Библиотека численных методов StudLib для PascalABC.Net 3.2

Ростов-на-Дону
2017

Описывается состав и даются рекомендации по использованию библиотеки численных методов StudLib, реализованной в среде программирования PascalABC.NET 3.2.

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

Оглавление

1. Введение

StudLib – свободно распространяемая библиотека (далее - пакет) программ, реализованная в системе программирования PascalABC.NET 3.2 и поставляющаяся вместе с ней в исходном коде (файл StudLib.pas). Для работы с пакетом модуль StudLib.pas следует загрузить и откомпилировать.

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

С помощью пакета StudLib можно решать задачи из следующих областей:

— нахождение корней нелинейных уравнений;
— статистическая обработка данных, заданных в табличном виде;
— интерполяция, дифференцирование и аппроксимация данных, заданных в табличном виде;
— операции с полиномами;
— линейная алгебра (операции с векторами и матрицами, решение систем линейных уравнений);
— решение систем дифференциальных уравнений;
— вычисление определенных интегралов;
— поиск минимума функций одного и многих переменных;
— 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.

2. Система тестирования StudLibTest

После установки или обновления PascalABC.NET 3.2 рекомендуется выполнить тестирование пакета при помощи модуля StudLibTest.pas.

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 равное -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 StudLib.

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

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 ($7FFFFFFFFFFFFFFF_{16}$), что и ожидалось.

3.2. Работа с пакетом

Для подключения пакета следует использовать секцию раздела uses:

```
uses StudLib;
```

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

```
var oL := new имя_класса(параметры);
```

В некоторых случаях этого уже достаточно чтобы воспользоваться результатами, в других – вызывается какой-либо метод класса, чаще всего Value, возвращаюший результаты.

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

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)

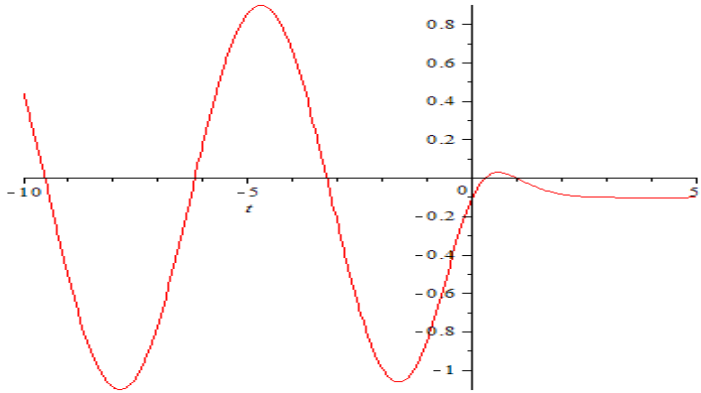
Может применяться для получения интервалов изоляции произвольного количества корней произвольной функции одной переменной. Заданный интервал просмотра [a;b] последовательно просматривается с шагом h. Если корень уравнения попадает на границу интервала b, интервал изоляции выбирается равным [b-h/2;b+h/2]. Правильный выбор шага h имеет важное значение.

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

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

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)] и мы 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 3.2.

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 StudLib;
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
```

Получаем 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. Статистическая обработка данных, заданных в табличном виде

Материал этого раздела предполагает, что пользователь знаком с основами статистики.

4.3. Интерполяция, дифференцирование и аппроксимация данных, заданных в табличном виде

4.3.1. Интерполяция табличной функции кубическим сплайном (класс 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.

Конечно, можно попытаться провести через n точек полином степени n-1, но теория доказывает, что практически это оказывается очень плохим решением для n>10.

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 перемещая 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.

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

$y' = F'(x) = 3ax^2 + 2bx + c,$ 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. Полиномы с действительными коэффициентами

4.4.1. Описание класса *Polynom*

Обычно полином u(x) принято записывать в порядке убывания степеней, т.е. в виде $u(x) = u_n x^n + \ldots + u_1 x + u_0$

В то же время, большинство алгоритмов для работы с полиномами используют обратный порядок записи – в порядке возрастания степеней: $u(x) = u_0 + u_1 x + \ldots + u_n x^n$

Именно такой порядок следования коэффициентов полинома используется, например, в пакетах программ SSPLIB фирмы IBM [3] и IMSL® фирмы Rogue Software [4]. Исходя из изложенного выше, было принято использовать такой же порядок следования и в пакете StudLib.

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.

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 4.4.2).

Перечислим методы класса *Polynom*:

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

Copy – создает копию объекта класса *Polynom*;

PDif – возвращает коэффициенты полинома, полученного дифференцированием исходного;

PInt – возвращает коэффициенты полинома, полученного при взятии неопределенного интеграла от исходного;
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;
Будет получен результат
-0.4x^5-0.75x^4+4x^3-3.5x^2+x
-8x^3-9x^2+24x-7
Запишем результаты в стандартной форме:
$p(x) = -0.4x^5 - 0.75x^4 + 4x^3 - 3.5x^2 + x + C,$
$q(x) = -8x^3 - 9x^2 + 24x - 7$

4.4.2. 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;
```

Будет получен ответ

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;
```

Результат

```
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
```

4.4.3. Корни полиномов. Операции с полиномами. Полиномиальная аппроксимация.

Из основной теоремы алгебры следует, что если полином u(x) имеет степень n, то уравнение u(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.

4.4.3.1. Нахождение всех корней полинома с действительными коэффициентами методом Ньютона – Рафсона (PolRt)

Класс PolRt выполнен на основе подпрограммы DPOLRT [3], 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 также, 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 получим 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);
```

Был получен результат (-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:

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

В то же время, если нужно разложить полином на рациональные линейные множители, удобнее воспользоваться классом Factors, описанным ниже.

4.4.3.2. 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 разложении. 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;
```

Результат:

```
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 понятно: к 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;
```

Результат:

```
k:=1
1 3
2 -5
```

Тут в разложении 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 (4.2.2):

```
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
```

Результат:

```
3x^2+x-3
0
```

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

А нельзя ли было не мудрить и сразу использовать для разложения полинома P(x) метод PolRt из 4.4.3.1? Попробуем.

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

Результат

(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 вида $(x + 2.5)(x - 3) = 0.5(2x + 5)(x - 3)$

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

4.4.4. Специальные полиномы

4.5. Линейная алгебра (операции с векторами и матрицами, решение систем линейных уравнений)

4.6. Решение систем дифференциальных уравнений

4.7. 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 комби-

нацию величины интервалов разбиения исходного отрезка и варианта квадратурной формулы так, чтобы обеспечивать необходимую точность.

4.7.1. Адаптивная квадратурная программа Quanc8

В пакет входит класс Quanc8, полученный переработкой для PascalABC.NET 3.2 подпрограммы QUANC8 [1], написанной на языке Fortran.

При создании класса 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 3.2 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);

4.8. Поиск минимума функций

4.9. Задачи оптимизации

Литература

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

2. Сборник научных программ на Фортране. Вып.1. Статистика. Нью-Йорк, 1960-1970, пер. с англ. (США). М., «Статистика», 1974.

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

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

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.