентов полинома в разложе-

нии. Каждая последующая i-я строка содержит коэффициенты множителя: [i,0]=p, [i,1]=q

В случае, если свободный член полинома нулевой, предварительно следует вынести аргумент за скобку и производить поиск в оставшемся полиноме с ненулевым свободным членом. **В противном случае разложение не будет найдено.**

Приведем пример. Пусть $P(x) = -42x^3 + 73x^2 + 7x - 20$

Запишем фрагмент программы для разложения полинома.

```
var oL:=new Factors(-20, 7, 73, -42);
var r:=oL.Factorize;
Writeln('k:=',r[0,1]);
for var i:=1 to r[0,0] do r.Row(i).Println;
```

Результаты: k:=-1; 2 -1; 3 5; 7 4

Анализ показывает, что полином третьей степени разложился на три монома, т.е. полностью. Тогда можно записать разложение в виде $P(x) = -(2x + 1)(3x - 5)(7x - 4)$

Знак минус – это коэффициент k. А далее к каждому первому значению приписываем x, второе берем с обратным знаком.

Рассмотрим еще один пример.

$P(x) = 6x^4 - x^3 - 52x^2 - 12x + 45$

```
var oL:=new Factors(45, -12, -52, -1, 6);
var r:=oL.Factorize;
Writeln('k:=',r[0,1]);
for var i:=1 to r[0,0] do r.Row(i).Println;
```

Результаты: k:=1; 1 3; 2 -5

В разложении присутствуют только два монома, а не четыре, поэтому полином полностью не раскладывается, т.е. его остальные множители не являются целыми числами. Найдено лишь частичное разложение $(x-3)(2x+5)$

При желании получить полное разложение полинома можно воспользоваться полиномиальной арифметикой (4.4.2):

```
var a:=new Polynom(45,-12,-52,-1,6);
var r:=a/(new Polynom(-3,1)*(new Polynom(5,2)));
r[0].PrintlnBeauty; r[1].PrintlnBeauty
```

Результаты: 3x^2+x-3; 0

Окончательно $P(x)=(x-3)(2x+5)(3x^2+x-3)$

Можно попытаться использовать для разложения полинома метод PolRt из 4.3.3.1:

```
var oP:=new Polynom(45,-12,-52,-1,6);
var oL:=new PolRt(oP);
oL.Value.Println;
```

Результаты

```
(0.847127088383037,0) (-1.18046042171637,0) (-2.5,0) (3,0)
```

Все четыре корня действительные. Два первых из них скорее всего не являются рациональными, остальные два дают разложение $(x+2.5)(x-3)=0.5(2x+5)(x-3)$

Для получения полного разложения и тут придется воспользоваться полиномиальной арифметикой, так что выбор первичного численного метода остается за пользователем.

4.4. Одномерная интерполяция и аппроксимация данных, заданных в табличном виде

Пусть имеется некий набор значений x и y, где y=f(x), т.е. каждому x поставлено в соответствие некоторое y, причем связь между x, y нам либо неизвестна, либо известна, но очень сложна. Такую связь просто и удобно отображать в табличном виде.

Если требуется получить значения y=f(x) для значений x, отсутствующих в таблице, возникает задача замены зависимости y=f(x) некоторой приближенной с заданной степенью точности зависимостью $y = \phi(x)$ так, чтобы отклонение приближенных значений от истинных было минимальным на всем заданном интервале. Такая задача называется задачей аппроксимации.

Поскольку приближение выполняется на некотором наборе исходных точек, аппроксимация называется точечной.

Существует отдельная разновидность аппроксимации – интерполяция, которая требует, чтобы во всех исходных точках $x = x_i$ выполнялось условие $\phi(x_i) = f(x_i) = y_i$

В этом случае исходные точки носят название узлов интерполяции и можно приведенное выше условие сформулировать иначе: значение аппроксимирующей функции в узлах интерполяции должно совпадать с исходным.

Самое очевидное решение состоит в том, чтобы для набора из n узлов интерполяции построить полином степени n-1. И оно действительно имеется в арсенале численных методов.

$\phi(x) = P_m(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_m x^m, \ \ m = 1,2, \ldots n-1$

К сожалению, теория и практика доказывают, что такое решение оказывается очень плохим для n>10.

Интерполирующая функция может быть построена для всего набора узлов интерполяции, либо строиться отдельно для различных подмножеств этого набора. В первом случае интерполяция называется глобальной, во втором – кусочной (локальной).

Еще одна вытекающая из интерполяции задача – это экстраполяция, при которой значения аргумента выходят за пределы отрезка, заданного набором узлов интерполяции.

На практике встречается и другая задача – собственно аппроксимация, при решении которой выполнение условия $\phi(x_i) = f(x_i) = y_i$ не требуется.

Например, исходные значения могли быть получены в результате обработки данных некоторого эксперимента или при проведении каких-то измерений, т.е. по определению могли содержать некоторые погрешности. Решая задачу интерполяции, мы фактически потребуем повторять при вычислении эти же ошибки, что лишено практического смысла. Гораздо полезнее требовать, чтобы аппроксимирующая функция обеспечивала необходимую точность приближения не только в узлах, но и между ними.

Можно выбирать различные критерии точности приближения, но на практике набольшее распространение получил метод наименьших квадратов (МНК).

$$S = \sum_{i=1}^{n} [\phi(x_i) - y_i]^2 \to \min$$

Вместо квадрата разности можно было бы, например, взять модуль разности, но модуль – это кусочная функция, которая затрудняет математические выкладки.

Можно выбирать в качестве аппроксимирующих функции самого разного вида, а также их комбинации. На практике часто строят графическую зависимость с тем, чтобы по виду кривой выбрать вид функции. Существует аппроксимация полиномами различных степеней (начиная от первой, т.е. прямой линией), параболами, гиперболами, степенными, экспоненциальными и логарифмическими кривыми, рациональными функциями в форме отношением полиномов, тригонометрическими функциями и т.д.

Лагранж предложил выполнять аппроксимацию полиномами l_j, каждый из которых на участке [0;1] обращается в единицу для заданного узла и принимает значение ноль в остальных узлах.

$$L_{n-1}(x_i) = \sum_{i=1}^{n} l_i(x_i) y_i$$

Если наложить дополнительное требование равенства в узлах значения вычисленной первой производной значению первой производной исходной табличной функции, получим аппроксимацию полиномами Эрмита.

Существует также иной способ построения интерполяционного полинома, предложенный Ньютоном, который в результате дает тот же самый полином, что и способ Лагранжа.

Замена интервала определения полинома [a;b] на требуемый интервал [0;1] или [-1;1] производится путем несложного преобразования, например, для l ∈ [a;b] → [-1;1]

$$l(x) = \frac{2x - (b + a)}{b - a}$$

На практике пользоваться интерполяционными полиномами Лагранжа неудобно, поскольку вне узлов они дают очень плохое приближение функции.

П.Л.Чебышев показал, что наилучшей результат аппроксимации по МНК достигается, если ошибка равномерно распределена по всему интервалу определения функции. Он предложил использовать для аппроксимации набор полиномов специального вида

$$T_n(x) = \cos(n \arccos x)\,,\ n = 0,1,2 \ldots\,,\ x \in [-1; 1]$$

4.4.1. Интерполяция табличной функции кубическим сплайном (Spline)

Задача одномерной интерполяции набора n точек *P(x,y)* с вещественными координатами сводится к построению некоторой функции *F(x)*, для которой $F(x_i) = y_i$ для всех n точек и при этом в промежутках между точками функция принимает некие «разумные значения» [1].

Считается, что если функция *F(x)* гладкая и вычисленные по ней значения не превышают допустимой ошибки, задача имеет удовлетворительное решение.

Если набор точек *P(x,y)* не зашумлен ошибками, то интерполяция гладкой функцией уместна. В противном случае используются приёмы, нейтрализующие шум.

Математики обожают, когда функция может быть дважды дифференцируема и при этом не выродится в ноль или иную константу, поэтому минимальная степень интерполяционного полинома равна трем и он будет проходить через четыре исходные точки. Кубический полином – это самая гладкая функция, обладающая необходимыми для интерполяции свойствами. Но ведь точек обычно куда больше, чем четыре…

С другой стороны, с древних времен известен чертежный инструмент, называемый лекало. Он позволяет гладко соединить множество точек, нанесенных на плоскость. Суть используемого приема в том, чтобы соединять точки линией по две-три, постепенно пе-

ремещая лекало вдоль заданных точек. Английские чертежники называли сплайном гибкую металлическую линейку, которую они использовали для соединения точек на чертеже плавной кривой, фактически проводя интерполяцию. «Скользящие» кубические полиномы также получили название кубических сплайн-функций или просто сплайнов.

Поскольку сплайн – это кубический полином, т.е.

$y = F(x) = ax^3 + bx^2 + cx + d,$

$y' = F'(x) = 3ax^2 + 2bx + c, \qquad y''=F''(x) = 6ax + 2b,$

то можно легко найти первую и вторую производную от таблично заданной функции.

В пакете имеется класс *Spline*, позволяющий выполнить интерполяцию кубическим сплайном. Он является результатом переработки программ SPLINE и SEVAL [1], написанных на языке Fortran.

Вспомогательный класс *Point* реализует точку с координатами x,y типа real. Класс *Spline* в своем свойстве P хранит вектор координат исходных точек (узлов интерполяции) класса *Point*.

Можно рекомендовать следующий порядок проведения интерполяции

- создать вектор исходных точек, например, следующим образом.

```
var f:=x->(3*x-8)/(8*x-4.1);
var pt:=Partition(1.0,10.0,18).Select(x->new Point(x,f(x))).ToArray;
```

- создать объект класса *Spline*, при этом конструктор автоматически вызовет метод MakeSpline, вычисляющий коэффициенты сплайна

```
var Sp:=new Spline(pt);
```

- для получения значения сплайна в нужной точке x использовать метод Value

var *r:=Sp.Value(x);*

Метод Diff возвращает кортеж из первой и второй производных в указанной точке

var *(d1,d2):=Sp.Diff(x);*

4.4.2. Аппроксимация табличной функции полиномами Чебышева по методу наименьших квадратов (ApproxCheb)

Результат аппроксимации полиномами Чебышева с первого взгляда несколько непривычен: мы не можем записать удобную для вычисления аппроксимирующую функцию, поскольку получаем только её значение, вычисленное в заданных точках. В связи с этим после аппроксимации проводят дополнительный этап получения коэффициентов аппроксимирующего полинома в обычном его виде P(x). Такую операцию можно провести, например, построив интерполяционный полином по некоторому подмножеству полученных в результате аппроксимации значений. Следует заметить, что способ получения коэффициентов полинома не имеет значения – может быть построен канонический полином, использована интерполяционная схема Лагранжа или Ньютона и т.д., но конечный результат не изменится. В самом деле, через n+1 точку может пройти только единственный полином степени n.

В целях понижения степени полученного полинома (при некотором приемлемом снижении точности аппроксимации) можно попытаться провести операцию экономизации.

Следует учесть, что аппроксимация проводится для имеющейся совокупности точек, обычно получаемых экспериментально, поэтому бессмысленно задавать высокие требования к точности получаемого результата. На практике часто ограничиваются значениями точности порядка нескольких процентов от величины аппроксимируемых значений.

Существует также задача аппроксимации полиномами Чебышева сложной функции, представленной в аналитическом виде (например, замена разложения функции в степенной ряд), с целью её упрощения и тогда требования к величине погрешности могут быть весьма высокими. Для решения такой задачи используется совсем другой алгоритм.

Класс ApproxCheb выполнен на базе авторской программы, которая была реализована на языке Алгол-60 ЭВМ «М-222» в конце 70-х годов прошлого столетия, поэтому источник алгоритма, к сожалению, утрачен.

Источником данных служит табличная функция; аргументы хранятся в массиве «x», соответствующие им значения функции – в массиве «y». Значения функции после аппроксимации помещаются в массив «f». Для проведения аппроксимации достаточно создать объект данного класса, определив желаемую погрешность «e». Необходимая для достижения заданной погрешности степень полинома «r» определяется автоматически. Методом MakeCoef можно получить коэффициенты полинома в массиве «c» [6].

При необходимости интерполяции посредством данного полинома, достаточно определить объект класса Polynom (4.3.1) на основе массива «c» и воспользоваться методом Value:

```
var oP:=new Polynom(c);
Writeln(oP.Value(x));
```

Пример 1.

Пусть задана табличная функция в виде набора из двенадцати точек с аргументами -2, -1.75, -1.5, … 0.75. Точки не обязательно должны быть равноотстоящими, но так проще получить нужные значения. Получим значения функции с помощью полинома $y(x) = 2x - 5x^2 + 8x^3$ – задачей будет разыскать эти коэффициенты.

```
var e:=0.1;   // оценка погрешности
var x:=ArrGen(12,i->0.25*i-2); x.Println;
var y:=x.Select(z->2*z-5*Sqr(z)+8*z*Sqr(z)).ToArray; y.Println;
var oC:=new ApproxCheb(x,y,e);
oC.f.Println; // аппроксимированные значения
Println(oC.r,oC.tol); // предлагаемая степень полинома и вычисленная погрешность
oC.MakeCoef;
oC.c.Println;
```

Результаты

Аргументы

-2 -1.75 -1.5 -1.25 -1 -0.75 -0.5 -0.25 0 0.25 0.5 0.75

Исходные значения функции

-88 -61.6875 -41.25 -25.9375 -15 -7.6875 -3.25 -0.9375 0 0.3125 0.75 2.0625

Аппроксимированные значения функции

```
-88 -61.6875 -41.25 -25.9375 -15 -7.6875 -3.25
-0.937500000000007
-7.105427357601E-15 0.312499999999996 0.750000000000002
2.06250000000001
```

Рекомендованная степень полинома и оценка отклонений

```
3 8.6662716579557E-15
```

Рекомендованные коэффициенты полинома

```
-2.41584530158434E-13 1.99999999999966 -5.00000000000011 8
```

С очень высокой степенью точности вычисленные коэффициенты совпадают с исходными (0, 2, -5, 8).

Пример 2.

Теперь зашумим вычисленные в предыдущем примере значения функции небольшой случайной погрешностью.

```
var e:=0.05;
var x:=ArrGen(12,i->0.25*i-2); x.Println;
var y:=x.Select(z->2*z-5*Sqr(z)+8*z*Sqr(z)).ToArray; y.Println;
y.Transform(t->t*(1+Random(1,100)/1e5)); y.Println;
var oC:=new ApproxCheb(x,y,e);
var f:=oC.f.Println;
Println(oC.r,oC.tol);
oC.MakeCoef;
oC.c.Println;
```

Результаты

Аргументы

```
-2 -1.75 -1.5 -1.25 -1 -0.75 -0.5 -0.25 0 0.25 0.5 0.75
```

Исходные значения функции

```
-88 -61.6875 -41.25 -25.9375 -15 -7.6875 -3.25 -0.9375 0
0.3125 0.75 2.0625
```

Зашумленные исходные значения функции

```
-88.04136 -61.728830625 -41.272275 -25.948653125 -15.0072
-7.69165125 -3.25234 -0.93785625 0 0.312696875 0.7507425
2.0639025
```

Аппроксимированные значения функции

```
-88.0459145238095 -61.7203951856477 -41.2725426731602
-25.9520527939422 -15.0086213555888 -7.69194416569542
-3.25171703185703 -0.937635761668894 0.000603837273823515
0.313305957375947 0.75077479104228 2.06331453067765
```

Рекомендованная степень полинома и оценка отклонений

```
3 0.00312111328025948
```

Рекомендованные коэффициенты полинома

```
0.000603837274967489 2.00168064315528 -5.00429983072348
8.003244718985
```

Попробуем найти значение функции в точке x=0.45

```
var oP:=new Polynom(oC.c);
Writeln(oP.Value(0.45));
```

Получаем значение `0.616499999999582`

Сравним его с найденным по незашумленной функции: 0.6165

Согласитесь, неплохо.

Пример 3.

Аппроксимация функции, заданной таблично.

x	-2.5	-1.0	0	1.35	3.1	4.2	6.6	9.8
f	18.4	2.1	-2.6	0.35	7.0	5.9	-4.4	-2.5

Решение

```
var e:=1.5;
var x:=Arr(-2.5,-1.0,0.0,1.35,3.1,4.2,6.6,9.8);
var y:=Arr(18.4,2.1,-2.6,0.35,7.0,5.9,-4.4,-2.5); y.Println;
var oC:=new ApproxCheb(x,y,e);
oC.f.Println;
Println(oC.r,oC.tol);
oC.MakeCoef;
oC.c.Println;
```

Результаты

- аппроксимированные значения

```
18.8787149210766 0.303449492091142 -1.58080462027389
1.59356781696437 5.98021464557467 5.65468278969934
-4.01730392692446 -2.5625211182078
```

- рекомендованная степень полинома и оценка приближения

```
4 1.02067832612246
```

- коэффициенты полинома

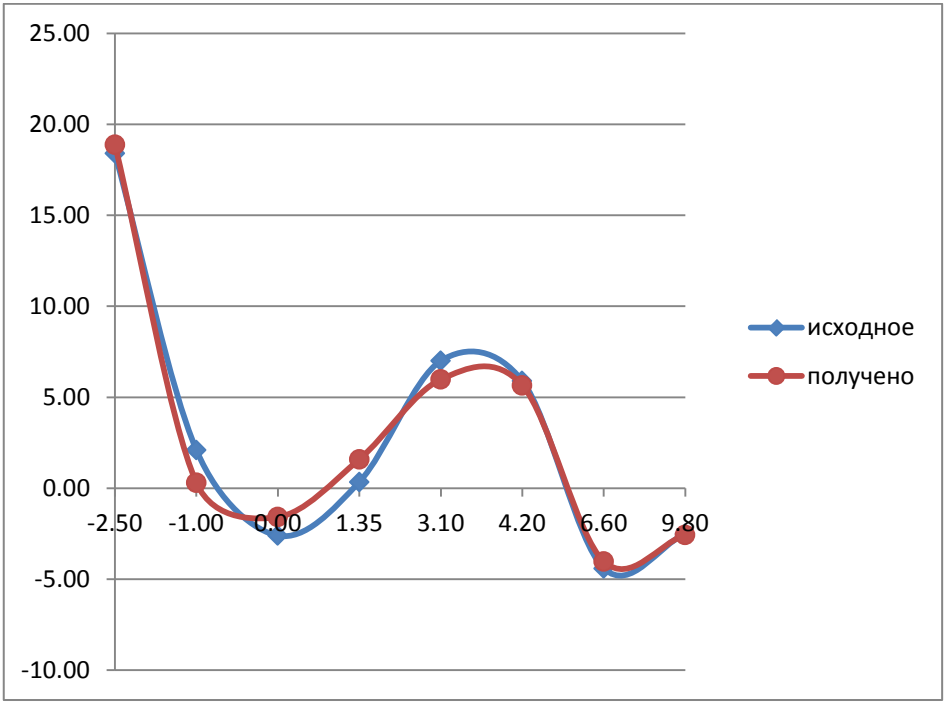
```
-1.58080462027389 0.641066344416852 1.94274720889745
-0.547701425011285 0.0348718228731455
```

Окончательно $y = -1.580805 + 0.641066x + 1.942747x^2 - 0.547701x^3 + 0.034872x^4$

Сравним результаты

x	-2.5	-1.0	0	1.35	3.1	4.2	6.6	9.8
f	18.40	2.10	-2.60	0.35	7.00	5.90	-4.40	-2.50
fрасч	18.88	0.30	-1.58	1.59	5.98	5.65	-4.02	-2.56

Результаты аппроксимации показаны на графике.



4.4.3. Экономизация полинома на интервале

Экономизация - прием, позволяющий уменьшить количество членов аппроксимирующего полинома путем корректировки его коэффициентов.

Запишем разложение функции f(x) по полиномам Чебышева

$$f(x) = \sum_{i=0}^{n} a_k x^k = \sum_{j=0}^{m} b_j T_j(x)$$

Показано [8], что если сначала перейти от степенного представления полинома к «чебышевскому», удалить один или более последних членов, а затем вернуться к исходному представлению, степень полинома может быть понижена без существенных потерь в точности.

Рассмотрим ряд, который получается при разложении функции ln(1+x):

$$\ln(x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \cdots + (-1)^{n+1}\frac{x^n}{n}$$

Ограничившись первыми пятью членами ряда, вычислим значение функции y=ln(x+1) для девяти точек в интервале (-1;1]

$$y(x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5} \quad (4.4.3-1)$$

x	-0.75	-0.5	-0.25	0	0.25	0.5	0.75	1
y(x)	-1.298	-0.689	-0.288	0.000	0.223	0.407	0.578	0.783

Используем известные соотношения для перехода к полиномам Чебышева:

$$x = T_1, \quad x^2 = \frac{T_0 + T_2}{2}, \quad x^3 = \frac{3T_1 + T_3}{4}$$
$$x^4 = \frac{3T_0 + 4T_2 + T_4}{8}, \quad x^5 = \frac{10T_1 + 5T_3 + T_5}{16}$$

Подставив эти соотношения в (4.4.3-1), после упрощения получим разложение по полиномам Чебышева

$$T(x) = -\frac{11}{32}T_0 + \frac{11}{8}T_1 - \frac{3}{8}T_2 + \frac{7}{48}T_3 - \frac{1}{32}T_4 + \frac{1}{80}T_5$$

Отбрасываем последний член в разложении и используем соотношения для возврата от полиномов Чебышева к полиномам в степенной форме

$$T_0 = 1, \ T_1 = x, \ T_2 = 2x^2 - 1, \ T_3 = 4x^3 - 3x, \ T_4 = 8x^4 - 8x^2 + 1$$

После проведения упрощений получаем

$$y_1(x) = \frac{15}{16}x - \frac{1}{2}x^2 + \frac{7}{12}x^3 - \frac{1}{4}x^4 \quad (4.4.3-2)$$

Пятая степень ожидаемо ушла, количество членов в выражении сократилось на один. Посмотрим, что произошло с точностью получаемых значений.

x	-0.75	-0.5	-0.25	0	0.25	0.5	0.75	1
y(x)	-1.298	-0.689	-0.288	0.000	0.223	0.407	0.578	0.783
y1(x)	-1.310	-0.682	-0.276	0.000	0.211	0.401	0.589	0.771

Максимальная абсолютная погрешность сделанной экономизации равна 0.012, относительная – 5.38%

Покажем, как проделать экономизацию посредством метода EconomSym класса Polynom (4.3.1)

```
var x:=ArrGen(9,-0.75,x->x+0.25);
var p:=new Polynom(0,1,-1/2,1/3,-1/4,1/5);
var r:=p.EconomSym(1,0.05);
Println(r.eps,r.n);
r.PrintlnBeauty;
for var i:=1 to x.Length do
  Write(r.Value(x[i-1]):0:3,' ');
```

x	-0.75	-0.5	-0.25	0	0.25	0.5	0.75	1
y(x)	-1.298	-0.689	-0.288	0.000	0.223	0.407	0.578	0.783
Y2(x)	-1.340	-0.698	-0.259	0.031	0.228	0.385	0.559	0.802

Здесь максимальная абсолютная погрешность сделанной экономизации равна 0.031, относительная – 8.06%

Результат, полученный при помощи метода EconomSym оказался хуже. Причина в том, что данный метод работает на симметричном интервале изменения аргумента. Был задан интервал [-1;1], в то время как аргумент определен на интервале [-0.75;1].

Повторим вычисления для интервала [-0.75;0.75]

```
var x:=ArrGen(8,-0.75,x->x+0.25);
var p:=new Polynom(0,1,-1/2,1/3,-1/4,1/5);
var r:=p.EconomSym(0.75, 0.05);
Println(r.eps,r.n);
r.PrintlnBeauty;
for var i:=1 to x.Length do
  Write(r.Value(x[i-1]):0:3,' ');
```

Результаты

```
0.003955078125 4
0.473958333333333x^3-0.640625x^2+0.980224609375x+
0.0098876953125
-1.286 -0.700 -0.283 0.010 0.222 0.399 0.585 0.823
```

x	-0.75	-0.5	-0.25	0	0.25	0.5	0.75
y(x)	-1.298	-0.689	-0.288	0.000	0.223	0.407	0.578
y1(x)	-1.310	-0.682	-0.276	0.000	0.211	0.401	0.589
y2(x)	-1.286	-0.700	-0.283	0.010	0.222	0.399	0.585

Здесь максимальная абсолютная и относительная погрешности экономизации, сделанной аналитически, по-прежнему составляют 0.012 и 5.38% соответственно, а применение метода EconomSym дало погрешности 0.04 и 5.21%, что несколько улучшило ситуацию.

Полученный полином не совпадает с найденным аналитически, поскольку в основу метода EconomSym положена достаточно простая рекуррентная формула, приведенная в [4] для алгоритма 38б, но дает вполне приемлемое для практики решение.

4.5. Векторная алгебра

Векторная алгебра представлена в NumLibABC операциями над векторами (класс Vector).

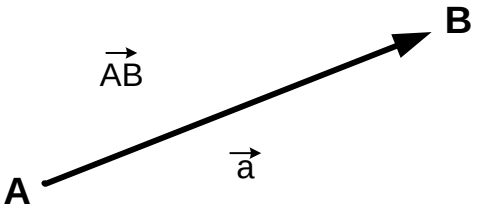
Векторы построены на основе одномерных динамических массивов типа real. Векторы нулевой размерности создавать бессмысленно: при попытке работать с таким вектором, он будет забракован.

Класс Vector был введен с тем, чтобы дать возможность работать с векторами в привычной терминологии. Нумерация компонентов векторов начинается с нуля, т.е. для вектора V в двухмерном пространстве (на плоскости) значение компоненты x запишется как V.Value[0], а значение компоненты y запишется как V.Value[1].

4.5.1. Базовые понятия векторной алгебры

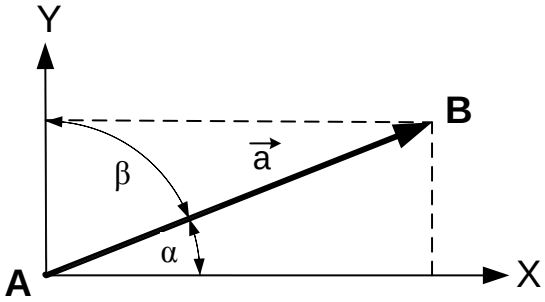
Изложение нестрогое и рассчитано на тех, кто этот предмет еще не изучал.

Вектором принято называть направленный отрезок, одна из граничных точек которого называется началом, а вторая – концом.



Обозначения вектора $\overrightarrow{AB}$ и $\vec{a}$ равноправны.

Вектор характеризуется длиной и направлением. Длина – это расстояние между концом и началом вектора. Направление в декартовой системе координат задается углами, которые вектор составляет с осями координат. Принято указывать не сами углы, а их косинусы, которые в этом случае называют *направляющими косинусами*.



Косинусами пользоваться достаточно удобно, поскольку умножая длину вектора на направляющий косинус мы получаем величину проекции вектора на соответствующую ось. В некоторых случаях бывает удобным пользоваться совокупностью проекций вектора на все оси системы координат (в векторной алгебре это называется разложением по ортонормированному базису. Орт – вектор с длиной равной единице, направленный из начала системы координат по координатной оси в сторону её положительного направления). В двухмерном пространстве орт абсциссы x обозначается i, орт ординаты y обозначается j. В трехмерном пространстве добавляется орт аппликаты z, обозначаемый k. Тогда, если x, y, z – проекции вектора, можно записать

$$\vec{a} = \{x, y, z\} = x\vec{i} + y\vec{j} + z\vec{k}$$

В таком представлении x, y, z – декартовы координаты вектора. Чаще всего векторы задаются именно набором декартовых координат.

Вектор можно найти по координатам его граничных точек. Пусть даны точка $A(x_a, y_a)$ и точка $B(x_b, y_b)$.

Тогда $\overrightarrow{AB} = \{x_b - x_a,\ y_b - y_a\} = \{x, y\}$

Длина вектора вычисляется как расстояние между его граничными точками и называется модулем вектора.

$$mod(a) = \left|\overrightarrow{AB}\right| = \sqrt{x^2 + y^2}$$

Два вектора называются *коллинеарными,* если они лежат на одной примой или на параллельных прямых. Если у коллинеарных векторов равны модули и направления, то вектора считаются *равными*.

Три вектора называются *компланарными*, если они лежат в одной плоскости или в параллельных плоскостях.

Для пары векторов определены операции сложения, вычитания, скалярного и векторного произведения. Для тройки векторов определено также смешанное произведение.

Скалярное произведение векторов – это сумма всех произведений их одноименных компонент. Скалярное произведение выражается скалярной величиной (числом) $\vec{a} \cdot \vec{b} = a_x b_x + a_y b_y + a_z b_z$

Результатом векторного произведения в декартовых координатах является вектор, разложенный по ортам.

$$\vec{a} \times \vec{b} = \begin{vmatrix} i & j & k \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{vmatrix} = i \begin{vmatrix} a_y & a_z \\ b_y & b_z \end{vmatrix} - j \begin{vmatrix} a_x & a_z \\ b_x & b_z \end{vmatrix} + k \begin{vmatrix} a_x & a_y \\ b_x & b_y \end{vmatrix}$$

Результат смешанного произведения трех векторов – скалярная величина, равная определителю, построенному из компонент этих векторов.

$$\vec{a} \times \vec{b} \times \vec{c} = \begin{vmatrix} a_x & a_y & a_z \\ b_x & b_y & b_z \\ c_x & c_y & c_z \end{vmatrix}$$

Отметим, что если три вектора компланарны, то их смешанное произведение равно нулю.

4.5.2. Класс Vector

Класс базируется на одномерных динамических массивах типа real. Доступны два открытых (public) свойства класса:

- Value: array of real – массив компонент вектора;
- Length: integer – размерность вектора (количество элементов в массиве Value).

Вектор создается одним из следующих способов:

а) вектор V с нулевыми компонентами размерности n

```
var V:=new Vector(n);
```

б) вектор V с заданными массивом “a” значениями компонент

```
var V:=new Vector(a);
```

в) вектор V с заданными перечислением компонентами

```
var V:=new Vector(x,y,z,…);
```

г) вектор V с заданными массивами “a” и “b” координатами граничных точек

var V:=**new** Vector(a,b);

Для вектора определены следующие методы и перегружена часть операций:

- нахождения модуля: V.ModV;
- унарный минус: –V;
- умножение вектора на скалярную величину V*R и умножение скалярной величины R на вектор R*V;
- разложение вектора по ортам: V.Ort, возвращающий вектор;
- создание копии вектора: V.Copy, возвращающий вектор:
- вывод компонент вектора на экран: V.Print; V.Println.

Методы вывода компонент вектора на экран имеют дополнительный параметр типа string, позволяющий задать разделитель между выводимыми значениями. По умолчанию используется пробел. Метод Printlnd после вывода значений всех компонент вектора дополнительно вызывает переход к новой строке.

Для пары векторов Va и Vb определены следующие методы

- сложение: vA+vB;
- вычитание: vA-vB;
- скалярное произведение: vA*vB;
- векторное произведение: vA.VP(vB);
- проверка коллинеарности: vA.IsCollinear(vB):boolean;
- проверка равенства: vA=vB.

Векторное произведение всегда определяется исходя из того, что векторы заданы в трехмерном пространстве. Если векторы двухмерные, то они приводятся к трехмерным заданием компоненты z=0. Результат также возвращается в виде трехкомпонентного вектора. Помним, что нумерация компонент в Value идет от нуля!

Для тройки векторов определены следующие методы

- условие компланарности: vA.IsCoplanar(vB, vC):boolean;
- смешанное произведение: vA.MP(vB, vC).

4.5.3 Примеры работы с классом Vector

1. $\left|2\overline{a}-\overline{b}\right|, \quad \overline{a}=\{3;-4;1\}, \quad \overline{b}=\{-1;0;5\}$

```
uses NumLibABC;

begin
  var a:=new Vector(3,-4,1);
  var b:=new Vector(-1,0,5);
  Writeln((2*a-b).ModV)
end.
```

Результаты

```
11.0453610171873
```

2. Разложить по ортам вектор {3;0;-4}

```
var p:=Arr(3.0,0.0,-4.0);
var a:=new Vector(p);
a.Ort.Println
```

Результаты

```
0.6 0 -0.8
```

3. Даны точки A(2;-1;2), B(1;2;-1) и C(3;2;1). Найти значение

$(\overline{BC} - 2\,\overline{CA}) \times \overline{CB}$

```
var A:=Arr(2.0,-1.0,2.0);
var B:=Arr(1.0,2.0,-1.0);
var C:=Arr(3.0,2.0,1.0);
var BC:=new Vector(B,C);
var CA:=new Vector(C,A);
var CB:=new Vector(C,B);
(BC-2*CA).VP(CB).Println(';')
```

Результаты

```
-12;8;12
```

4. Найти косинус угла между векторами a{2;-2;1} и b{2;3;6}

Косинус угла можно найти по формуле

$$\cos\varphi = \frac{a \cdot b}{|a| \cdot |b|}$$

```
var a:=new Vector(2,-1,1);
var b:=new Vector(2,3,6);
Writeln(a*b/(a.ModV*b.ModV))
```

Результаты

```
0.408248290463863
```

5. Найти площадь треугольника с координатами вершин A{7;3;4}, B{1;0;6} и C{4;5;-2}

Если пару смежных сторон треугольника, например, AB и AC, принять за векторы p и q, то из свойства векторного произведения можно найти площадь треугольника $S\Delta = |(\overline{p} \times \overline{q})|/2$

```
uses NumLibABC;

begin
  var A:=Arr(7.0,3.0,4.0);
  var B:=Arr(1.0,0.0,6.0);
  var C:=Arr(4.0,5.0,-2.0);
  var p:=new Vector(A,B);
  var q:=new Vector(A,C);
  Writeln(p.VP(q).ModV/2)
end.
```

```
Результаты: 24.5
```

4.6. Линейная алгебра

Линейная алгебра представлена в NumLibABC операциями над матрицами (класс Matrix).

Матрицы построены на основе двумерных динамических массивов типа real.

Класс Matrix дает возможность работать с матрицами в привычной терминологии.

ВАЖНО! Большинство методов класса оперируют понятием базы – индекса первого элемента в строке и столбце. По умолчанию значение базы равно единице, но можно задать значение 0, как у динамических массивов. Если в методе параметр базы отсутствует, то счет индексов ведется от нуля, как в динамических массивах..

Единичное значение базы, принятое по умолчанию, связано с принятой в линейной алгебре системой отсчета номеров строк и столбцов: они нумеруются от единицы.

Матричные операции тесно связаны с векторными. Любая строка или столбец матрицы может рассматриваться, как вектор, поэтому многие методы принимают или возвращают векторы.

4.6.1. Базовые понятия линейной алгебры

Как и для векторной алгебры, изложение материала будет нестрогим, поскольку оно рассчитано, в основном на тех, кто этот предмет еще не изучал.

Линейная алгебра – достаточно обширный предмет, но нас будут интересовать в основном матрицы и то, что с ними тесно связано.

Под матрицей мы будем понимать прямоугольную табличку, содержащую некоторые числовые значения. Как в любой таблице, в матрице различают строки и колонки, только колонки принято называть столбцами. Если число строк и столбцов совпадает, матрицу называют квадратной. Матрица в языке программирования отображается на двумерный массив.

Нумерация строк идет сверху вниз; столбцы нумеруются слева направо. Нумерация всегда ведется от единицы и элемент матрицы A, стоящий в левом верхнем можно записать в виде $A_{1,1}$. Принято сначала указывать номер строки, а затем номер столбца.

Пусть матрица имеет m строк и n столбцов. В таком случае говорят, что матрица имеет размер m×n. Если матрица содержит только одну строку, то её называет «вектор-строка», а если только один столбец – соответственно, «вектор-столбец». Это легко запомнить: как и для вектора, достаточно только одного номера, чтобы указать нужный элемент.

У квадратной матрицы элементы, идущие по направлению из левого верхнего угла в правый нижний образуют *главную диагональ*. У элемента на главной диагонали номер строки равен номеру столбца. Линейная алгебра обобщает понятие главной диагонали и на прямоугольные матрицы, но это нам не нужно. Диагональ, идущая из правого верхнего угла в левый нижний, называется *побочной диагональю* и используется гораздо реже.

Если у матрицы все элементы главной диагонали (они называются диагональными) ненулевые, а остальные равны нулю, матрица называется *диагональной*. Диагональная матрица, у которой все элементы на главной диагонали равны единице, называется *единичной*.

Если элементы, расположенные ниже и левее главной диагонали равны нулю, мы получаем *верхнюю треугольную матрицу*. А если равны нулю все элементы, расположенные выше и правее главной диагонали, получаем *нижнюю треугольную матрицу*.

Матрицу можно сложить с числом, можно умножить на число. Вычитание и деление – это сложение с обратным знаком и умножение на обратную величину, так что эти операции тоже корректны. Каждый элемент матрицы складывается (умножается и т.д) с заданным числом.

Можно найти сумму и разность двух матриц, но лишь при условии, что у них совпадают размеры. Совпадение размеров матриц означает, что у них совпадает число строк и столбцов. В этом

случае каждый элемент матрицы-результата является суммой (или разностью) соответствующих элементов исходных матриц.

С умножением матриц дело обстоит сложнее.

Во-первых, матрицы-сомножители должны быть совместимы: число столбцов первой матрицы должно равняться числу строк второй. В результате получается матрица с число строк, как у первой матрицы и с числом столбцов, как у второй: $a_{m,n} \times b_{n,k} = c_{m,k}$

Требование совместимости легко запомнить: выпишем индексы (m, ~~n~~) × (~~n~~, k) = (m, k). Одинаковые «внутренние» n зачеркиваем, а «внешние» размеры остаются.

Элементы матрицы-результата вычисляется по формуле

$$c_{i,j} = \sum_{p=1}^{n} a_{i,p} \times b_{p,j}$$

Легко понять, что матрицы a и b в операции умножения нельзя менять местами, потому что будет нарушено условие их совместимости. А если матрицы квадратные и одного размера? Тогда поменять можно, но результаты умножения будут различаться.

Можно ли умножить матрицу на вектор и вектор на матрицу? Да, можно, если будет выполнено условие совместимости. Вот тут нам и понадобится понятие вектора-строки и вектора столбца.

$p_{1,n} \times b_{n,k} = c_{1,k}$ $a_{m,n} \times q_{n,1} = c_{m,1}$

Вектор-строка может умножаться на матрицу и матрица может умножаться на вектор-столбец. Никак иначе.

А что произойдет при умножении вектора-строки на вектор-столбец?

$p_{1,n} \times q_{n,1} = c_{1,1}$

Получится матрица размером 1x1. Значение единственного элемента этой матрицы совпадет со скаляром, который дало бы скалярное произведение двух обычных векторов (4.5.1). Но отличие именно в типе результата, о чем нельзя забывать.

Существует ли операция деления матрицы на матрицу? Нет, она не существует. Но вспомним, что деление всегда можно заменить умножением на обратное делителю значение. И матрица имеет такую операцию – обращение, т.е. нахождение матрицы обратной данной. И запись классическая: A^{-1}. Операция обращения матрицы еще называется инверсией, поэтому можно также встретить обозначение Inv(A).

Обращение матрицы – сложная операция, которая не всегда возможна. Нас же не удивляют проблемы при делении на ноль? При обращении матрицы могут возникать аналогичные проблемы.

Итак, деление на матрицу мы заменяем умножением на обратную ей: $A/B = A \times B^{-1}$

Над матрицей A определена операция транспонирования A^T, при которой строки матрицы помещаются с сохранением порядка в столбцы матрицы-результата.

$$\begin{pmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{pmatrix}^T = \begin{pmatrix} a_{1,1} & a_{2,1} \\ a_{1,2} & a_{2,2} \end{pmatrix}$$

Одной из важнейших характеристик квадратной матрицы является ее определитель (детерминант) det. У него достаточно сложное математическое описание, из которого не вытекает особой практической пользы, поэтому достаточно знать: определитель – это некоторым образом составленная из элементов матрицы таблица, которая путем определенных вычислений приводится к числовому значению. В дальнейшем будет понятна польза знания значения определителя.

4.6.2. Класс Matrix

Класс базируется на двумерных динамических массивах. Пользователю доступны три открытых (public) свойства класса:

- RowCount: integer – количество строк в матрице;
- ColCount: integer – количество столбцов в матрице;
- Value: array[,] of real – элементы матрицы.

Для создания матрицы конструктор класса может быть вызван в одной из трех форм.

а) var Z:=new Matrix(m,n: integer) – создается матрица Z размером mxn, заполненная нулями;

б) var Z:=new Matrix(a: array [,] of real – создается матрица Z, элементы которой копируются из двумерного массива a;

в) var Z:=new Matrix(m,n: integer; params a: array of real) – создается матрица Z размером mxn, элементы которой построчно заносятся из массива a. Массив должен содержать mxn элементов.

Для матрицы определен довольно большой набор методов, а также перегружена часть операций:

- унарный минус –A;
- сложение матриц A+B;
- вычитание матриц A-B;
- умножение скаляра R на матрицу R*A;
- умножение матрицы на скаляр A*R;
- умножение матриц A*B;
- умножение вектора на матрицу V*A;
- умножение матрицы на вектор A*V;
- формирование диагональной матрицы Diag(n, R) размера nxn, диагональ которой заполнена значением R;
- выделение строки Row(k, base=1): Vector, где номер строки k берется в соответствии со значением базы;
- выделение столбца Col(k, base=1): Vector, где номер столбца k берется в соответствии со значением базы;
- замена содержимого k-й строки элементами вектора V - SetRow(k,V,base=1), где номер строки k берется в соответствии со значением базы;
- замена содержимого k-го столбца элементами вектора V – SetCol(k,V,base=1), где номер столбца k берется в соответствии со значением базы;

- вставка содержимого вектора V после k-й строки - InsertRowAfter(V,k,base=1), где номер строки k берется в соответствии со значением базы;

- вставка содержимого вектора V после k-го столбца - InsertColAfter(V,k,base=1), где номер столбца k берется в соответствии со значением базы;

- вставка содержимого вектора V перед k-й строкой - InsertRowBefore(V,k,base=1), где номер строки k берется в соответствии со значением базы;

- вставка содержимого вектора V перед k-м столбцом - InsertColBefore(V,k,base=1), где номер столбца k берется в соответствии со значением базы;

- удаление k-й строки - DeleteRow(k,base=1), где номер строки k берется в соответствии со значением базы;

- удаление k-го столбца - DeleteCol(k,base=1), где номер столбца k берется в соответствии со значением базы;

- суммирование j-й строки с i-й; результат в i-й строке – AddRow(i,j, base=1), где номера строк берутся в соответствии со значением базы;

- суммирование j-го столбца с i-м; результат в i-м столбце – AddCol(i,j, base=1), где номера столбцов берутся в соответствии со значением базы;

- вычитание j-й строки из i-й; результат в i-й строке – SubRow(i,j, base=1), где номера строк берутся в соответствии со значением базы;

- вычитание j-го столбца из i-го; результат в i-м столбце – SubCol(i,j, base=1), где номера столбцов берутся в соответствии со значением базы;

- умножение k-й строки на скаляр R – MultRow(k,R, base=1), где номер строки берется в соответствии со значением базы;

- умножение k-го столбца на скаляр R – MultCol(k,R, base=1), где номер столбца берется в соответствии со значением базы;

- обмен строк номер i и j – SwapRows(i,j, base=1), где номера строк берутся в соответствии со значением базы;

- обмен столбцов номер i и j – SwapCols(i,j, base=1), где номера столбцов берутся в соответствии со значением базы;

- транспонирование матрицы Transpose;

- детерминант матрицы Det: real;

- обратная матрица Inv: Matrix;

- копия матрицы Copy;

- вывод на экран элементов матрицы Println(w=6, d=2): Matrix, где w- ширина поля вывода, d – количество разрядов после точки;

- решение системы линейных алгебраических уравнений (СЛАУ) вида Ax=B, где A – матрица системы, B – вектор правых частей – SLAU(B, var cond:integer): Vector.

Метод SLAU – это оболочка к самостоятельному классу Decomp (4.6.4). Типичное обращение к методу имеет вид

```
var res:=oL.SLAU(Vrp, cond);
```

где oL – матрица системы, Vrp – вектор правых частей, cond – переменная типа real, в которую будет помещено число обусловленности матрицы M. Вектор решения будет помещен в res. Все «непонятные слова» разъясняются в описании класса Decomp.

4.6.3. Примеры работы с классом Matrix

1. Дана матрица A(3x4). Уменьшить каждый элемент второй строки на два, затем утроить каждый элемент первого столбца. Вывести на экран результат. Вставить вектор V {1,-1,0,2} перед третьей строкой. Обменять местами вторую строку с третьей, затем первую колонку с третьей. Вывести на экран результат. Транспонировать полученную матрицу на место исходной. Получить обратную матрицу и вывести её на экран. Исходная матрица A:

$$\begin{pmatrix} -2 & 4 & 0 & 3 \\ 6 & 11 & -5 & 7 \\ 0 & 8 & -4 & 1 \end{pmatrix}$$

```
uses NumLibABC;

begin
 var A:=new Matrix(3,4,-2,4,0,3,6,11,-5,7,0,8,-4,1);
 A.SetRow(new Vector(A.Row(2).Value.Select(x->x-2).ToArray),2);
 A.MultCol(1,3); // утроить каждый элемент 1-го столбца
 A.Println(3,0); Writeln;
 var V:=new Vector(1,-1,0,2);
 A.InsertRowBefore(V,3); // Вставить вектор V перед 3-й строкой
```

```
A.SwapRows(2,3); // Обменять местами строки 2 и 3
A.SwapCols(1,3); // Обменять местами колонки 1 и 3
A.Println(3,0); Writeln;
A:=A.Transpose; // Транспонировать полученную матрицу на место исходной
var Atr:=A.Inv; // Получить обратную матрицу
Atr.Println(13,9)
end.
```

Оператор, уменьшающий на 2 каждый элемент второй строки

```
A.SetRow(new Vector(A.Row(2).Value.Select(x->x-2).ToArray),2);
```

может вызвать у новичков вопросы, поэтому остановимся на нем подробнее.

- *A.Row(2)* возвращает в виде вектора строку матрицы A;
- *A.Row(2).Value* дает доступ к элементам вектора, как к динамическому одномерному массиву;
- *A.Row(2).Value.Select(x->x-2)* проецирует последовательность элементов массива на новую, уменьшая каждый элемент на 2;
- *A.Row(2).Value.Select(x->x-2).ToArray* превращает последовательность в динамический одномерный массив;
- ***new*** *Vector(A.Row(2).Value.Select(x->x-2).ToArray)* создает вектор из динамического массива;
- полученный вектор методом SetRow замещает вторую строку матрицы A.

Если такой способ непонятен, вместо этой строки кода можно вставить фрагмент с традиционным циклом.

```
for var j:=0 to A.ColCount-1 do A.Value[1,j]-=2;
```

Здесь будет полезно не ошибиться, помня, что индексы начинаются от нуля, поэтому вторая строка – это 1, а колонки меняются не от 1 до 4, а от 0 до 3, т.е. до *A.ColCount-1*

Результаты

```
-6  4  0  3
12  9 -7  5
 0  8 -4  1

 0  4 -6  3
 0 -1  1  2
-7  9 12  5
-4  8  0  1

 0.681102362  0.326771654  0.106299213  0.110236220
-1.677165354 -0.874015748 -0.440944882  0.283464567
 0.480314961  0.236220472  0.173228346  0.031496063
-1.090551181 -0.413385827 -0.303149606 -0.055118110
```

2. Для указанных матриц найти значение выражения

$$M = det(A - B \times C)^T \times A)$$

$$A = \begin{pmatrix} -3 & 0 & 4 & -1 \\ 2 & -7 & 5 & 6 \end{pmatrix} \quad B = \begin{pmatrix} 8 & 1 & -5 \\ 6 & 7 & 2 \end{pmatrix}$$

$$C = \begin{pmatrix} 1 & -1 & 7 & 0 \\ 3 & 2 & 9 & 4 \\ 5 & 0 & -2 & -4 \end{pmatrix}$$

```
uses NumLibABC;
begin
 var A:=new Matrix(2,4,-3,0,4,-1,2,-7,5,6);
 var B:=new Matrix(2,3,8,1,-5,6,7,2);
 var C:=new Matrix(3,4,1,-1,7,0,3,2,9,4,5,0,-2,-4);
 var M:=(((A-B*C).Transpose)*A).Det;
 Writeln(M)
end.
```

Результат: `1.12628132684428E-23`. На самом деле решением является точный ноль, но и полученное решение фактически представляет собой ноль.

3. Решить СЛАУ

$$\begin{cases} 2x_1 + 3x_2 - x_3 = 9 \\ x_1 - 2x_2 + x_3 = 3 \\ x_1 + 2x_3 = 2 \end{cases}$$

Решение

Матрица системы $A = \begin{pmatrix} 2 & 3 & -1 \\ 1 & -2 & 1 \\ 1 & 0 & 2 \end{pmatrix}$

Вектор правых частей $B = \begin{pmatrix} 9 \\ 3 \\ 2 \end{pmatrix}$

Матричное уравнение Ax=B

Вариант 1. В матричном виде $x = A^{-1} \times B$

```
uses NumLibABC;
begin
 var A:=new Matrix(3,3,2,3,-1,1,-2,1,1,0,2);
 var B:=new Vector(9,3,2);
 var x:=A.Inv*B;
 x.Println
end.
```

Результаты: `4 1.11022302462516E-16 -1`

Вариант 2. Метод Крамера

$\Delta = |A| = \det(A)$

$\Delta_{x1} = |A_{x1}| = \det(A_{x1})$, $x = \Delta_{x1}/\Delta$

где $A_{x1} = \begin{pmatrix} 9 & 3 & -1 \\ 3 & -2 & 1 \\ 2 & 0 & 2 \end{pmatrix}$ получается заменой столбца 1 вектором B.

Аналогичным образом строятся матрицы для других переменных.

```
var A:=new Matrix(3,3,2,3,-1,1,-2,1,1,0,2);
var B:=new Vector(9,3,2);
var det:=A.Det;
for var i:=0 to B.Length-1 do begin
 var t:=A.Copy;
 t.SetCol(B,i,0);
 var detx:=t.Det;
 var x:=detx/det;
 Print(x)
 end;
```

Результаты: `4 0 -1`

Вариант 3. Использование метода SLAU

```
uses NumLibABC;
begin
 var A:=new Matrix(3,3,2,3,-1,1,-2,1,1,0,2);
 var B:=new Vector(9,3,2);
 var cond:real;
 var x:=A.SLAU(B,cond);
 x.Print; Writeln(' Число обусловленности = ',cond)
end.
```

Результаты

```
4 0 -1 Число обусловленности = 1.97935318837932
```

Анализ трех вариантов решения СЛАУ показал, что все использованные способы решения дают схожие результаты. При этом матричный метод программируется наиболее коротко, но требует помнить формулу решения и обладает наихудшей точностью. Метод Крамера дает длинный код. Метод SLAU выглядит оптимальным, вследствие чего он и был добавлен в класс Matrix.

4.6.4. Решение СЛАУ с действительными коэффициентами (Decomp)

Класс Decomp является результатом переработки одноименной фортран-программы, приведенной в [1]. Там же приведены подробности для желающих разобраться в алгоритме работы одноименного метода Decomp.

Реализуется вычисление LU-разложения вещественной матрицы посредством гауссова исключения и оценка обусловленности матрицы. Класс используется для решения систем линейных уравнений, для чего надо вызвать метод Solve, передав ему вектор правых частей уравнения. Также имеется возможность получить определитель матрицы.

Метод Decomp позволяет получить оценку числа обусловленности матрицы cond, величина которой показывает близость матрицы к вырожденности. При точной вырожденности cond=MaxReal (1.7976931348623157E+308), а матриц со значением cond=0, похоже, не бывает. Если значение cond очень велико, вызывать метод Solve смысла нет: решение будет неверным.

Несмотря на то, что метод Decomp преобразует исходную матрицу системы A к упакованной комбинации верхней треугольной матрицы U и нижней треугольной матрицы L, его не следует пытаться использовать для получения LU-разложения матрицы A. Обе матрицы хранятся в модифицированном виде, не совпадающем с общепринятыми LU и PLU-разложениями.

После проведения разложения матрицы системы A можно вызвать метод Solve, передав ему вектор правых частей СЛАУ. Если правых частей несколько, достаточно вызывать этот метод необходимое число раз.

В классе Decomp имеются четыре public-свойства:

- a: array[,] of real – матрица системы перед вызовом метода Decomp; результат приведения к LU-форме после работы Decomp;
- ipvt: array of integer – вектор перестановок (подробности в [1]);
- cond: real – оценка числа обусловленности матрицы a;
- det:real – детерминант матрицы a.

Конструктор класса Decomp принимает матрицу системы A и передает её копию в свойство a. Затем конструктор вызывает метод Decomp. Этот метод извне класса недоступен, но вызывать его нет нужды.

Метод Solve вызывается с вектором правых частей и возвращает на её место вектор решений.

Пример решения СЛАУ из предыдущего раздела.

```
uses NumLibABC;
begin
  var A:=new real[3,3] ((2,3,-1),(1,-2,1),(1,0,2));
  var B:=new real[3] (9,3,2);
  var oL:=new Decomp(A);
  oL.Solve(B);
  B.Println;
  Writeln('cond=',oL.cond)
end.
```

Результаты

```
4 0 -1
cond=1.97935318837932
```

Результаты ожидаемо идентичны приведенным выше, поскольку во всех этих случаях фактически был использован один и тот же метод Decomp.

Теперь у вас есть четыре варианта решения СЛАУ. Вывод о том, нужен ли в классе Matrix метод SLAU, будет маленьким домашним заданием.

4.7. Решение обыкновенных дифференциальных уравнений

Обыкновенное дифференциальное уравнение (далее – ОДУ) связывает между собой значения независимой переменной x, некоторой неизвестной функции y(x) и её производных, либо дифференциалов

$$F\left(x, y, y', y'' \dots y^{(n)}\right) = 0 \quad (4.7 - 1)$$

Максимальный порядок входящей в ОДУ производной (n) называется порядком уравнения.

ОДУ имеет общее решение и частные решения.

Частное решение ОДУ на некотором, возможно бесконечном интервале – это функция y=φ(x), дифференцируемая n раз, которая на этом интервале обращается в ноль.

Общее решение ОДУ – это функция y=Φ(x,c1,c2, … cn), которая обращается на заданном интервале в ноль при конкретном наборе постоянных c1,c2, … cn, образуя частные решения.

Теория ОДУ доказывает, что уравнению (4.7 – 1) эквивалентна система уравнений

$$\varphi_k(x,\ y_1,\ y_1',\ y_2,\ y_2', \dots y_n,\ y_n'), \qquad k = 1,2, \dots n \quad (4.7 - 2)$$

Уравнение (4.7 – 1) имеет бесконечное множество решений. Чтобы решение было единственным (частным), нужно задать некоторые дополнительные условия и в зависимости от вида таких условий выделяют три основных типа задач [7].

1. Задача с начальными условиями (задача Коши) предполагает задание для некоторой точки x0 значений функции y(x0) и значений её производных. Начальные условия для (4.7 – 2) задаются в виде

$$y_1(x_0) = y_{10},\ y_2(x_0) = y_{20}, \dots y_n(x_0) = y_{n0}$$

2. Задача с граничными (краевыми) условиями содержит дополнительные условия в виде функциональных зависимостей между искомыми решениями. Минимальный порядок таких ОДУ равен двум.

Задача на собственные значения, включающая дополнительно к задаче Коши набор m неизвестных параметров $\lambda_1, \lambda_2, \dots \lambda_m$, называемых собственными значениями.

Лишь немногие ОДУ могут быть решены аналитически, что делает весьма актуальной задачей создание численных методов для приближения решения ОДУ. Большинство этих методов основано на решении задачи Коши.

Использование численного метода предполагает необходимость задания дополнительной информации [1]:

- указание величины ошибки, которую пользователь готов допустить в решении;

- указание цены, которую пользователь готов заплатить за получение решения.

Авторы книги [1] выделяют четыре основные категории методов численного решения задачи Коши для ОДУ.

- метод рядов Тейлора;
- метод Рунге – Кутты;
- многошаговые методы;
- метод экстраполяции.

В этой же книге авторы подробно обосновывают выбор численного метода, базирующийся на методе Рунге – Кутты.

4.7.1. Решение задачи Коши (RKF45)

Класс является переработкой для PascalABC.NET программы RKF45 [1,9], написанной на языке Fortran-90.

Алгоритм основан на формулах Рунге – Кутты – Фельберга (1970г) четвертого порядка и требует вычисления шести значений функции на каждом шаге. Точность определяется разностью значений между результатами методов четвертого и пятого порядков, что и определяет название алгоритма.

Метод Рунге – Кутты предполагает разбиение всего интервала изменения независимого аргумента x на некоторое количество точек, возможно, с переменным шагом. В каждой точке вместо истинного решения y=f(x) берется его некоторым образом аппроксимированное значение y(x). Формулы аппроксимации содержат значения функции и её первой производной, найденные для одной или нескольких предыдущих точек. Количество вычислений значений функции определяет порядок метода, т.е. классический метод Рунге – Кутты четвертого порядка требует четыре вычисления значения функции.

Сам метод Рунге – Кутты представляет собой несложный набор готовых формул для вычисления значений в одной точке. На его базе создается ***одношаговый интегратор***, позволяющий найти и оценить решение в точке, отстоящей от заданной на некоторый указываемый интегратору шаг. Полная программа RKF45 дает решение во всем наборе точек, вызывая внутренний одношаговый интегратор, а затем оценивая ход решения и его качество. Подробности можно найти в снабженном комментариями коде программы.

Фельберг (Fehlberg) в 1969-1970 годах предложил усовершенствовать методику вычислений с тем, чтобы получить эффективную оценку точности.

Порядок решения рассмотрим на примере, взятом из [1].

В задаче рассматривается движение двух тел под действием взаимного гравитационного притяжения. Начало системы координат зафиксировано в одном теле, а координаты второго тела во времени определяются как x(t) и y(t).

Показано, что задача сводится к системе уравнений

$$\begin{cases} x''(t) = \dfrac{-\alpha^2 x(t)}{R(t)} \\ y''(y) = \dfrac{-\alpha^2 y(t)}{R(t)} \end{cases}$$

где $R(t) = [x(t)^2 + y(t)^2]^{\frac{3}{2}}$

В качестве начальных условий предлагается задать

$$x(0)=1-e,\ x'(0)=0,\ y(0)=0,\ y'(0) = \alpha\sqrt{\frac{1+e}{1-e}}$$

Полагаем $y_1 = x,\ y_2 = y,\ y_3 = x',\ y_4 = y'$

Получаем систему из четырех уравнений с начальными условиями

$$\begin{cases} y_1' = y_3 \\ y_2' = y_4 \\ y_3' = -\dfrac{y_1}{R} \\ y_4' = -\dfrac{y_2}{R} \end{cases} \qquad \begin{cases} y_1(0) = 1 - e \\ y_2(0) = 0 \\ y_3(0) = 0 \\ y_4(0) = \alpha\sqrt{\dfrac{1+e}{1-e}} \end{cases}$$

$$R = \frac{[x(t)^2 + y(t)^2]^{\frac{3}{2}}}{\alpha^2}$$

Следует отметить, что в данном примере имя «e» для переменной было выбрано неудачно: это константа, означающая эксцентриситет (орбита представляет собой эллипс, в одном из фокусов которого помещено начало координат), а не основание натуральных логарифмов.

Требуется определить процедуру, принимающую значение независимого аргумента и возвращающую массивы с наборами вычисленных значений функции и её первой производной. Набор параметров фиксированный, поэтому в теле процедуры часть параметров может не использоваться. Для данной задачи процедура может выглядеть так:

```
procedure Orbit(t:real; y,yp:array of real);
// для независимого аргумента e возвращаются
// значения функции y[] и её первой производной yp[]
begin
  var alpha:=Sqr(ArcTan(1.0));
  var r:=y[0]*y[0]+y[1]*y[1]; r:=r*Sqrt(r)/alpha;
  yp[0]:=y[2]; yp[1]:=y[3]; yp[2]:=-y[0]/r; yp[3]:=-y[1]/r
end;
```

Размеры массивов y и yp определять в процедуре не требуется, но следует помнить, что нумерация элементов в них ведется от нуля.

Создадим и заполним начальными условиями массив y

```
var e:=0.25;
var y:=Arr(1.0-e,0.0,0.0,ArcTan(1)*Sqrt((1.0+e)/(1.0-e)));
```

Объявим и инициализируем переменные abserr и relerr определяющие абсолютную и относительную точности решения. Не следует задавать значение relerr меньше чем 1.0e-8; в то же время значение abserr чаще всего полагают нулевым.

```
var (abserr,relerr):=(0.0,0.3e-6);
```

Далее, создадим объект класса RKF45, передав ему четыре параметра. Существует еще и пятый параметр MsgOn, получающий по умолчанию значение true, подходящее в большинстве случаев и освобождающее пользователя от написания собственного обработчика ошибок (см. ниже).

```
var oL:=new RKF45(Orbit, y, abserr, relerr);
```

Дальнейшее решение задачи может проходить по различным схемам. Наиболее часто делается циклическое обращение к методу Solve, дающее решение для задаваемого значения t. В примере выводится решение для t, значение которого меняется от 0 до 12 с шагом 0.5

```
var (t,tb,th):=(0.0,12.0,0.5);
var t_out:=t;
repeat
  oL.Solve(t,t_out);
  Writeln(t:5:1,oL.y[0]:15:9,oL.y[1]:15:9);
  case oL.flag of
  -3,-2,-1,1,8:begin Writeln('Flag=',oL.flag); Exit end;
  2:t_out:=t+th;
  end
until t>=tb;
```

Результаты

```
 0.0    0.750000000    0.000000000
 0.5    0.619768004    0.477791367
  …
 3.0   -1.054031732    0.575705690
 3.5   -1.200735203    0.300160297
 4.0   -1.250000108   -0.000000390
 4.5   -1.200735112   -0.300161044
 5.0   -1.054031579   -0.575706338
 …
11.5   -1.200735807    0.300158340
12.0   -1.250000209   -0.000002338
```

Метод RKF45.Solve имеет два параметра и управляется свойством flag.

Первый параметр t – это значение независимой переменной, для которого ищется решение. Второй параметр t_out определяет значение t, на котором следует завершить работу метода.

Значение flag=1 (устанавливается по умолчанию при создании объекта класса RKF45) или flag=-1 обеспечивает начальную настройку для каждой новой задачи. Пользователь должен задавать flag=-1 лишь в том случае, когда необходимо самостоятельное

управление одношаговым интегратором. В этом случае предпринимается попытка продолжить решение на один маленький шаг в направлении t_out при каждом очередном вызове. Такой режим работы весьма неэкономичен и его следует применять лишь в случае крайней необходимости. Чтобы установить такое значение, после создания объекта oL следует указать oL.flag:=-1.

По результатам работы метода RKF45.Solve свойство flag может иметь одно из следующих значений:

-5	Критическая ошибка. Продолжение работы невозможно: пользователь не выполнил необходимых действий после получения на предыдущем шаге значения flag, равного 5,7 или 8
-4	Критическая ошибка. Продолжение работы невозможно: пользователь не выполнил необходимых действий после получения на предыдущем шаге значения flag, равного 6,7 или 8
-3	Это значение появляться не должно. Если оно все же появится, просьба сообщить автору.
-2	Сделан один шаг в направлении t_out, оказавшийся успешным. Это нормальное завершение, если пользователь выполнял одношаговое интегрирование.
2	При интегрировании успешно достигнуто значение t_out. Это нормальное завершение для типового режима работы.
3	Интегрирование не завершено, поскольку заданное значение границы относительной ошибки relerr слишкои мало. Для продолжения необходимо увеличить значение relerr. Если результат устраивает, установите flag=2 и продолжите работу.

4	Интегрирование не завершено, поскольку потребовалось более 3000 вычислений значения производной, что соответствует примерно 500 шагам. Пользователь может повторить обращение, при этом счетчик количества вычислений функции будет сброшен в ноль и, возможно, это приведет к успешному решению.
5	Интегрирование не завершено, поскольку решение обратилось в ноль, вследствие чего тест одной только относительной ошибки не проходит. Пользователь должен задать ненулевое значение abserr и продолжить работу. Возможно, использование одношагового интегратора будет хорошим выходом.
6	Интегрирование не завершено, поскольку требуемая точность не может быть достигнута даже при наименьшем допустимом шаге. Пользователь должен увеличить границу допустимой погрешности abserr и/или reller прежде чем пытаться продолжить решение. Также потребуется установить flag=2 (или -2 для вызова одношагового интегратора). flag=6 возникает в случае обнаружения проблемной точки, в которой или решение меняется очень резко, или присутствует сингулярность.
7	По всей видимости RKF45 неэффективен для данной задачи. Слишком большое значение требуемых для вычисления точек препятствует выбору шага. Можно попытаться использовать режим одношагового интегратора. Если пользователь продолжит решение, задав flag=2, работа будет завершена.
8	Неверно заданы входные параметры. Продолжение работы невозможно. Допущена одна из следующих ошибок: - t=tout и при этом flag $\neq \pm 1$; - relerr или abserr < 0; - flag = 0 или выходит за пределы [-5;8]

Установка свойства MsgOn в true позволяет обрабатывать большинство кодов ошибки внутренним методом класса. Если пользователю нужна более гибкая обработка, следует установить MsgOn в false и написать собственный обработчик.

Приведем пример обработки кода ошибки для случая MsgOn=true:

```
case oL.flag of
  -3,-2,-1,1,8: begin Writeln('Flag=',oL.flag); Exit end;
  2: t_out:=t+th;
  end;
```

Результат работы метода RKF45.Solve находится в свойствах -массивах y (значение функции) и yp (значение первой производной).

Рассмотрим еще два примера, приведенных в [10] (http://people.sc.fsu.edu/~jburkardt/f_src/rkf45/rkf45_prb.f90).

1. Решение одиночного дифференциального уравнения первого порядка при начальном условии y(0)=1 для t, меняющегося от 0 до 20 с шагом 5 (пример взят из [10]).

$$y'(t) = \frac{1}{4} y(t) \left(1 - \frac{y(t)}{20}\right)$$

Это уравнение имеет точное аналитическое решение, которое при указанных начальных условиях может быть записано

$$y(t) = \frac{20}{1 + 19e^{-0.25t}}$$

В приводимом примере получаемые значения сравниваются с вычисленными по приведенной выше формуле.

```
uses NumLibABC;

procedure f(t:real; y,yp:array of real);
 begin
   yp[0]:=y[0]/4*(1-y[0]/20)  //yp – массив производных y
 end;

begin
 var (abserr,relerr):=(0.0,1e-6);
 var (t,tb,th):=(0.0,20.0,5.0);
 var y:=Arr(1.0);
 var t_out:=t;
 var oL:=new RKF45(f,y,abserr,relerr);
 Writeln('  t        y((t)       эталон');
 Writeln('-----------------------------------');
 repeat
   OL.Solve(t,t_out);
   Writeln(t:5:1,oL.y[0]:15:6,20/(1+19*Exp(-0.25*t)):15:6);
   ss1+=oL.y[0];
   case oL.flag of
   -3,-2,-1,1,8: begin Writeln('Flag=',oL.flag); Exit  end;
   2: t_out:=t+th
   end
 until t>=tb
end.
```

Результаты

```
   t          y((t)          эталон
-----------------------------------
  0.0       1.000000       1.000000
  5.0       3.103859       3.103859
 10.0       7.813674       7.813675
 15.0      13.823252      13.823256
 20.0      17.730168      17.730166
```

2. Решим задачу при помощи одношагового интегратора.

```
uses NumLibABC;


 procedure f(t:real; y,yp:array of real);
 begin
   yp[0]:=y[0]/4*(1-y[0]/20)
 end;

begin
 var (abserr,relerr):=(0.0,1e-6);
 var (t,t_out,tb,te,ns):=(0.0,0.0,0.0,20.0,4);
 var y:=Arr(1.0);
 var oL:=new RKF45(f,y,abserr,relerr);
 Writeln('  t        y((t)       эталон');
 Writeln('-----------------------------------');
 oL.flag:=-1;
 f(t,y,oL.yp);
 Writeln(t:10:5,oL.y[0]:10:6,20/(1+19*Exp(-0.25*t)):15:6);
 for var i:=1 to ns do begin
   t:=((ns-i+1)*tb+(i-1)*te)/ns;
   t_out:=((ns-i)*tb+i*te)/ns;
   while oL.flag<0 do begin
     oL.Solve(t,t_out);
     Writeln(t:10:5,oL.y[0]:10:6,20/(1+19*Exp(-0.25*t)):15:6);
     case oL.flag of
       -3,-1,1,8:begin Writeln('Flag=',oL.flag); Exit end;
      end
     end;
   oL.flag:=-2;
 end;
end.
```

Результаты

(строки подчеркнуты для сравнения с предыдущим примером)

t	y((t)	эталон
0.00000	1.000000	1.000000
0.08411	1.020167	1.020167
0.50468	1.126899	1.126899
1.45355	1.407357	1.407357
2.43485	1.764139	1.764139
3.44044	2.212585	2.212586
4.22022	2.626294	2.626294
5.00000	3.103859	3.103859
6.10707	3.900533	3.900534
7.28157	4.905233	4.905234
8.55543	6.176652	6.176654
9.27771	6.972853	6.972855
10.00000	7.813673	7.813675
11.94332	10.206922	10.206922
13.47166	12.086292	12.086293
15.00000	13.823254	13.823256
16.81220	15.575736	15.575736
18.40610	16.796930	16.796928
19.20305	17.297358	17.297357
20.00000	17.730168	17.730166

4.8. Вычисление определенных интегралов

Известно большое количество машинных алгоритмов, реализующих численное вычисление определенных интегралов. Выбор алгоритма существенно зависит от того, как представлены исходные данные.

Пусть требуется найти численное значение некоторого интеграла, определенного на заданном отрезке, т.е.

$$I = \int_a^b f(x)\,dx$$

Далее предполагается, что либо значения f(x) заданы на интервале [a;b] множеством точек $y_i=f(x_i)$, либо имеется некоторое аналитическое выражение f(x), позволяющее вычислять значения для $x \in [a;b]$.

Можно представить значение интеграла I суммой интегралов на каждом из подинтервалов, образованных парой соседних точек x

$$I = \sum_{i=1}^{n} \int_{x_i}^{x_{i+1}} f(x)\,dx$$

«Интегральчик», вычисляемый на отрезке $[x_i;x_{i+1}]$, называется квадратурой. Когда-то его предлагалось считать, как площадь прямоугольника со сторонами длиной x_{i+1}-x_i и $y_i=(f(x_{i+1})+f(x_i))/2$ – это и есть основа квадратурной формулы прямоугольников. Затем появились квадратурные формулы трапеций, криволинейных трапеций (формула Симпсона), её развитие – формула Ромберга и т.д.

Имеется также метод, при котором значения функции интерполируются сплайном, а поскольку сплайн – это кубический полином, аналитическое выражение интеграла от такой функции хорошо известно. К сожалению, сплайн-интерполяция не всегда позволяет получить результаты с нужной точностью.

Сочетание высокой точности и большой скорости вычисления квадратуры дают адаптивные программы, подбирающие комбинацию величины интервалов разбиения исходного отрезка и варианта квадратурной формулы так, чтобы обеспечивать необходимую точность.

4.8.1. Адаптивная квадратурная программа (Quanc8)

В пакет входит класс Quanc8, полученный переработкой для PascalABC.NET подпрограммы QUANC8 [1], написанной на языке Fortran.

Для создания объекта требуется набор переменных:

f – интегрируемая функция;

a, b - границы интервала интегрирования;

abserr - заданная предельная абсолютная ошибка;

relerr - заданная предельная относительная ошибка;

Метод Value возвращает кортеж из четырех элементов:

[0] - результат интегрирования (res)

[1] - оценка абсолютной ошибки результата (errest)

[2] - индикатор надежности (flag)

[3] - число точек, в которых вычислялась функция (nofun)

Если flag ненулевой, но мал, результат еще можно принять, если велик, то Quanc8 нельзя применять для вычисления данной функции. Значение flag имеет вид XXX.YYY, где XXX - количество подинтервалов длины $(b-a)/2^{30}$ с недопустимо большой ошибкой вычисления, 0,YYY - часть необработанного основного интервала.

В качестве иллюстрации работы с Quanc8 найдем значение интегрального синуса на [0;2]

$$\mathrm{Si}(2) = \int_0^2 \frac{\sin x}{x}\,dx$$

```
var f:real->real := x->x=0?1.0:sin(x)/x;
var oL := new Quanc8(f,0,2,1e-7,0);
Writeln(oL.Value);
```

Результат: (1.60541297680269, 1.13735457459156E-16, 0, 33)

Прежде всего проверяем flag – его значение равно нулю, следовательно требуемая точность достигнута. Мы запрашивали точность 10^{-7}, но получили оценку порядка 1.137×10^{-16}, т.е. в пределах машинной точности. Отлично. Значение 1.60541297680269 – результат, в котором, по-видимому, все цифры верны. И для такой точности понадобилось всего 33 итерации. Отметим, что в данном случае указание точности роли не играет: можно задать даже abserr=relerr=1.0, а результат будет тот же!

Так Quanc8 ведет себя потому, что функция «хорошая» для взятого алгоритма. Если заменить sin x на tg x, функция f(x) будет иметь особенность в окрестности точки $x=\pi/2\approx1.5708$ и это создаст

проблему. Величина flag станет недопустимо большой для того, чтобы принять результат. Справедливости ради следует отметить, что такую квадратуру не смогли вычислить и другие машинные программы.

```
var f:real->real:=x->x=0?1.0:tan(x)/x;
```

Интересно, что в [1] рассматривается поведение Quanc8 при вычислении квадратуры функции y=tg(x)/x на интервале [0;2] и приводится решение, в котором flag=91.21 с последующей оценкой участка интервала, где выявлена особенность. Но получить это значение в PascalABC.NET не удается. Более того, если приведенную в [1] программу перевести на работу с двойной точностью (DOUBLE PRECISION), она выдает такой же результат, как и метод Quanc8 (flag=30). Причина оказалась в точности представления коэффициентов формулы Ньютона-Котеса восьмого порядка, которая заложена в основу Quanc8. Достаточно их вычислить с использованием 32-битной арифметики чтобы получить решение, совпадающее с приведенным в [1].

PascalABC.NET использует единственный вещественный тип real, которому в классах платформы .NET соответствует System.Double, реализующий 64-битную арифметику. Для работы с 32-битной арифметикой в PascalABC.NET можно использовать класс System.Single.

В Quance8 используются следующие константы формул Ньютона-Котеса

$$\frac{3956}{14175}\ \frac{23552}{14175} - \frac{3712}{14175}\ \frac{41984}{14175} - \frac{18160}{14175}$$

При желании вычислить их с 32-битной точностью (аналог REAL в Fortran) можно использовать следующую запись

```
var w0 := System.single(3956.0)/System.single(14175.0);
```

4.9. Задачи оптимизации функций

Под термином «оптимизация функции» понимается нахождение её минимума или максимума. Сама функция в терминах задачи оптимизации называется целевой. Поскольку максимум целевой функции можно представить как её минимум, взятый с обратным знаком, достаточно рассматривать только задачи нахождения минимума.

Пусть задана некоторая действительная функция от n действительных аргументов F(x1,x2, ... xn). Если ее минимум ищется безотносительно каких-либо ограничений на аргументы, то говорят о безусловной оптимизации. В противном случае задаются некоторые ограничения, обычно имеющие вид нелинейных функций, удовлетворяющих набору неравенств или уравнений.

Если функция F(x1,x2, ... xn) и все ограничения имеют вид линейных функций, задачу относят к области линейного программирования (ЛП-задача), в противном случае можно говорить о задаче нелинейного программирования.

Решение ЛП-задач приводит к необходимости решения разреженных систем линейных уравнений высоких порядков и для этой цели используются специализированные алгоритмы, теория которых достаточно хорошо разработана.

Оптимизация может быть глобальной, если разыскиваемый минимум является глобальным и локальной в противном случае. Задача поиска глобального минимума может быть сведена к поиску

всех локальных минимумов и выбором среди них нужного. Если аргумент единственный, говорят о поиске минимума функции одной переменной (одномерной оптимизации), если аргументов несколько – о поиске минимума функции многих переменных.

4.9.1. Одномерная оптимизация (FMin)

FMin – класс, написанный на основе программы Джона Буркардта для языка Fortran-90 [10]. Использованный алгоритм был опубликован Ричардом Брентом [11].

Ищется локальный минимум функции f(x) на интервале [a;b]. Метод использует комбинацию поиска золотого сечения и последовательной параболической интерполяции. Сходимость никогда не бывает хуже, чем при фибоначчиевом поиске. Если функция имеет непрерывную положительную вторую производную в точке минимума, не совпадающем с текущими границами интервала поиска, сходимость сверхлинейная и обычно имеет порядок 1.324...»

Типичное использование FMin:

- определить оптимизируемую функцию *f(x:real):real;*
- создать объект ***var*** *oL=**new** FMin(f, a, b);*
- получить минимум ***var*** *(x, fx):=(oL.x, oL.Value);*

По умолчанию точность нахождения аргумента принята равной 1.05e-8, т.е. найденное значение аргумента будет иметь максимум восемь верных цифр. Это особенность алгоритма: точность решения не может превышать квадратного корня из машинной точности.

Предполагается без проверки, что интервал изоляции минимума [a;b] определен, в противном случае пользоваться FMin некорректно. При желании можно задать более грубое приближение, указав при создании объекта четвертым параметром величину максимально допустимого интервала неопределенности решения, т.е длину отрезка, содержащего минимум, что можно рассматривать как некий аналог точности решения.

```
var oL=new FMin(f, a, b, 1.5e-4);
```

Правильно задавая интервал изоляции, можно находить локальные минимумы и для функций, имеющих разрывы.

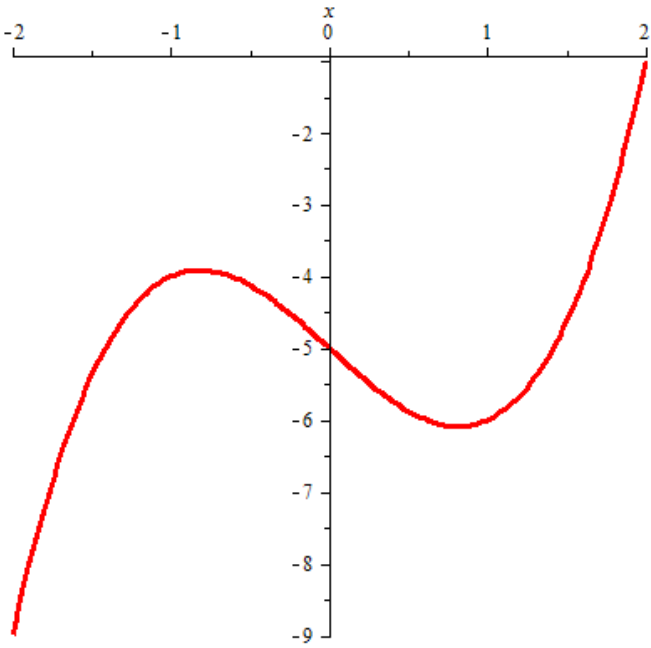
Рассмотрим работу с классом на примере. Пусть дана функция $f(x) = x^3 - 2x = 5 \rightarrow min$

Точки локальных экстремумов можно найти аналитически.

$$f'(x) = 3x^2 - 2;\ f'(x) = 0;\ 3x^2 = 2 \rightarrow x = \sqrt{\frac{2}{3}} \approx \pm 0.81649658$$

$f''(x) = 6x$
$f''(x_1) \approx -0.816 < 0;\ (x_1) \rightarrow max$
$f''(x_2) \approx +0.816 > 0;\ (x_2) \rightarrow min$

Можно заключить, что функция имеет две точки перегиба. При x<-$\sqrt{(2/3)}$ функция монотонно убывает, при x>$\sqrt{(2/3)}$ функция монотонно возрастает. Следовательно, нужно аккуратно отнестись к указанию левой границы интервала изоляции минимума.



Минимум функции будем искать на интервале [-1;1].

```
var fun:real->real:=x->x*Sqr(x)-2*x-5;
var oL:=new Fmin(fun,-1,1);
Println(oL.x, oL.Value)
```

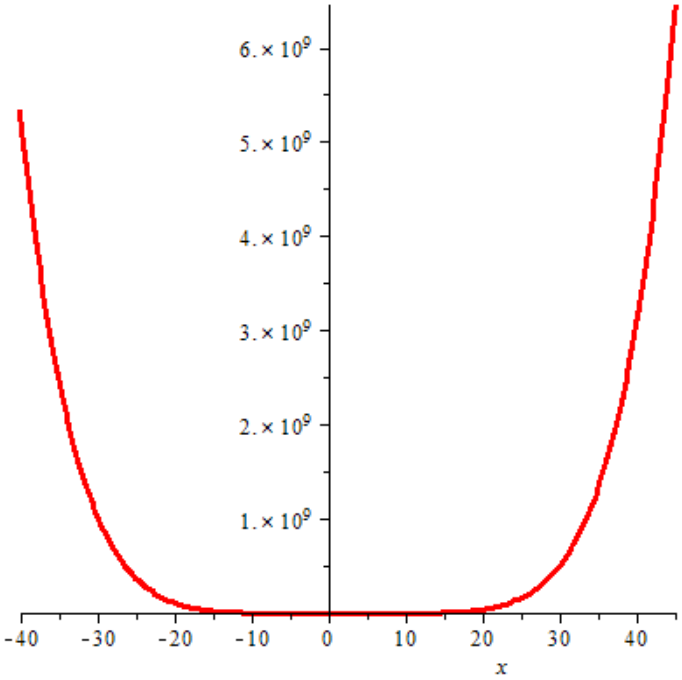
Получаем результат **0.8164965**71453584 -6.08866210790363, в котором для аргумента можно доверять восьми цифрам и это предел точности решения, которое в данном случае должно быть равно 0.81649658092772. «Виновата» функция – значение минимума на самом деле примерно равно -6.08866210790363, т.е. все найденные цифры значения минимума функции верны. В этих условиях уточнить значение аргумента не удается.

Зная примерное положение минимума, можно попытаться найти более точное приближение, сужая интервал изоляции, но и в

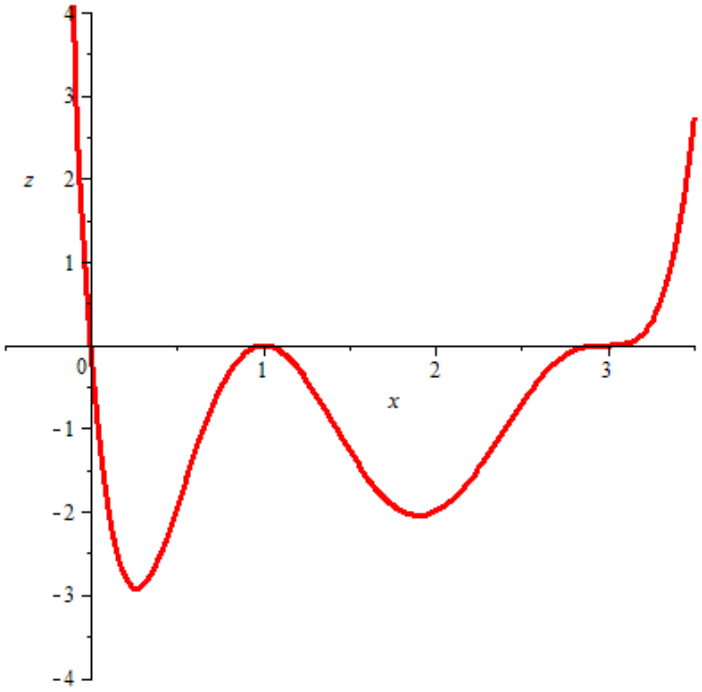
этом случае не удается найти решение точнее восьми знаков после запятой.

В качестве второго примера рассмотрим «коварную» функцию, имеющую в области минимума достаточно плоское и к тому же, волнистое донышко

$y(x) = x(x-1)^2(x-3)^3$



Рассмотрим «донышко» подробнее.



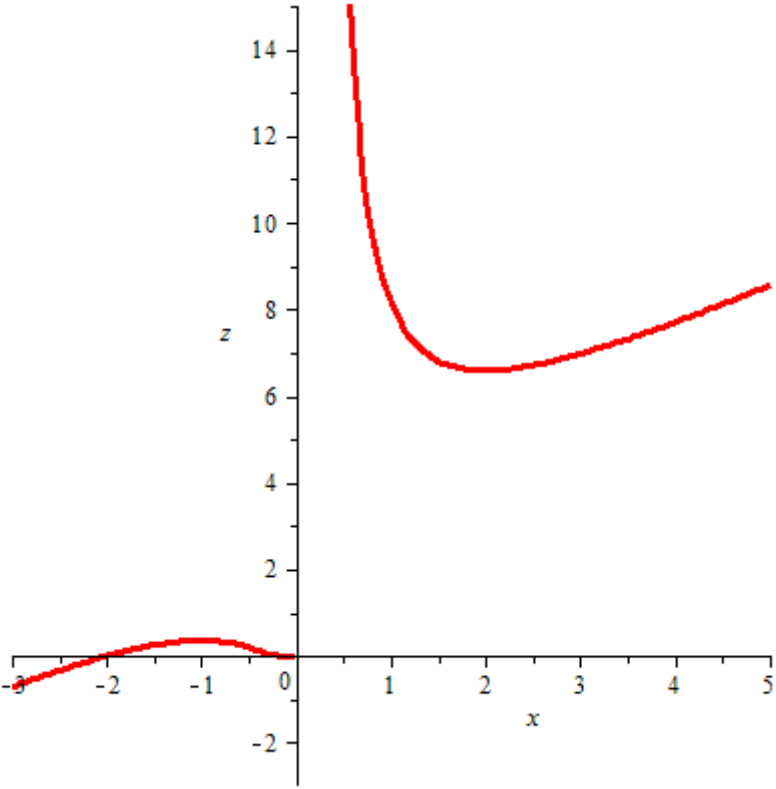
Собственно, ничего неожиданного. Достаточно одного взгляда на заданную функцию, чтобы понять: её значение обращается в ноль в точках 0, 1 и 3. Поскольку это полином, то естественно ожидать наличие локальных экстремумов в интервалах, окружающих нули. Поэтому тут нужна особая аккуратность при задании интервалов изоляции.

Задав интервал [-5;5], находим минимум в точке $x \approx 1.904$ и это неожиданность. Интервал [-5;1] приводит к минимуму в точке $x \approx 0.263$ (что ожидаемо). Интервал [1;2] приводит к ожидаемой точке $x \approx 1.904$.

Можно утверждать, что метод успешно справляется с подобными функциями при наличии правильного подхода к выбору интервала изоляции.

Последний пример – это еще одна «трудная» функция, имеющая разрыв.

$$y(x) = (x+2)e^{1/x}$$



Локальный минимум у функции есть и он находится в точке x=2. Особенность в точке x=0 создает определенные трудности при вычислении, но мы помним, что при работе с плавающей точкой

PascalABC.NET умеет самостоятельно управляться с делением на ноль, не давая переполнений. А вот сумеет ли FMin?

```
var fun:real->real:=x->(x+2)*Exp(1/x);
var oL:=new Fmin(fun,-1,3);
Println(oL.x, oL.Value);
```

Для отрезка [-1;3] получены значения 1.99999999814853 6.59488508280051 и это отличный результат. А вот отрезок [-5;3] показал другой минимум, который на самом деле не является таковым: -0.00083255433715704 0. Функция слева лишь стремится к нулю, но не достигает его. Тут все зависит от интерпретации результата.

$$\lim_{x\to 0^-} y(x) = 0$$
$$\lim_{x\to 0^+} y(x) = +\infty$$

4.9.2. Многомерная оптимизация (FMinN)

Решение задачи многомерной оптимизации является достаточно непростой проблемой. Объём вычислений растет по степенной функции с увеличением количества аргументов. Если для одного аргумента требовалось, к примеру, сделать 100 вычислений, то для пяти понадобится сделать не менее 10 миллиардов таких вычислений.

Принято различать прямые и косвенные методы оптимизации.

Прямые методы основаны на вычислении значений функции по набору значений её аргументов. Косвенные – на использовании условий математического определения экстремумов. Использова-

ние прямых методов приближает решение путем итераций. Косвенные методы ведут к решению без рассмотрения точек, где экстремумов быть не может. В прямых методах используется только вычисленное значение функции, поэтому они еще называются методами нулевого порядка. Косвенные методы требуют вычислять также первую и вторую производные функции и называются методами первого и второго порядка соответственно.

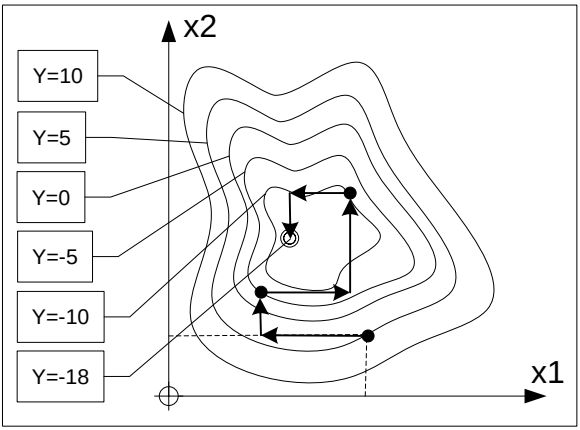
К методам нулевого порядка относят методы Розенброка, Хука – Дживса, симплекс-метод, методы, использующие случайные числа. К методам первого порядка относят градиентные методы. Методы второго порядка – это, например, метод Дэвидона – Флетчера – Пауэлла.

Любой существующий в настоящее время алгоритм многомерной оптимизации позволяет найти только один из экстремумов, а какой именно - зависит от выбора начального приближения. В общем случае эффективными могут оказаться комбинированные методы, в которых сначала находятся некоторые начальные приближении простыми разновидностями методов прямого поиска, а затем делается уточнение с использованием более сложных методов.

Почему для решения задач многомерной оптимизации имеется столько различных методов и ни один из них не признан наилучшим? Все дело в характере целевых функций. Если они гладкие, то любой простой метод дает быстрое хорошее решение. К со-

жалению, таких целевых функций очень мало. Гораздо чаще встречаются «плохие» с точки зрения оптимизации функции,

Рассмотрим функцию двух аргументов. Введем понятие «линия уровня» - линии на плоскости, во всех точках которой функция принимает одно и то же значение.



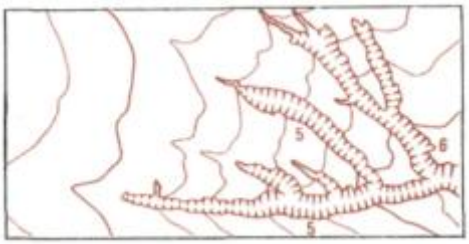
Если вспомнить, как строится рельеф на топографической карте, можно представить это изображение, как некоторую яму. Например, мы можем считать, что её внешний край находится на высоте 10м. Следующая замкнутая кривая соединяет точки ямы, где высота равна 5м. И так далее. Дно нашей ямы – это отметка -18 метров.

Проиллюстрирована работа метода покоординатного спуска. Берется некоторая точка. Из нее делается шаг по первому аргументу в направлении, уменьшающем значение функции. Затем делается шажок по второму аргументу. Получаем новую точку. Повторяем эти шаги, пока не окажемся на дне ямы. Поиск ведется ме-

тодом проб и ошибок: если при очередном шаге значение функции растет – делаем удвоенный шаг в обратном направлении.

С виду все просто и достаточно эффективно. Но еще Л.Н.Толстой в стихотворении, посвященном Крымской войне писал: «Чисто вписано в бумаге, Да забыли про овраги, А по ним ходить...».

Овраги у математиков тоже есть. Точнее, есть понятие «овражная функция». Это такая, у которой линии уровня имеют точки излома. Почему такое странное название? Вспомните, как на карте выглядит овраг. Покоординатный спуск в этом случае не работает. Кроме того, при попадании в одно из ответвлений оврага, шансов перебраться в другое, более глубокое, практически нет.



Другая проблема подстерегает нас на функциях типа «тарелка». У такой функции большое плоское «дно» - целое днище, т.е. её минимум достигается для огромной комбинации параметров и многие методы зацикливаются, попадая на это дно и пытаясь найти направление для следующего шага.

Еще одна, часто встречающаяся проблема – наличие различных ограничений, накладываемых на аргументы, таких как целочисленность аргумента, принадлежность к определенному интер-

валу и т.д. Решить эту проблему часто помогает введение штрафной функции.

Штрафная функция строится так, чтобы любое нарушение условий увеличивало значение этой функции. Каждое нарушение может вносить вклад в рост значения штрафной функции в зависимости от величины этого нарушения и его важности. Штрафная функция складывается с целевой и оптимизируется результирующая функция. К сожалению, ввести штрафную функцию позволяет не каждый метод оптимизации.

Есть и другие проблемы, но даже перечисленные позволяют понять, что задача оптимизации – отнюдь не простая задача.

4.9.2.1. Поиск минимума методом Хука-Дживса

Класс FMinN содержит метод HJ, написанный на основе программы из пакета [11].

Алгоритм Хука – Дживса (Hooke R., Jeeves T.A., 1961) решает задачу оптимизации путем применения двух подзадач – исследующего поиска и поиска по образцу. Он предполагает, что целевая функция унимодальна (т.е. имеет один экстремум) и ограничения отсутствуют. Если эти условия не выполняются, результат оптимизации может оказаться неверным. Существуют приемы, позволяющие обходить эти условия. Работа алгоритма хорошо описана в [7].

Если функция не унимодальна и алгоритм встречает локальный минимум, имеется существенная вероятность, что этот минимум после уточнения станет решением. Этим объясняется очень

большая важность правильного выбора набора аргументов, определяющего стартовую точку решения. Проблему такого выбора могут помочь решить включенные в класс методы случайного поиска.

Рассмотрим схемы решения задач оптимизации при помощи метода HJ.

I. В задаче нет ограничений, начальное приближение известно.

В качестве содержательного примера найдем минимум функции Розенброка с начальным приближением (-1.2 ; 1.0). Эта функция почти всегда используется для оценки алгоритмов оптимизации, поскольку поиск её глобального минимума считается непростой задачей. Функция Розенброка имеет глобальный минимум 0 в точке (1,1).

1. Описываем целевую функцию с аргументами x[0],x[1], …

```
function f(x:array of real):real; // функция Розенброка
begin
  Result:=100*Sqr(x[1]-Sqr(x[0]))+Sqr(1-x[0])
end;
```

2. Задаем вектор стартового решения

```
var xb:= Arr(-1.2,1.0);
```

3. Создаем объект класса FMinN

```
var oL:=new FMinN(xb, f);
```

4. Находим конечный вектор решения

```
var r:=oL.HJ;
```

5. Выводим результаты

```
Writeln('Количество итераций: ',oL.iter);
```

```
Write('Значения аргументов: '); r.Println;
Writeln('Полученное значение функции: ',f(r));
```

Результаты

```
Количество итераций: 19
Значения аргументов: 1.00000076293945 1.00000190734863
Полученное значение функции: 1.51339481672938E-11
```

Метод HJ имеет три параметра со следующими значениями по умолчанию: eps=1e-6; rho=0.5; itermax=5000, определяющие соответственно точность решения, параметр изменения шага и предельное количество итераций. Два последних параметра без веских причин лучше не трогать.

II. В задаче нет ограничений, начальное приближение неизвестно.

Функция, имеет два равных минимума со значением -5 в точках с координатами (-√2, √2) и (√2, -√2).

$$f(x,y) = x^4 + y^4 - 2x^2 + 4xy - 2y^2 + 3 \to min$$

Алгоритм Хука-Дживса требует задать начальное приближение, а по условию задачи у нас его нет. Зададим начальную точку (0,0) и посмотрим, что получится.

```
var f:function(x:array of real):real:= x->Power(x[0],4)+
   Power(x[1],4)-2*Sqr(x[0])+4*x[0]*x[1]-2*Sqr(x[1])+3;
var xb:=Arr(0.0, 0.0);
var oL:=new FMinN(xb,f);
var r:=oL.HJ();
Write('Вектор аргументов: '); r.Println;
Writeln('Значение функции: ', f(r));
```

Результаты

```
Вектор аргументов: 1.41421318054199 -1.41421318054199
Значение функции: -4.99999999999767
```

Минимум найден. А как быть со вторым минимумом? Мы можем попытаться варьировать координаты исходной точки. Эксперимент с наборами координат (-20,-20), (-20,20) дал те же значения. В то же время наборы (20,-20) и (20,20) дали решение

```
Вектор аргументов: -1.41422271728516 1.41422271728516
Значение функции: -4.99999999865899
```

Мы искали два решения потому, что заранее знали: их два. В реальных условиях такого знания, как правило, нет. В подобной ситуации может быть полезным случайный поиск.

4.9.2.2. Случайный поиск

Класс FMinN содержит четыре метода, основанные на случайном поиске.

Вернемся к аналогии с ямой. Если пойдет дождь, то каждая капля упадет на некоторое место в этой яме. Сравнивая отметки глубин, на которых оказались капли, можно сделать вывод о рельефе, в частности, найти самые глубокие места и их количество. Главное – чтобы было достаточно капель и они падали равномерно. А это может обеспечить датчик случайных чисел.

Первый из методов, MKSearch, реализует классический алгоритм Монте-Карло.

Если минимум не один, можно несколько раз обратиться к MKSearch и проанализировать результаты.

```
var f:function(x:array of real):real:= x->Power(x[0],4)+
   Power(x[1],4)-2*Sqr(x[0])+4*x[0]*x[1]-2*Sqr(x[1])+3;
var a:=Arr(-20.0,-20.0);
var b:=Arr(20.0,20.0);
var y:real;
var oL:=new FMinN(a,f);
oL.MKSearch(a,b,y);
Write('Вектор аргументов: '); oL.x.Println;
Writeln('Значение функции: ', y);
```

Здесь a – вектор начальных значений границ по каждому аргументу, b – вектор конечных значений. У метода есть четвертый параметр m, принимающий по умолчанию значение 1000. Это число испытаний - тех самых «капель дождя». Найденный вектор аргументов является свойством x класса FMinN.

Рассмотрим результат пятикратного обращения к этому методу с границами от -20 до 20 по каждому аргументу.

```
Вектор аргументов: 1.2654088164053 -1.64368239307948
Значение функции: -4.06245279895189
Вектор аргументов: 1.29792704307378 -0.892074956042727
Значение функции: -3.1209971083186
Вектор аргументов: 1.66633742939045 -1.37476657581272
Значение функции: -4.21463667660643
Вектор аргументов: -1.03002524982674 0.759174088369669
Значение функции: -1.94467546413463
Вектор аргументов: -1.44708926856848 1.60030805580332
Значение функции: -4.62949418968186
Вектор аргументов: -1.46613692001725 1.41936480133765
Значение функции: -4.97304424548323
```

Один из пяти вызовов дал минимальное значение функции. Отметим вектор x(-1.47, 1.42) и значение f=-4.97.

Есть похожий вектор, дающий несколько большее значение – отбросим его.

Вектор (-1.03, 0.76), возможно дает какой-то локальный экстремум; во всяком случае, пока отбрасывать его не станем, поскольку его значения достаточно далеко находятся от (-1.47, 1.42).

Вектор (1.27, -1.64) дает еще один минимум y=-4.06.

Вектор (1.28, -0.89) можно оставить, можно отбросить. Допустим, отбросим его.

Теперь полезно повторить поиск, сформировав границы в окрестностях оставленных векторов.

Для первого вектора примем границы a(-2, 1), b(-1, 2). Множественные вызовы MKSearch возвращают примерно одинаковый результат

```
Вектор аргументов: -1.42195035024637 1.42478648686073
Значение функции: -4.99860143398751
```

Примем за начальную точку вектор (-1.42, 1.42) со значением функции -5.

Для второго вектора примем границы a(-1.7, 0.5), b(-1.3, 0.9). Вызовы MKSearch возвращают примерно следующий результат

```
Вектор аргументов: -1.33035008200926 0.897994584589263
Значение функции: -3.14846520672297
```

Это может быть локальный минимум, а может быть промежуточная точка. Без знания дополнительных условий ничего утверждать нельзя.

На основе последнего оставленного вектора примем границы a(1, -2), b(1.5, -1.3). Вызовы MKSearch возвращают примерно одинаковый результат

```
Вектор аргументов: 1.40756581393423 -1.39751597065363
Значение функции: -4.99724190165322
```

Примем за начальную точку вектор (-1.41, 1.4) со значением функции -5. Теперь можно дважды обратьcя к методу HJ и уточнить решение.

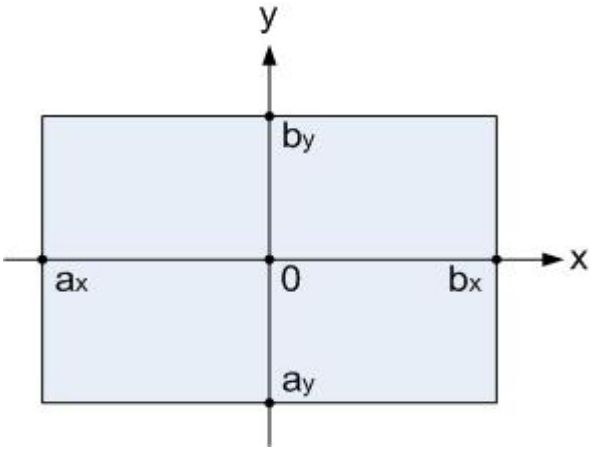
Второй метод, названный BPHS, реализует алгоритм наилучшей пробы с направляющим гиперквадратом.

Алгоритм Монте-Карло имеет два неоспоримых преимущества перед всеми прочими методами оптимизации: он очень прост и абсолютно универсален. Увы, на этом его достоинства заканчиваются. Он ищет всегда лишь один, глобальный на заданном интервале минимум. Да и находит его с очень низкой точностью, если диапазон достаточно широк. Выше мы рассмотрели «шаманство» с этим алгоритмом, когда функция имела два минимума.

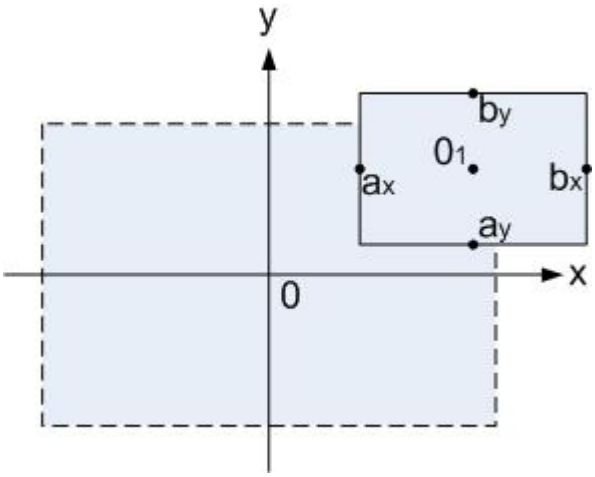
Получается, что широкие границы – это плохо для точности и плохо для нахождения локальных минимумов. Узкие границы – это тоже плохо: минимум на самом деле может оказаться недалеко, но… вне одной из границ. Поэтому хотелось бы иметь метод, который учитывает заданные границы, но в процессе работы их «подвигает», если это оказывается полезным. Одним из алгоритмов, реализующих такое поведение, является алгоритм наилучшей пробы с направляющим гиперквадратом [12].

Фактически, метод BPHS занимается тем же, чем мы занимались, когда многократно вызывали MKSearch: корректирует векторы границ по результатам каждого вызова, стремясь найти минимум функции. Расплатой служит увеличение времени работы, поскольку MKSearch вызывается k раз, делая при вызове m проб. Это немало. Но то, что было «преступлением» на ЭВМ два десятилетия назад, сейчас незаметно: даже при миллионах вычисляемых точек счет идет на секунды.

Принцип работы алгоритма BPHS проще всего понять на примере все того же поиска самой глубокой точки в «яме» - задачи оптимизации функции двух переменных. Пусть мы имеем некое прямоугольное поле, на котором выкопана яма. Пока что мы не знаем, где именно она располагается. Поместим в центр поля начало системы координат, а границы поля зададим значениями координат четырех точек, в которых эти границы пересекают координатные оси. Наборы значений координат a и b образуют исходные векторы начальных и конечных значений.



Применим к выделенному полю метод MKSearch, который даст нам некоторую «наилучшую» точку. Примем эту точку за центр нового прямоугольника, одновременно вдвое уменьшив его границы по каждому измерению



Мы можем повторять этот процесс многократно, например, пока один из линейных размеров прямоугольника не станет равным нулю с машинной точностью. Либо, повторять указанное число раз. Либо использовать какой-то иной критерий. Метод BPHS по умолчанию делает не более k=100 шагов, но при этом также наблюдает за длиной по каждому измерению.

Если обобщить этот алгоритм на случай оптимизации функции n переменных, то он будет работать с некими «гиперпрямоугольниками» в n-мерном пространстве. Почему в названии алгоритма указан гиперквадрат? Наверно, для благозвучия – математики иногда позволяют себе подобные вольности.

Если работу метода своевременно не прервать (т.е. задать большие значения m и k), он найдет минимум с машинной точностью и никакие другие методы не понадобятся. Для большого количества параметров это может быть долгим процессом, поэтому тут важна умеренность. Не самая плохая идея – вовремя остановиться и затем уточнить полученное решение методом HJ (4.9.2.1).

Проиллюстрируем пример работы с методом BPHS для рассмотренной выше функции

$$f(x,y) = x^4 + y^4 - 2x^2 + 4xy - 2y^2 + 3 \to min$$

```
var f:function(x:array of real):real:= x->Power(x[0],4)+
    Power(x[1],4)-2*Sqr(x[0])+4*x[0]*x[1]-2*Sqr(x[1])+3;
var a:=Arr(-20.0,-20.0); // нижние границы
var b:=Arr(20.0,20.0); // верхние границы
var y:real; // значение функции
var oL:=new FMinN(a,f);
oL.BPHS(a,b,y);
Write('Вектор аргументов: '); oL.x.Println;
Writeln('Значение функции: ', y);
```

Результаты пятикратного запуска программы:

```
Вектор аргументов: -1.41421431756713 1.41421404746994
Значение функции: -4.99999999999341
Вектор аргументов: -1.41421356168858 1.41421356425442
Значение функции: -5
Вектор аргументов: 1.414213875863 -1.41421393243357
Значение функции: -4.99999999999811
Вектор аргументов: -1.41421356063545 1.41421355949525
Значение функции: -5
Вектор аргументов: 1.41421356078401 -1.41421356905899
Значение функции: -5
```

Найдены оба экстремума. Полученное решение оказывается даже точнее, чем по методу Хука – Дживса. По времени оно выполняется так же – за доли секунды.

Метод BPHS хорош тем, что он всегда завершается и всегда находит некоторый экстремум в окрестностях первоначально заданных границ. Но если экстремум не один и первоначальные границы определены неудачно, результат может неприятно удивить.

Третий метод, названный BestP, позволяет освободиться от рутинной работы по многократным вызовам BPHS и последующему сравнению результатов каждого вызова. Он производит «p» обращений к BPHS, сохраняет результаты каждого вызова, а потом сравнивает их, выбирая из схожих лучшие. Критерий сравнения – близость значений всего набора параметров в каждой паре проб, который регулируется величиной абсолютной погрешности eps. По умолчанию принято eps=0.01. Задание высокой точности увеличивает количество результатов, признаваемых различными. При чересчур низкой точности часть локальных минимумов может быть потеряна.

Работа метода BestP показана на примере поиска минимума функции, приведенной выше.

```
var f:function(x:array of real):real:= x->Power(x[0],4)+
    Power(x[1],4)-2*Sqr(x[0])+4*x[0]*x[1]-2*Sqr(x[1])+3;
var a:=Arr(-20.0,-20.0); // нижние границы
var b:=Arr(20.0,20.0); // верхние границы
var x:=new real[a.Length]; // для конструктора MinHJ
```

```
var oL:=new FMinN(x,f);
var r:=oL.BestP(a,b,0.01);
var y:real;
var fet:=f(Arr(Sqrt(2),-Sqrt(2)));
foreach var t in r do begin
 (y,x):=(t[0],t[1]);
 Write('Вектор аргументов: '); x.Println;
 Write('Абс.погрешности: ');
 x.Foreach(z->WriteFormat('{0:0.0e0} ',Abs(z)-Sqrt(2)));
 Writeln;
 Writeln('Значение функции: ', y, ', абс.погрешность: ',Abs(y-fet));
 Writeln
 end
```

Результаты

```
Вектор аргументов: -1.41421355926384 1.41421356822497
Абс.погрешности: -3.1e-9 5.9e-9
Значение функции: -5, абс.погрешность: 0

Вектор аргументов: 1.41421356623548 -1.41421356244106
Абс.погрешности: 3.9e-9 6.8e-11
Значение функции: -5, абс.погрешность: 0
```

Результаты в данном случае оказались превосходными. Найдены оба минимума, точность по аргументам не хуже восьми знаков, точность по функции в пределах машинной точности.

Следует отметить что метод BestP возвращает список List, элементами которого являются кортежи <(real, array of real)>. В приведенном примере использован цикл foreach, в котором каждый элемент списка List, представляющий собой решение, распаковывается в переменную y (значение функции) и массив x (вектор аргументов) оператором (y,x):=(t[0],t[1]).

Четвертый метод (ARS), написанный автором на основе алгоритма, приведенного в [13], реализует адаптивный случайный поиск.

Как и в алгоритме Хука – Дживса, приближение ведется из заданной точки. Каждая последующая точка ищется в пределах гиперсферы (вспоминаем гиперкуб алгоритма BPHS), центр которой отстоит от предыдущей точки на некоторый шаг, сделанный в случайном направлении. Если этот шаг улучшает решение, в найденном направлении делается следующий шаг увеличенной длины. В противном случае делается шаг уменьшенной длины. Если при этом решение не улучшается, осуществляется возврат в предыдущую точку для поиска нового направления.

Неплохая альтернатива алгоритму Хука – Дживса. Также может использоваться для повышения надежности решения (задача решается обоими методами и результаты сравниваются), либо может служить первым шагом решения с последующим уточнением.

Пример работы с методом ARS для минимизации функции

$$F(x,y) = 4(x-5)^2 + (y-6)^2$$

```
uses NumLibABC;

begin
  var f:function(x:array of real):real:=
    x->4*Sqr(x[0]-5)+Sqr(x[1]-6);
  var x:=Arr(-8.0,9.0);
  var (t,R):=(1.0,1e-6);
  var oL:=new FMinN(x,f);
  oL.ARS(R,t);
  Write('Аргументы: '); oL.x.Println;
  Writeln('Значение функции: ',f(oL.x))
```

end.

Результат

```
Аргументы: 4.99999981501099 5.99999980292426
Значение функции: 1.75722587828616E-13
```

4.9.2.3. Целевые функции с ограничениями

Описанные выше методы рассматривались для случаев отыскания безусловных экстремумов, что обычно представляет абстрактный интерес. В реальных задачах всегда имеются некоторые ограничения, например, на соотношение между параметрами, диапазон значений целевой функции. К счастью, все предлагаемые методы можно использовать и для таких задач, для чего необходимо определенным образом составить целевую функцию. Выше уже упоминались штрафные функции – теперь уместно познакомиться с ними поближе.

Пусть требуется найти максимум функции F(x,y)=4x+3y-1 со следующими ограничениями

$$\begin{cases} x + y \le 8 \\ 3y - 2x \le 9 \\ 2x - y \le 10 \\ x, y \in \{0, \mathbb{N}\} \end{cases}$$

Здесь ищется не минимум, а максимум функции F, на взаимосвязь параметров при помощи неравенств наложены три ограничения, а сами параметры должны быть целыми неотрицательными числами.

Такие задачи можно часто встретить, например, в экономике и их принято решать симплекс – методом. Но, во-первых, у нас нет

готовой программы, реализующей симплекс-метод, а во-вторых требуется показать, что методы класса FMinN достаточно универсальны и успешно справляются с подобными задачами, пусть делают они это не оптимальным образом. Тут у пользователя всегда есть выбор: использовать неоптимальный, но имеющийся под руками инструмент, или тратить время в поисках лучших инструментов, возможно, сэкономив этим секунду процессорного времени. Вспоминаем народную пословицу «За морем телушка – полушка, да рубль перевоз» и возвращаемся к решению задачи подручными методами.

Поменяв знак функции, приходим к обычной задаче поиска минимума

$$f(x, y) = -4x - 3y + 1 \rightarrow min$$

Конечно, было бы отлично подавать целевой функции сразу целые числа, но… Все методы работают с вектором вещественных параметров, так что формировать целые числа из вещественных придется или в целевой функции, или уже по результатам поиска. Остановимся на последнем варианте.

Если бы не было системы ограничений, заданной тремя неравенствами, на этом обсуждение целевой функции можно было счесть законченным.

Введем вспомогательную функцию, выдающую штраф за нарушение любого из ограничений. Поскольку мы ищем минимум, в качестве штрафа можно взять некое большое значение, например,

машинное «плюс бесконечность». Если нарушений нет, штраф равен нулю. Сумма значения штрафной функции с вычисленным значением f(x,y) образует результирующую целевую функцию.

Полная программа решения задачи

```
uses NumLibABC;

function f(x:array of real):real;
begin
  var x1:=x[0];
  var x2:=x[1];
  var s:=0.0; // штрафная функция
  if x1+x2>8 then s:=real.MaxValue
  else if -2*x1+3*x2>9 then s:=real.MaxValue
  else if 2*x1-x2>10 then s:=real.MaxValue
  else if x1<0 then s:=real.MaxValue
  else if x2<0 then s:=real.MaxValue;
  Result:=-4*x1-3*x2+1+s
end;

begin
  var a:=Arr(0.0,0.0);
  var b:=Arr(8.0,8.0);
  var y:real;
  var oL:=new FMinN(a,f);
  oL.MKSearch(a,b,y);
  oL.x.Transform(t->real(Round(t)));
  Write('Полученные значения аргументов: '); oL.x.Println;
  Writeln('Полученное значение функции: ',f(oL.x))
end.
```

Результаты

```
Полученные значения аргументов: 6 2
Полученное значение функции: -29
```

Литература

1. Дж.Форсайт, М.Малькольм, К.Моулер. Машинные методы математических вычислений. Пер. с англ. – М.: Мир, 1980.

2. System/360 Scientific Subroutine Package (360A-CM-03X) Programmer's Manual: IBM, Technical Publication Department, 112 East Post Road, White Plains, , N.Y. 10601

3. IMSL® Fortran Math Library. Version 7.1.0. Rogue Wave Software, Inc. 5500:Flatiron Parkway, Boulder, CO 80301 USA

4. Агеев М.И., Алик В.П., Марков Ю.И. Библиотека алгоритмов 1б-50б. (Справочное пособие). М., "Сов.радио", 1975

5. Агеев М.И., Алик В.П., Марков Ю.И. Библиотека алгоритмов 51б-100б. (Справочное пособие). Вып.2. М., "Сов.радио", 1976

6. Мудров А.Е. Численные методы для ПЭВМ на языках Бейсик, Фортран и Паскаль. – Томск: МП «РАСКО», 1991.

7. Шуп Т. Решение инженерных задач на ЭВМ: Практическое руководство. Пер. с англ. – М.: Мир, 1982.

8. Р.В. Хемминг. Численные методы. М., «Наука», 1968.

9. Dept. of Scientific Computing, Florida State University, RKF45 Runge-Kutta-Fehlberg ODE Solver, Fortran90 version by John Burkardt, https://www.sc.fsu.edu

10. Dept. of Scientific Computing, Florida State University, LOCAL_MIN_RC, Reverse Communication Function for Local Minimum, by Richard Brent, https://www.sc.fsu.edu

11. Richard Brent, Algorithms for Minimization Without Derivatives, Dover, 2002, ISBN: 0-486-41998-3, LC: QA402.5.B74.

12. Рейзлин В.И. Численные методы оптимизации: учебное пособие / В.И. Рейзлин; Томский политехнический университет. - Томск: Изд-во Томского политехнического университета, 2011.

13. Методы оптимизации в примерах и задачах: Учеб. пособие/А.В. Пантелеев, Т.А. Летова. — 2-е изд., исправл. — М. Высш. шк., 2005.