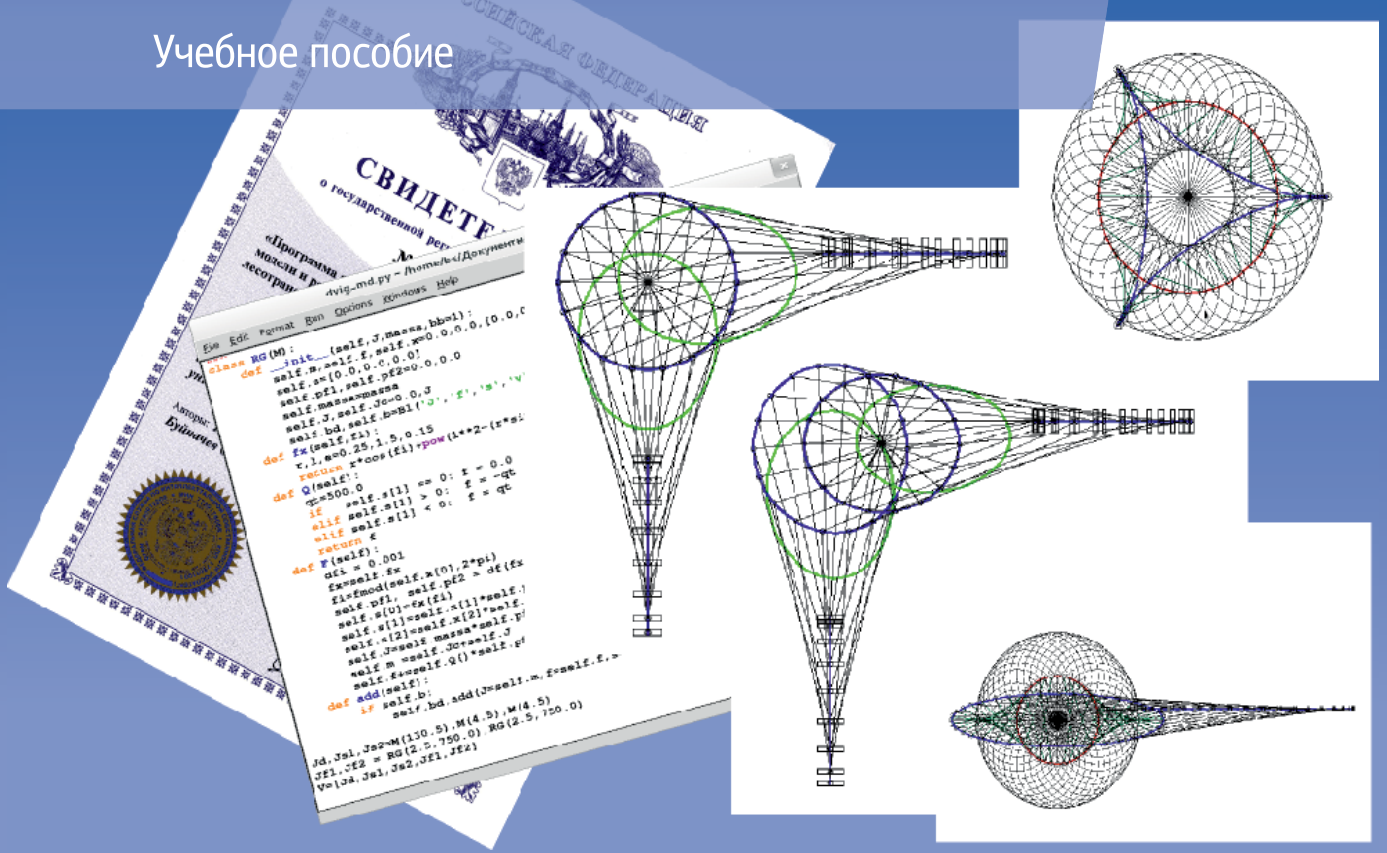
Уральский федеральный университет
имени первого Президента России Б.Н.Ельцина
Механико-машиностроительный институт

С. К. БУЙНАЧЕВ

ПРИМЕНЕНИЕ ЧИСЛЕННЫХ МЕТОДОВ В МАТЕМАТИЧЕСКОМ МОДЕЛИРОВАНИИ

Учебное пособие



Министерство образования и науки Российской Федерации
Уральский федеральный университет
имени первого Президента России Б. Н. Ельцина

С. К. Буйначев

ПРИМЕНЕНИЕ ЧИСЛЕННЫХ МЕТОДОВ В МАТЕМАТИЧЕСКОМ МОДЕЛИРОВАНИИ

Рекомендовано методическим советом УрФУ в качестве ***учебного пособия*** *для студентов, обучающихся по программе специалитета по направлению 151000 «Технологические машины и оборудование» и бакалавриата по направлению 151000.62 «Технологические машины и оборудование»*

Екатеринбург
Издательство Уральского университета
2014

УДК 519.6(035+06)
ББК 22.19я7
Б90

Рецензенты: доц., канд. техн. наук В. П. Подогов (Российский государственный профессионально-педагогический университет);

доц., канд. техн. наук Е. Е. Баженов (Уральский государственный экономический университет)

Научный редактор доц., канд. техн. наук Ю. В. Песин

Буйначев, С. К.

Б90 Применение численных методов в математическом моделировании : учебное пособие / С. К. Буйначев. – Екатеринбург: Издательство Уральского университета, 2014. – 70, [2] с.

ISBN 978-5-7996-1197-2

Учебное пособие содержит сведения о численных методах. Наибольшее внимание уделяется последовательности вычислений и их программированию на языке программирования Python.

Пособие может быть рекомендовано студентам различных специальностей технических вузов, занимающихся математическим моделированием и изучающим численные методы, служить справочным материалом при выполнении курсовых и дипломных работ, связанных с расчетами на компьютере. Также может быть использовано преподавателями, аспирантами и научными сотрудниками.

Библиогр.: 7 назв. Рис. 11.

УДК 519.6(035+06)
ББК 22.19я7

ISBN 978-5-7996-1197-2

ВВЕДЕНИЕ

Математика как наука возникла в связи с необходимостью решения различных практических задач — вычисления каких-либо параметров изучаемого явления. Для этого создается математическое описание объекта. Математическое описание называется математической моделью объекта, т. е. объект заменяется математической моделью, которая описывает необходимые изучаемые свойства этого объекта. Математическая модель состоит из уравнений и дополнительных условий. Классическая математика занимается получением аналитических методов решения математических моделей, отдельных уравнений или систем уравнений. Аналитическое решение представляет собой формулу или ряд формул, по которым можно вычислить необходимые параметры. Преимущества аналитического решения:

1. Обычно для получения результата требуется небольшое количество вычислений и можно обойтись без компьютера.
2. Заранее известен диапазон, в котором справедливо данное решение.
3. Так как решение представлено в виде формул, то можно проанализировать влияние различных коэффициентов и параметров на решение.

Поэтому во всех случаях, когда оно не очень громоздко, рекомендуется применять аналитическое решение задачи. К сожалению, очень небольшое количество уравнений и систем уравнений, описывающих реальные процессы, удается решить аналитически. Здесь на помощь приходят численные методы, которые начали развивать еще Ньютон, Эйлер, Лобачевский, Гаусс. Быстрое развитие численные методы получили с появлением ЭВМ. Собственно говоря, ЭВМ появились в результате потребностей вычислительной математики в большом количестве вычислений для получения численного решения.

Распространенное мнение о всемогуществе современных компьютеров часто порождает впечатление, что математики избавились почти от всех хлопот, связанных с численным решением задач, и разработка новых методов не столь существенна. Это ошибочное представление. И в настоящее время есть задачи, которые имеют достаточно хорошее математическое описание, но до сих пор не решены в общем виде. Например, это задачи обтекания трехмерного тела потоком сжимаемого газа (движение самолета, автомобиля и т. п.); задачи, описывающие процесс ядерного взрыва; задачи, связанные с предсказанием погоды, и многие другие сложные процессы.

Решения, полученные численными методами, представляют собой числа или ряд чисел, в отличие от формул, полученных при аналитических решениях. Это является недостатком численных решений, т. к. их трудно исследовать. Достоинством численных методов является то, что метод решения не зависит от вида уравнений для того класса уравнений, для которого данный метод может быть применен. Например, один и тот же метод может быть применен и для алгебраических, и для тригонометрических уравнений.

Решения, полученные численными методами, являются приближенными, поэтому необходимо оценить погрешность, с которой получен результат. А это, в свою очередь, зависит от ресурсов вычислительной техники и качества программного обеспечения.

Методология ‘Extreme Programming’

Подробное обсуждение *методологии экстремального программирования* (extreme_programming), или сокращенно XP, можно найти на сайтах http://www.extremeprogramming.org и http://www.xprogramming.org. Если коротко, то в основе этой новой парадигмы программирования лежит идея простоты и общения. Методика ориентирована на небольшие коллективы программистов, которые должны быстро создать продукт в условиях постоянно изменяющихся спецификаций. Если не усложнять программу и поддерживать постоянную связь между всеми членами коллектива, а также между программистами и заказчиком, то приемлемый продукт можно будет получить в короткие сроки и с меньшими затратами.

Один из действительно новых аспектов XP заключается в том, что приверженцы этой методики начинают с написания тестов для всесторонней проверки продукта еще до того, как он создан. Затем разрабатывается продукт, который прогоняется через все тесты. По мере обнаружения ошибок добавляются новые тесты, и в код вносятся исправления. Таким образом, тесты пишутся до, во время и после создания продукта, т. е. тестирование продолжается на протяжении всего цикла разработки. Иными словами, к моменту появления готового продукта он уже протестирован.

Акцент делается на идею команды, в которую входят менеджеры, заказчики и программисты. Все заинтересованные стороны принимают участие в каждом этапе процесса разработки. Во многих традиционных моделях процесса разработки ПО заказчик предостав-

ляет требования, а программисты создают на их основе готовый продукт. Когда заказчик получит продукт и увидит его в действии, он может с сожалением подумать о том, что стоило бы попросить в самом начале о конкретизации каких-либо свойств программы, но, увы, будет уже поздно. В XP заказчик участвует во всех этапах разработки и может предлагать дополнения и изменения к проекту, хотя он уже реализуется. Поэтому проектные изменения становятся более органичными. В методике XP перечислено 12 принципов, определяющих структуру XP-проекта:

1. *Процесс планирования, иногда называемый игрой в планирование.* Процесс планирования позволяет заказчику XP-проекта определить ценность тех или иных функций для своего бизнеса и, руководствуясь полученными от программистов оценками трудозатрат, решить, что необходимо реализовать сразу, а что можно отложить. В результате проект легче довести до успешного завершения.

2. *Частый выпуск версий.* XP предполагает ранний выпуск простой версии работающей системы и частые обновления.

3. *Метафора.* В XP используется общая система имен и единое описание системы, что упрощает разработку и общение между членами команды.

4. *Простота проекта.* Программа, созданная с применением XP, должна быть максимально простой и в то же время отвечать текущим требованиям. «Задумки на будущее» практически отсутствуют, уступая место соображениям ценности для заказчика. Разумеется, это не отменяет необходимости иметь хороший проект, для чего в XP применяется идея реорганизации (refactoring), о которой говорится далее.

5. *Тестирование.* Акцент в XP ставится на проверку правильности программы на всех этапах разработки. Программисты сначала пишут тесты, а уже затем собственно программу, которая отвечает требованиям, выраженным в виде тестов. Заказчики предоставляют приемосдаточные тесты, которые позволяют убедиться в наличии всех необходимых им функций.

6. *Реорганизация.* В XP проект системы совершенствуется на протяжении всего цикла разработки. Это позволяет вычищать программу: устранять дублирование, не жертвуя полнотой.

7. *Программирование парами.* Программисты, работающие в рамках методологии XP, пишут весь код парами, то есть на одной машине совместно трудятся два специалиста. Экспериментально доказано, что при этом удается получить более качественную программу с такими же или даже меньшими затратами, чем при работе поодиночке.

8. *Коллективное владение.* Весь код принадлежит всем программистам. Это позволяет ускорить работу, так как любое изменение реализуется без задержки.

9. *Непрерывная интеграция.* Команда XP интегрирует и собирает новые версии системы несколько раз на дню. Это дает возможность всем программистам точно знать текущее состояние дел и добиваться очень быстрого продвижения по пути к конечной цели. Как это ни удивительно, но частая интеграция позволяет устранить проблемы, преследующие коллективы, которые собирают полную систему существенно реже.

10. *Ограниченная рабочая неделя.* Усталый программист делает больше ошибок. В командах XP не практикуется сверхурочная работа, поэтому все их члены, как правило, трудятся максимально эффективно.

11. *Участие заказчика.* Проект XP управляется специально выделенным для этой цели человеком, задача которого – определять требования, расставлять приоритеты и отвечать на вопросы программистов. В результате повышается эффективность общения, причем без написания бумажных документов, что зачастую оказывается самой дорогостоящей составляющей программного проекта.

12. *Стандарты кодирования.* Чтобы работа парами была эффективной, а совместное владение кодом имело смысл, все программисты должны следовать единым правилам написания кода, тогда ни у кого не возникнет трудностей при чтении текста, написанного разными членами команды.

1. ОБЩИЕ СВЕДЕНИЯ

1.1. Погрешность вычислений

Точность результатов является основным критерием качества вычислений. Рассмотрим следующие источники погрешностей:

1. Математическая модель никогда не отражает описываемый объект полностью. Возможно, некоторые данные, параметры математической модели, имеют погрешность. Погрешность математической модели называется неустранимой погрешностью.

2. Погрешность численного метода связана с достаточно большим числом вычислительных операции. Например, вычисление интеграла предполагает вычисление суммы ряда

$$S = 1 + 1/n + 1/n^2 + 1/n^3 + 1/n^4 + \ldots \; (|n|>1)$$

и ограничение количества слагаемых.

3. Вычислительная погрешность — это погрешность округления чисел в компьютере из-за ограничения разрядов процессора.

Рассмотрим подробнее указанные виды погрешностей:

1. Представление о требуемой точности результатов математической модели позволяет производить уточнения или упрощения в математическом описании задачи. Необходимо все исходные данные представить примерно одинаковой точности. Очевидно, что уточнение одних исходных данных при наличии больших погрешностей в других данных не приводит к повышению точности результатов.

2. Погрешность численного метода регулируема. Она может быть уменьшена до разумного значения путем изменения шага интегрирования, числа членов усеченного ряда и т. п. Погрешность метода доводят до величины, в несколько раз меньшей погрешности исходных данных (это должна быть величина примерно того же порядка).

3. Погрешность вычислений в компьютере зависит от представления чисел. Числа в компьютере представлены в двух видах: целые и действительные. Целые числа и действия с ними производятся точно. Действительные числа можно записать в форме с плавающей точкой, например:

$$-1535\cdot10^{-1};\ -1{,}535\cdot10^{2};\ -0{,}1535\cdot10^{3}.$$

Последняя запись — нормализованная форма числа с плавающей точкой. Примерно так записываются числа в память компьютера. Значащими цифрами числа называются все цифры в его записи справа начиная с первой ненулевой цифры, например число 0,003045 имеет 4 значащие цифры. Для каждого типа действительных чисел в памяти компьютера выделено определенное число байт. При записи с плавающей точкой воспринимается только определенное количество значащих цифр, остальные отбрасываются. Компьютер оперирует приближенными значениями действительных чисел, однако в языке Python можно задать любое требуемое количество значащих цифр.

Различают два вида погрешностей:

1. Абсолютная погрешность — это разность между точным значением числа x и его приближенным значением a:

$$\Delta x = x - a.$$

2. Относительная погрешность

$$\delta x = \Delta x / |a|.$$

Истинное значение x обычно неизвестно – имеем лишь приближенное значение числа a. Поэтому обычно говорят о предельной погрешности числа a – Δa, которая является верхней границей абсолютной погрешности Δx.

$$| \Delta x | \leqslant | \Delta a |;$$

$$\delta a = \Delta a/|a|.$$

Для приближенного числа, полученного в результате округления, абсолютная погрешность Δa принимается равной половине единицы первого отброшенного разряда числа. При вычислении на компьютере округление не производится, а цифры, не умещающиеся в разрядную сетку, отбрасываются. В этом случае погрешность в два раза больше по сравнению с результатом округления.

Действия над приближенными числами:

$$\Delta(a \pm b) = \Delta a + \Delta b;$$

$$\delta(a \cdot b) = \delta a + \delta b;$$

$$\delta(a/b) = \delta a + \delta b;$$

$$\delta(a^k) = k \cdot \delta a,$$

где k – точное число.

1.2. Устойчивость

Поскольку погрешности исходных данных – это неустранимые погрешности, то они тоже влияют на точность конечных результатов. Будем считать, что погрешность результатов имеет порядок погрешности исходных данных. Но это не всегда так. Задача называется устойчивой по исходному параметру x, если функция $y = f(x)$ непрерывна и монотонна, т. е. малое приращение Δx приводит к малому приращению Δy. Малые погрешности в исходной величине приводят к малым погрешностям в результатах расчетов. Отсутствие устойчивости означает, что даже незначительные погрешности в исходных данных приводят к большим погрешностям или даже к неверному результату.

Пример. Найти действительные корни многочлена

$$(x - a)^n = b,$$

$$N < 1;\ b > 1,$$

где a, n — точные числа; b — приближенное число.

Решение можно представить в виде

$$x = a \pm b^{1/n}.$$

Относительная погрешность решения x

$$\delta x = \delta(b^{1/n}) = \frac{1}{n} \cdot \delta b.$$

При $n = 0{,}1$ относительная погрешность $\delta x = 10\delta b$, т. е. она в 10 раз больше погрешности исходного параметра b.

Иногда для устойчивой по исходным данным задачи может оказаться неустойчивым метод ее решения. Неустойчивый метод решения рассмотрен выше при определении погрешности вычисления суммы ряда

$$S = \Sigma_1^{\,1\,000\,000} (1/n^2).$$

1.3. Корректность

Задача называется поставленной корректно, если для любых значений исходных данных из некоторого класса ее решение существует единственно и устойчиво по исходным данным. Рассмотренная выше неустойчивая задача является некорректно поставленной. Применять для некорректно поставленных задач обычные численные методы нецелесообразно, т. к. на результат сильно влияют погрешности исходных данных. Кроме того, при расчетах появятся погрешности вычисления на компьютере, которые будут возрастать в ходе вычислений, что приведет к неправильным результатам.

В настоящее время существуют методы решения некоторых некорректных задач. Это методы регуляризации. Они основаны на замене исходной корректно поставленной задачи, содержащей некоторый параметр, при стремлении которого к нулю решение этой задачи переходит в решение исходной задачи.

1.4. Сходимость

При анализе вычислительного процесса одним из важнейших критериев является сходимость численного метода. Он означает близость полученного численного решения к истинному.

Рассмотрим понятие сходимости итерационного процесса. Этот процесс состоит в том, что для решения некоторой задачи строится

метод последовательных приближений. В результате многократного повторения этого процесса (или итераций) получаем последовательность значений x_1, x_2, x_3, …, x_n. Эта последовательность сходится к точному решению x, если при неограниченном возрастании числа итераций n предел этой последовательности существует и равен

$$\lim x_n \mid_{n\to\infty} = x.$$

В этом случае имеем сходящийся численный ряд, т. е. решение с заданной точностью можно получить при конечном числе итераций.

Другой подход к понятию сходимости используется в методах дискретизации. Эти методы заключаются в замене задачи с непрерывными параметрами на задачу, в которой значения функций вычисляются в фиксированных точках. Это относится, в частности, к численному интегрированию, к решению дифференциальных уравнений. Здесь под сходимостью метода понимается стремление решения дискретной задачи к решению исходной задачи при стремлении к нулю параметра дискретизации (например, шага интегрирования).

2. РЕШЕНИЕ УРАВНЕНИЙ

Основой численных методов является метод итераций (метод последовательных приближений). Он состоит в следующем. Пусть надо решить уравнение $f(x) = 0$. Зададим некоторое приближенное значение x_1 и вычислим $f(x_1)$. Получим $f(x_1) = \Delta_1$. Зададим новое значение x_2, получим $f(x_2) = \Delta_2$. Но x_2 надо задать так, чтобы $|\Delta_2|$ было меньше $|\Delta_1|$, т. е. на каждой итерации должно соблюдаться условие: модуль значения функции следующего меньше модуля значения функции предыдущего. И тогда на некоторой k-й итерации получим $f(x_k) < e$, где e — заданная точность решения.

```
##Вычисление и график функции
# -*- coding: utf-8 -*-
from Tkinter import *
from bsc_groups import *
from math import*
##=========================================== Function
def f(x): return x**2*sin(x)
##=========================================== Calculating
```

```
bl=Bl('x','y')
x=0.0
xk=2.0*pi
dx=0.01
while x<=xk:
    y=f(x)
    bl.add(x=x,y=y)
    x+=dx
##================================================ Labels
crl=["red","green","#0DF","#C06","#A64","#686","blue","#AC3"]
bg='White'
Lbd =[Bd(500,55,'График',crl[5],bg)]
Lbd+=[Bd(165, 600,"y = [%6.3f  %6.3f]" % (min(bl.L['y']),
max(bl.L['y'])),crl[0],bg)]
Lbd+=[Bd(165, 615,"x = [%6.3f  %6.3f]" % (min(bl.L['x']),
max(bl.L['x'])),crl[1],bg)]
##================================================ Tkinter
A,B=1000,700
tk=Tk()
tk.title('График')
fr=Frame(tk)
fr.pack()
c=Canvas(fr,bg=bg,width=A,height=B)
c.pack(expand=1,fill=BOTH)
#================================================ Output
desine_l(60,130,400,400,(255,255,255),crl,bl.L['x'],[bl.L['y']],0,50,50,c)
for ibd in Lbd: ibd.show©
tk.mainloop()
```

Здесь возникают следующие вопросы:

1. Как задать начальное приближение, чтобы получить решение за конечное число итераций?
2. Как задать значение x на следующей итерации, зная x на предыдущей итерации, и получить итерационную формулу?
3. Как организовать итерационный процесс, чтобы получить решение за минимальное конечное число итераций, обеспечивающих условие сходимости?

Уравнение с одним неизвестным можно представить в виде

$$f(x) = 0,$$

где функция $f(x)$ определена и непрерывна на некотором интервале $[a, b]$. Необходимо найти корни этого уравнения. Число, обращающее функцию $f(x)$ в нуль, называется корнем уравнения. В общем случае аналитическое решение имеет значительно более сложный вид, чем у квадратного и кубического уравнения, или вообще отсутствует. Тогда необходимо использовать численные методы.

2.1. Алгоритм отделения корней

Прежде чем использовать какой-либо численный метод, необходимо сделать отделение корней. Отделение корней — это определение количества корней в интересующей нас области и выделение достаточно малых интервалов, в каждом из которых заключен только один корень. Условием существования корня непрерывной функции на интервале является следующее уравнение: $f(a) \cdot f(b) < 0$, т. е. это означает, что на данном интервале функция изменяет знак, а значит, пересекает ось x.

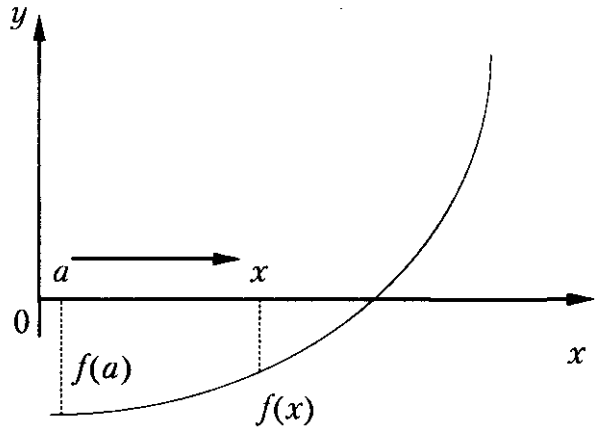


Рис. 1

Рассмотрим алгоритм отделения корней функции $f(x)$ (рис. 1) на интервале $[a, b]$, где a, b — соответственно левый и правый концы интервала; n — количество участков, на которых производится отделение корней.

```
## Определение функции f(x) и отделение корней
## Ввод a,b,n
```

```
dx=(b-a)/n
x1=a
while x1<=b-a:
   x2=x1+dx
   if f(x1)*f(x2):
      L+=[(x1,x2)]
   x1=x2
print 'Список интервалов с корнями', L
```

2.2. Метод дихотомии (деление отрезка на две части)

Пусть необходимо решить уравнение $f(x) = 0$, где функция непрерывна на отрезке $[a; b]$, и единственный корень x заключен в том же интервале:

1. Разделим отрезок $[a; b]$ пополам и найдем $x = (a + b) / 2$.
2. Вычислим значение $f(x)$ в этой точке.
3. Проверим знак условия $f(x) \cdot f(a)$. Если $f(x) \cdot f(a) > 0$, то корень находится на отрезке $[x; b]$.
4. Половина отрезка $[a; x]$ отбрасывается.
5. Левая граница интервала перемещается в точку x, $a = x$ (рис. 2).
6. Если $f(x) \cdot f(a) < 0$, то корень находится на отрезке $[a; x]$ и отбрасывается $[x; b]$.

При повторном делении интервала производятся те же самые операции: новый отрезок $[a; b]$ делится пополам, вычисляется значение функции в точке деления $f(x)$ и определяется отрезок, содержащий корень. Процесс деления продолжается до тех пор, пока длина отрезка $[a; b]$, содержащего корень, не станет меньше некоторого малого заданного числа e.

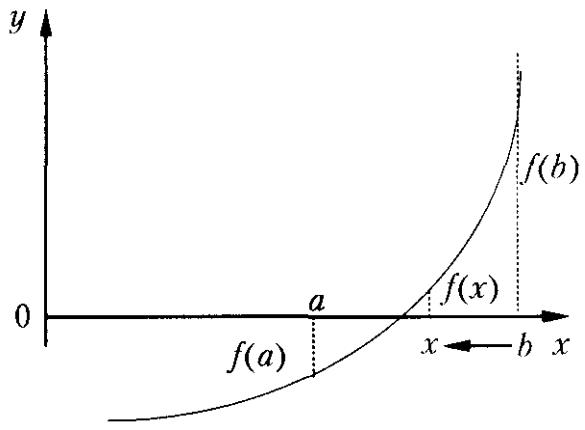


Рис. 2

```
## Метод дихотомии (деление отрезка пополам)
def f(x):   return sin(x)-pow(x,3)

x1,x2,e=0.0,1.0,0.00001
y1,y2=f(x1),f(x2)
while abs(x2-x1)>e:
    x3=(x1+x2)/2.0
    y3=f(x3)
    if y1*y3<0.0:   x2,y2=x3,y3
    else:               x1,y1=x3,y3
    print x3,y3
```

2.3. Метод простых итераций

Использование метода предполагает представление уравнения $f(x) = 0$ в виде $x = \varphi(x)$. Выберем на отрезке $[a; b]$ первое приближение x_1 и подставим его в правую часть уравнения, затем $x_2 = \varphi(x_1)$.

В общем виде можно записать:

$$x_{n+1} = \varphi(x_n).$$

Геометрически способ простых итераций можно представить следующим образом (рис. 3). Построим графики двух функций: $y = x$ и $y = \varphi(x)$. Абсцисса точек пересечения этих двух графиков является корнем уравнения $f(x) = 0$.

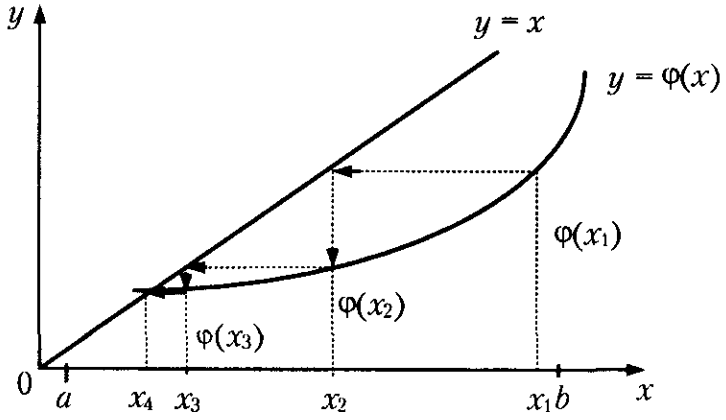


Рис. 3

Процесс итераций сходится при условии $|\varphi'(x)| < 1$. При $|\varphi'(x)| > 1$ процесс может расходиться (рис. 4). Для выбора начального приближения вычисляют значения первых производных функции $\varphi(x)$

в граничных точках интервала [a; b], содержащего корень, и за начальное приближение принимают тот конец интервала, для которого выполняется условие $|\varphi'(x)| < 1$.

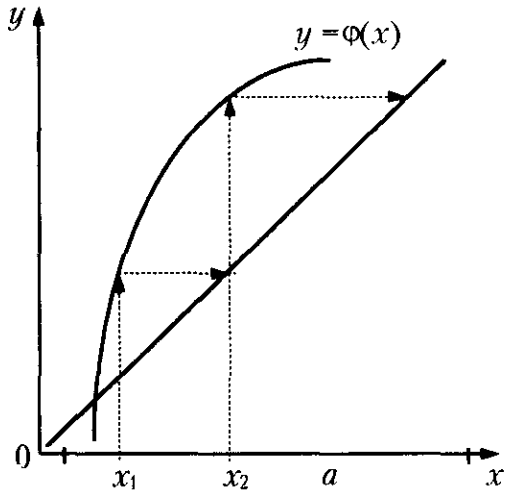


Рис. 4

Если обе производные по абсолютной величине меньше единицы, то за начальное приближение берут ту границу, для которой значение производной по абсолютной величине меньше, тогда процесс итераций сходится быстрее.

```
## Метод простых итераций
# -*- coding: utf-8 -*-
from math import *
def f(x):
    '" функция правой части уравнения вида x=f(x) '"
    pass
y=f(x)
c=abs(x-y)
while c<e:
    y=f(x)
    c=abs(x-y)
    x=y
print x
```

2.4. Метод касательных (метод Ньютона)

Пусть уравнение $f(x) = 0$ имеет один корень на отрезке [a; b], а первая и вторая производные функции $f(x)$ определены, непрерывны и сохраняют постоянные знаки на этом интервале. Геометрическое представление метода – на рис. 5.

Выберем в качестве первого приближения точку $x_1 = b$. Через точку p_1 с координатами $[x_1; f(x_1)]$ проведем касательную к кривой $y = f(x)$. В качестве второго приближения x_2 возьмем абсциссу точки пересечения этой касательной с осью x. Через точку p_2 снова проведем касательную, пересечение которой с абсциссой дает следующее приближение x_3 и т. д.

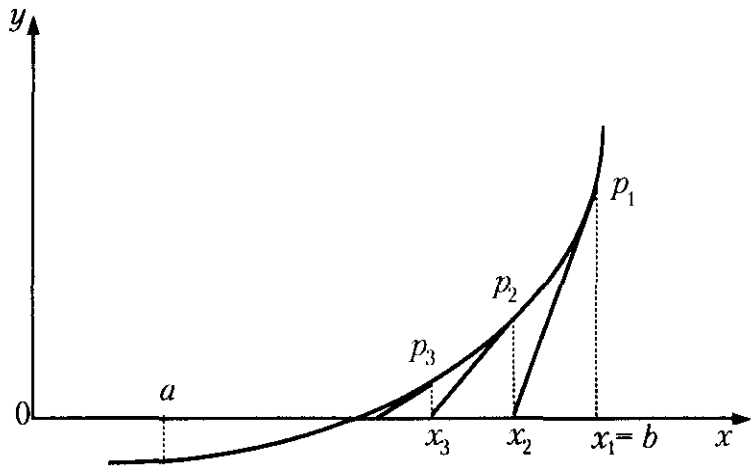


Рис. 5

Уравнение касательной, проходящей через точку x_1, имеет вид

$$y - f(x_1) = f'(x_1) \cdot (x - x_1),$$

уравнение касательной, проходящей через точку x_2, $x_2 = x_1 - f(x_1)/f'(x_1)$. В общем случае можно записать рекуррентную формулу:

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

Если в качестве первого приближения взять $x_1 = a$ и провести касательную к функции $f(x)$ в точке p_a, то она пересечет ось x вне отрезка $[a; b]$. В качестве первого приближения выбирается тот конец интервала $[a; b]$, для которого выполняется условие $f(x) \cdot f''(x) > 0$, где $f''(x)$ — вторая производная. Это условие сходимости метода является достаточным, но не необходимым: если условие выполняется, то итерационный процесс обязательно сойдется, а если не выполняется, то может сойтись, а может и не сойтись.

```
# Метод касательных
from math import *

def f(x):
    """ Функция, определяющая уравнение """
    pass
def df(x):
    """Производная функции f(x)"""
    pass
```

```
x,e=b,0.0000001
xx=x2
while 1:
    if df(x)<>0:
        x=x-f(x)/df(x)
    else:
        print 'Error'
        break
    if abs(x-xx)<e:
        print 'x=%10.6f' % x
        break
    xx=x
```

2.5. Метод хорд

Пусть уравнение $f(x) = 0$ имеет один корень на отрезке $[a; b]$, а первая и вторая производные функции $f(x)$ определены, непрерывны и сохраняют постоянные знаки на этом интервале. Рассмотрим геометрическое представление метода (рис. 6).

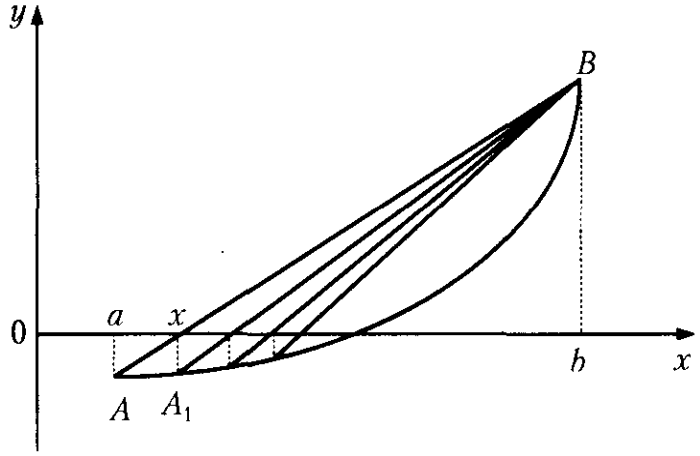


Рис. 6

Проводим хорду через точки A и B. В точке пересечения хорды с осью x находим значение функции $f(x)$, получаем точку A_1, затем проводим новую хорду через точки A_1 и B и т. д.

Уравнение прямой, проходящей через две точки (x_1, y_1) и (x_2, y_2), имеет вид

$$(x - x_1)/(x_1 - x_2) = (y - y_1)/(y_1 - y_2). \qquad (1)$$

Из этого уравнения надо определить точку x – точку пересечения хорды с осью x. Значит, в уравнении (1) $y = 0$. Один из концов отрезка $[a; b]$ является подвижным (в нашем случае a), другой конец – неподвижным. В качестве подвижного конца выбирается точка, для которой выполняется условие $f(x) \cdot f''(x) < 0$.

```
##Метод хорд
# -*- coding: utf-8 -*-
from math import *

def f(x): return sin(x)-pow(x,3)

x1,x2,e=0.01,1.0,0.0000001
y1,y2=f(x1),f(x2)
print x1,y1,x2,y2
xx=x2
while 1:
    x3=(abs(y2)*x1+abs(y1)*x2)/(abs(y2)+abs(y1))
    y3=f(x3)
    if y1*y3<0: x2,y2=x3,y3
    else:       x1,y1=x3,y3
    if abs(x3-xx)<e: break
    xx=x3
    print x3,y3
```

3. СРАВНЕНИЕ МЕТОДОВ РЕШЕНИЯ УРАВНЕНИЙ

Методы решения уравнений сравнивают по двум параметрам:

1. Сложность алгоритма.
2. Время решения уравнения на компьютере.

Если сравнивать сложность алгоритмов рассмотренных выше методов, то все они достаточно просты. Здесь можно выделить метод половинного деления, т. к. он всегда сходится, если функция непрерывна и имеет корень на рассматриваемом интервале. Это удобно, поскольку не надо исследовать функцию и выбирать первое приближение для x. Но метод половинного деления требует большего количества итераций, чем другие методы.

Время решения уравнений зависит от количества итераций и времени, затрачиваемого на одну итерацию. Время одной итерации зависит от того, сколько раз вычисляется функция и (если это требуется) ее производная на одной итерации. Во всех рассмотренных алгоритмах функция на каждой итерации вычисляется один раз, но в методе Ньютона необходимо вычислить еще и производную функции. Если сравнивать количество итераций, то все зависит от вида функции. В большинстве случаев меньше всего итераций требует метод Ньютона. И хотя он требует вычисления производной на каждой итерации, время решения по этому методу обычно меньше, чем для других методов. Недостатком метода является то, что не всегда удается получить аналитическое выражение для производной. Кроме того, в итерационной формуле производная находится в знаменателе и возможны случаи равенства нулю производной на некотором шаге. Число итераций для оставшихся двух методов зависит от вида функции.

4. АППРОКСИМАЦИЯ

Пусть в точках $x_1, x_2, ..., x_n$ известны значения функции $f(x)$ – это $y_1, y_2, ..., y_n$. Необходимо определить значения функции $f(x)$ при любых других значениях x. Связь $y = f(x)$ неизвестна. Это задача точечной аппроксимации.

Если зависимость $y = f(x)$ известна, то она имеет очень сложный вид или требует громоздких вычислений. Тогда вместо непосредственного вычисления функции во многих точках целесообразно заранее вычислить ее значения в отдельных точках, а в других точках вычислять по каким-либо простым формулам, т. е. решение данной задачи сводится к предыдущей. Это задача непрерывной аппроксимации.

Функцию $f(x)$ требуется приближенно заменить (аппроксимировать) некоторой функцией $\varphi(x)$ так, чтобы отклонение $\varphi(x)$ от $f(x)$ в заданной области было наименьшим. Функция $\varphi(x)$ при этом называется аппроксимирующей. На практике чаще всего применяется аппроксимация многочленами:

$$\varphi(x) = a_0x^m + a_1x^{m-1} + a_2x^{m-2} + \ldots + a_m.$$

Всякую функцию можно аппроксимировать многочленом, а значение рассчитать по методу Горнера. Это ускоряет вычисления.

Теорема Вейерштрасса. Если функция $f(x)$ непрерывна на отрезке $[a; b]$, то для любого $e > 0$ существует многочлен $\varphi(x)$ степени m с коэффициентами

$$a = [a_0 , a_1 , a_2 , \ldots, a_m],$$

абсолютное отклонение которого от функции $f(x)$ на отрезке $[a; b]$ меньше e.

Теорема говорит о том, что любую функцию можно как угодно точно аппроксимировать многочленом, но она ничего не говорит ни о способах нахождения этого многочлена, ни о количестве точек, ни об их расположении. Многочлены не являются единственным способом аппроксимации. Иногда удобно использовать тригонометрические, логарифмические и другие функции. Определить аппроксимирующую функцию – это значит задать вид функции и найти ее пара-метры. При аппроксимации отклонение значений полинома $\varphi(x)$ и функции $f(x)$ при любом количестве точек должно быть минимальным.

```
# -*- coding: utf-8 -*-
##  Вычисление полинома по методу Горнера
from Tkinter import *
from bsc_groups import *
from math import*

##============================================== Function
def p(x,a):
   s=a[0]
   for i in a[1:]: s=s*x+i
   return s
##============================================== Calculating
a=[0.9,-9.0,3.0,12.0]   ## Коэффициенты полинома
bl=Bl('x','y')
x=-3.0
xk=11.0
dx=0.01
while x<=xk:
```

```
    y=p(x,a)
    bl.add(x=x,y=y)
    x+=dx
##================================================ Labels
crl=["red","black","#0DF","#C06","#A64","#686","#AC3"]
bg='White'
Lbd =[Bd(200,35,'Полином p(x)',crl[0],bg)]
Lbd+=[Bd(165, 500,"y = [%6.3f  %6.3f]" % (min(bl.L['y']),
max(bl.L['y'])),crl[0],bg)]
Lbd+=[Bd(165, 515,"x = [%6.3f  %6.3f]" % (min(bl.L['x']),
max(bl.L['x'])),crl[1],bg)]
##================================================ Tkinter
A,B=500,550
tk=Tk()
tk.title('График полинома')
fr=Frame(tk)
fr.pack()
c=Canvas(fr,bg=bg,width=A,height=B)
c.pack(expand=1,fill=BOTH)
#================================================ Output
desine_l(60,80,400,400,bg,crl,bl.L['x'],[bl.L['y']],0,50,50,c)
for ibd in Lbd: ibd.show(c)
tk.mainloop()
```

Производная аппроксимирующего полинома часто вычисляется быстрее, чем аппроксимируемая функция, имеет вид

$$\varphi'(x) = ma_0x^{m-1} + (m-1)a_1x^{m-2} + (m-2)a_2x^{m-3} + \ldots + a_{m-1}$$

и носит название метод Горнера.

```
# -*- coding: utf-8 -*-
##  Вычисление производной полинома по методу Горнера
from Tkinter import *
from bsc_groups import *
from math import*
```

```
##============================================ Function
def dp(x,a):
    l=len(a)
    s=a[0]
    for i in a[1:-1]:
        s=s*x*l+i
        l-=1
    return s
##============================================ Calculating
a=[0.9,-9.0,3.0,12.0]   ## Коэффициенты полинома
bl=Bl('x','y')
x,xk,dk=-3.0,11.0,
xk=11.0
dx=0.01
while x<=xk:
    y=dp(x,a)
    bl.add(x=x,y=y)
    x+=dx
##============================================= Labels
crl=["red","black","#0DF","#C06","#A64","#686","#AC3"]
bg='White'
Lbd =[Bd(200,35,'Полином p(x)',crl[0],bg)]
Lbd+=[Bd(165, 500,"y = [%6.3f  %6.3f]" % (min(bl.L['y']),
max(bl.L['y'])),crl[0],bg)]
Lbd+=[Bd(165, 515,"x = [%6.3f  %6.3f]" % (min(bl.L['x']),
max(bl.L['x'])),crl[1],bg)]
##============================================ Tkinter
A,B=500,550
tk=Tk()
tk.title('График полинома')
fr=Frame(tk)
fr.pack()
c=Canvas(fr,bg=bg,width=A,height=B)
```

```
c.pack(expand=1,fill=BOTH)
#========================================== Output
desine_l(60,80,400,400,bg,crl,bl.L['x'],[bl.L['y']],0,50,50,c)
for ibd in Lbd: ibd.show(c)
tk.mainloop()
```

4.1. Интерполяция

Одним из типов аппроксимации является интерполяция. Если коэффициенты a_j функции $\varphi(x)$ определяются из условия совпадения

$$f(x_j) = \varphi(x_j),$$

функции совпадают в заданных известных точках, то такой способ аппроксимации называется интерполяцией. Точки x_j – узлы интерполяции, а функция $\varphi(x)$ — интерполирующий полином. Интерполяционные кривые могут строиться отдельно для разных частей заданного интервала изменения x. Такая интерполяция называется кусочной или локальной.

При локальной интерполяции для каждого интервала строится своя функция. Рассмотрим линейную интерполяцию.

Заданы точки x_i (i = 1, 2, …, n) на отрезке [a; b] и значения функции в этих точках y_i. Уравнение на i-м интервале прямой, проходящей через точки (x_{i-1}, y_{i-1}) и (x_i, y_i), имеет следующий вид:

$$(y - y_{i-1})/(y_i - y_{i-1}) = (x - x_{i-1})/(x_i - x_{i-1}).$$

Отсюда $y = y_{i-1} + (y_i - y_{i-1})(x - x_{i-1})/(x_i - x_{i-1})$. По этой формуле можно определить функцию y в любой точке. Но сначала надо определить, в какой интервал отрезка [a; b] попадет искомая точка, т. е. определить i. Формулу можно использовать и для локальной интерполяции.

Локальная интерполяция по формуле Лагранжа

Примем, что $x(n)$, $y(n)$ — массивы узлов интерполяции, y_k — значение интерполяционного многочлена в точке x_k. В этом случае формула Лагранжа запишется так:

$$L(x) = \Sigma_{i=1,n} y_i \Pi_{j=1,n,j<>i} (x_k - x_i)/(x_i - x_j).$$

Обычно при локальной интерполяции не применяют многочлен выше четвертой степени. Для повышения точности надо увеличить количество узлов. Недостатком локальной интерполяции является наличие точек сопряжения интерполирующих кривых, а значит и разрыва производных, что недопустимо для некоторых задач.

Сплайн-интерполяция

Широкое распространение получило использование специальных функций – сплайнов. Обычно на каждом отдельном интервале это многочлен третьей степени. Уравнение заимствовано из теоретической механики, где оно представляет гибкую линейку и имеет вид

$$f(x) = a_i + b_i(x - x_{i-1}) + c_i(x - x_{i-1})^2 + d_i(x - x_{i-1})^3, \qquad (2)$$
$$x_{i-1} < x < x_i;\ i = 0, 1, 2, \ldots, n.$$

Оно обладает свойством минимальной кривизны, т. е. это самая гладкая из всех функций данного класса. Условия сопряжения в узлах:

$$f_i = y_i.$$
$$f'(x_i - 0) = f'(x_i + 0), \qquad (3)$$
$$f''(x_i - 0) = f''(x_i + 0).$$

Кроме того, задаются условия равенства нулю вторых производных на границах:

$$\text{при } x = x_0 \quad f'(x_0) = 0,$$
$$\text{при } x = x_n \quad f'(x_n) = 0.$$

Необходимо определить $4n$ коэффициентов, т. к. для каждого интервала — свои коэффициенты.

При $x = x_{i-1}$ из уравнения (3) получим

$$f(x_{i-1}) = a_{i-1} = y_{i-1}. \qquad (4)$$

При $x = x_i$ из того же уравнения получим:

$$f(x_i) = a_i + b_i h + c_i h^2 + d_i h^3, \qquad (5)$$

где $h = x_i - x_{i-1}$.

Продифференцируем уравнение (2) и запишем производные. Первая и вторая производные на i-м интервале:

$$f'(x) = b_i + 2c_i(x - x_i) + 3d_i(x - x_i)^2, \qquad (6)$$
$$f''(x) = 2c_i + 6d_i(x - x_i).$$

Производные на $(i + 1)$ интервале:

$$f'(x) = b_{i+1} + 2c_{i+1}(x - x_i) + 3d_{i+1}(x - x_i)^2,$$
$$f''(x) = 2c_{i+1} + 6d_{i+1}(x - x_i). \qquad (7)$$

Производные на границе интервала в точке $x = x_i$:

$$b_{i+1} = b_i + 2c_i h + 3d_i h^2,$$
$$c_{i+1} = c_i + 3d_i h. \qquad (8)$$

Получили систему уравнений (4), (5), (7), (8). Число уравнений (4), (5) – $2n$, уравнений (7), (8) – $(2n - 2)$, неизвестных – $4n$. Необходимы еще два уравнения. Их получим из условия равенства нулю вторых производных на концах интервала из уравнения (5):

$$\text{при } x = x_0 \quad c_1 = 0;$$
$$\text{при } x = x_n \quad c_n + 3d_n h = 0.$$

Получили систему из $4n$ линейных уравнений. Можно уменьшить размерность этой системы. Преобразуем систему. Из (8) получим

$$d_i = h(c_{i+1} - c_i)/3,$$
$$d_n = -hc_n/3. \qquad (9)$$

Подставим (9) в (5), учитывая, что $a_i = y_i$, получим

$$y_{i-1} + b_i h + c_i h_i^2 + h_i^3 (c - c_i)/3 = y_i.$$

Отсюда

$$b_i = (y_i - y_{i-1})/h - h(2c_i + c_{i+1})/3,$$
$$b_n = (y_n - y_{n-1})/h - 2c_n h/3.$$

Запишем уравнение (8) следующим образом:

$$b_i = b_{i-1} + 2c_{i-1}h + 3d^2_{i-1}$$

и подставим в него b_i и d_i:

$$(y_i - y_{i-1})/h - h(2c_i + c_{i+1})/3 =$$
$$= (y_{i-1} - y_{i-2})/h - h(2c_{i-1} + c_i)/3 + 2c_{i-1}h + h(c_i - c_{i-1})/3;$$

$$h(c_{i-1}/3 + 4c_i/3 + c_{i+1}/3) = -(y_{i-2} - 2y_{i-1} + y_i)/h;$$
$$hc_{i-1} + 4hc_i + hc_{i+1} = 3(y_i - 2y_{i-1} + y_{i-2})/h; \qquad (10)$$

$i = 2, \ldots, n$;

$c_1 = 0$; $c_{n+1} = 0$.

Получили систему линейных уравнений (10) с трехдиагональной матрицей.

Порядок решения методом сплайн-интерполяции:

1. Решение системы линейных уравнений (10) методом прогонки. В результате находим коэффициенты c_i ($i = 1, 2, ..., n$).
2. Определение коэффициентов b_i.
3. Определение коэффициентов d_i.
4. Определение коэффициентов $a_i = y_i$.

Недостатки сплайн-интерполяции:

1. Принято условие равенства нулю вторых производных на концах интервала. Это несколько произвольное условие, которое может ухудшить сходимость вблизи концов интервала. Существуют и другие методы задания дополнительных условий на концах интервала.
2. Все коэффициенты сплайна взаимосвязаны, поэтому приходится решать систему линейных уравнений размерности n. Если изменится хоть один узел, надо будет пересчитывать все коэффициенты.

Несмотря на недостатки, сплайн-интерполяция является одним из самых распространенных методов.

Выбор метода интерполяции зависит от расположения узлов интерполяции и их количества. Прежде чем проводить интерполяцию, рекомендуется построить график из имеющихся узлов и по его виду решить, какой способ лучше подходит для данного случая.

Интерполяционные многочлены используются для аппроксимации функций в промежуточных точках между крайними узлами интерполяции, т. е. $x_1 < x < x_n$. Иногда бывает нужно вычислить функцию вне рассматриваемого отрезка. Это приближение называют экстраполяцией. Для экстраполяции бывает недостаточно данных о поведении функции только внутри интервала. Нужны дополнительные данные о поведении функции вне рассматриваемого интервала.

4.2. Метод наименьших квадратов

Как показано выше, при интерполировании основным условием является прохождение графика интерполяционного многочлена через данные значения функции в узлах интерполяции. Однако в ряде случаев выполнение этого условия затруднительно или даже нецелесообразно. Например, при большом количестве узлов получается высокая степень многочлена в случае глобальной интерполяции. Кроме

того, данные могли быть получены из эксперимента путем измерений и содержать ошибки. Построение интерполяционного многочлена в таком случае означало бы повторение допущенных при измерениях ошибок. Тогда подбирается многочлен, график которого проходит не через заданные точки, а близко от них. Одним из таких методов является *среднеквадратичное приближение* функций с помощью многочлена

$$\varphi(x) = a_0x^m + a_1x^{m-1} + a_2x^{m-2} + \ldots + a_m,$$

при этом $m < n$. При $m = n$ получим интерполяцию (имея $(n + 1)$ узел). На практике стараются подобрать аппроксимирующий многочлен как можно меньшей степени, обычно $m = 1, 2, 3$.

Мерой отклонения многочлена $\varphi(x)$ от заданной функции $f(x)$ на множестве точек (x_i, y_i) (где $i = 1, ..., n$), при среднеквадратичном приближении является S, равная сумме квадратов разностей между значениями многочлена и функции в заданных точках:

$$S = \Sigma_{i=1,n}[\varphi(x_i) - y_i]^2.$$

Коэффициенты многочлена надо подобрать так, чтобы величина S была минимальной. В этом состоит *метод наименьших квадратов.* Запишем сумму квадратов отклонений для всех точек:

$$S = \Sigma_{i=1,n}[\varphi(x_i, a_0, a_1, ..., a_m) - y_i]^2. \qquad (11)$$

Коэффициенты $a_0, a_1, ..., a_m$ надо определить из условия минимума функции $S = S(a_0, a_1, ..., a_m)$.

Минимум функции S найдем, приравнивая нулю частные производные по этим переменным:

$$S'_a = 0,\ a = (a_0, a_1, ..., a_m).$$

Эти соотношения — система уравнений для определения $a_0, a_1, ..., a_m$. Найдем частные производные функции (11).

$$\Sigma_{i=1,n}[\varphi(x_i, a_0, a_1, ..., a_m) - y_i]\varphi(x_i, a_0, a_1, ..., a_m)'_{ai} = 0$$

$$\text{при } i = 1, 2, 3, ..., m.$$

Для того чтобы найти коэффициенты, надо задать вид функции:

$$\varphi(x_i, a_0, a_1, \ldots, a_m).$$

Пример. Зададим эту функцию как линейную:

$$\varphi(x) = a_0x + a_1, \qquad (12)$$

тогда $d\varphi/da_0 = x$, $d\varphi/da_1 = 1$.

Подставим значения функции и производных в систему (11), получим

$$\Sigma_{i=1,n}(a_0 x_i + a_1) - \Sigma_{i=1,n} y_i = 0,$$
$$\Sigma_{i=1,n}(a_0 x_i + a_1)x_i - \Sigma_{i=1,n} x_i y_i = 0.$$

Обозначим $S_1 = \Sigma x_i$; $S_2 = \Sigma y_i$; $S_3 = \Sigma x_i^2$; $S_4 = \Sigma x_i y_i$. Тогда систему можно записать по-другому:

$$na_1 + S_1 a_0 = S_2,$$
$$S_1 a_1 + S_3 a_0 = S_4.$$

Окончательно получим

$$a_1 = (S_2 S_3 - S_1 S_4)/(nS_3 - S_1^2),$$
$$a_0 = (nS_4 - S_1 S_2)/(nS_3 - S_1^2). \qquad (13)$$

Порядок решения методом наименьших квадратов:

1. Определить $S_1 S_2 S_3 S_4$.
2. Найти a_0 и a_1 по формулам (13).
3. Задав значение x из интервала $x_1 < x < x_2$, по формуле (11) найти значение функции.

Зададим аппроксимирующую функцию в виде квадратного трехчлена:

$$\varphi(x) = a_0 x^2 + a_1 x + a_2,$$
$$d\varphi/da_2 = 1,$$
$$d\varphi/da_1 = x,$$
$$d\varphi/da_0 = x^2.$$

Система относительно $a = (a_0, a_1, a_2)$ также является системой линейных уравнений. Аналогично можно получить систему уравнений для любой степени многочлена. Если в качестве аппроксимирующей функции выбрать многочлен, то получим систему линейных уравнений. Если бы мы выбрали не многочлен, то получили бы систему нелинейных уравнений. Иногда бывает удобнее выбрать аппроксимирующую функцию не в виде многочлена, а, например, в виде

$$\varphi(x) = ab^x. \qquad (14)$$

Приведем эту функцию к линейному виду

$$z = A_0 + A_1 x.$$

Для этого прологарифмируем (14):

$$\ln[\varphi(x)] = \ln[a] + x \ln[b].$$

Тогда получим

$$z = \ln[\varphi(x)];\ A_0 = \ln[a];\ A_1 = \ln[b]. \qquad (15)$$

Теперь A_0 и A_1 можно найти по обычным формулам и

$$a = \exp(A_0);\ \ b = \exp(A_1). \qquad (16)$$

Порядок решения методом наименьших квадратов:

1. Найти суммы S_i.
2. Определить A_0 и A_1 по формулам (15).
3. Найти a и b по формулам (16).
4. Задав любое значение x $(x_1 < x < x_n)$, по формуле (14) найти значение функции.

5. ЧИСЛЕННОЕ ДИФФЕРЕНЦИРОВАНИЕ

Производной функцией $y = f(x)$ называется предел отношения приращения функции Δy к приращению аргумента Δx при стремлении Δx к нулю:

$$y' = f'(x) = \lim_{\Delta x \to 0} (\Delta y/\Delta x),\ \Delta y = f(x + \Delta x) - f(x).$$

Обычно при вычислении производных используют аналитические формулы, но это не всегда возможно, в частности функция $y = f(x)$ может быть задана в виде таблицы. В этом случае используют приближенное равенство

$$y' \approx \Delta y/\Delta x. \qquad (17)$$

Это соотношение называется *аппроксимацией производной с помощью отношения конечных разностей.* Значения Δy и Δx в формуле (17) конечны.

Рассмотрим случай, когда функция $y = f(x)$ задана в виде таблицы экспериментальных значений, а значит, они определены с некоторой погрешностью. Геометрически производная представляет собой тангенс угла наклона касательной к функции в данной точке. Вычислим производную в точке x (рис. 7). В соответствии с выражением (17) запишем: $\Delta x = x_2 - x_1$; $\Delta y = y_2 - y_1$.

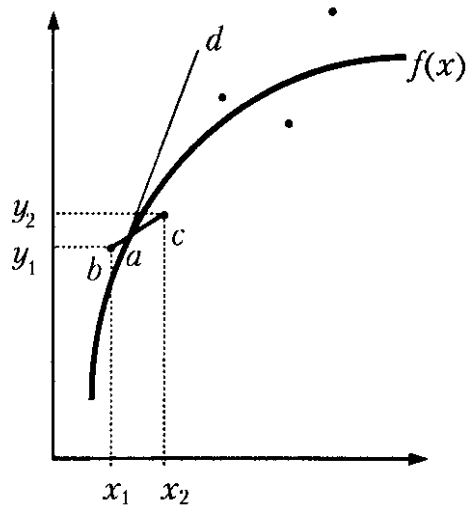


Рис. 7

Производная y' равна тангенсу угла наклона *касательной* к оси x. На самом же деле производная определяется касательной к кривой $f(x)$ при $x = x_1$. Задача численного дифференцирования в такой постановке является неустойчивой по исходным данным, т. е. при малой погрешности исходных данных можно получить большую погрешность результата решения.

Можно аппроксимировать функцию методом наименьших квадратов, получить аппроксимирующую функцию и вычислить ее производную аналитически.

Рассмотрим погрешность численного метода. Пусть функция $y = f(x)$ задана в виде таблицы и погрешностями исходных данных можно пренебречь. В таком случае мы касательную заменили хордой, соединяющей две точки на кривой (рис. 8). Погрешность зависит от кривизны кривой и шага: $h = x_2 - x_1$.

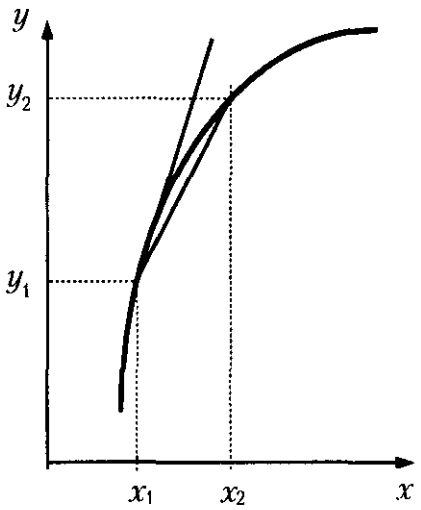


Рис. 8

Если кривизну кривой изменить нельзя, то, чтобы уменьшить погрешность, надо уменьшить шаг.

При этом нельзя неограниченно уменьшать шаг, т. к. увеличивается погрешность вычислений в компьютере и дальнейшее уменьшение шага не повысит точности результатов. Иногда функция бывает задана таблицей, и не имеется возможности определить дополнительные точки для уменьшения шага. В этом случае можно произвести «сгущение табличных данных»: провести интерполяцию сплайном, а затем определить производную численно или аналитически.

```
##Вычисление производной функции
# -*- coding: utf-8 -*-
from Tkinter import *
from bsc_groups import *
from math import*
##=========================================== Function
def f(x): return x**2*sin(x)
##=========================================== Calculating
bl=Bl('x','y','ys')
x =2.288930
xk=5.086985
dx=0.01
while x<=xk:
   y=f(x)
   bl.add(x=x,y=y,ys=0.0)
   x+=dx
l=len(bl.L['y'])-1
for i in range(l):
   ip=i+1
   im=i-1
   if im>0 and ip<l-1:
      ym=bl.L['y'][im]
      yp=bl.L['y'][ip]
      bl.L['ys'][i]=(yp-ym)/2.0/dx
```

```
    bl.L['ys'][0]=0.0
    bl.L['ys'][l]=0.0
##================================================ Labels
crl=["red","green","#0DF","#C06","#A64","#686","blue","#AC3"]
bg='White'
Lbd =[Bd(500,55,'График',crl[5],bg)]
Lbd+=[Bd(165, 600,"y = [%6.3f  %6.3f]" % (min(bl.L['y']),
max(bl.L['y'])),crl[0],bg)]
Lbd+=[Bd(165, 615,"x = [%6.3f  %6.3f]" % (min(bl.L['x']),
max(bl.L['x'])),crl[1],bg)]
##================================================ Tkinter
A,B=1000,700
tk=Tk()
tk.title('График')
fr=Frame(tk)
fr.pack()
c=Canvas(fr,bg=bg,width=A,height=B)
c.pack(expand=1,fill=BOTH)
#================================================= Output
desine_l(60,130,400,400,(255,255,255),crl,bl.L['x'],[bl.L['y'],bl.L['ys']],1,
50,50,c)
desine_l(510,130,400,400,(255,255,255),crl,bl.L['y'],[bl.L['ys']],0,50,50,c)
for ibd in Lbd: ibd.show(c)
tk.mainloop()
```

Рассмотрим аппроксимацию производной для функции $y = f(x)$, заданной в табличном виде: $y_0, y_1, \ldots, y_n$ при $x_0, x_1, \ldots, x_n$. Пусть шаг — разность между соседними значениями аргумента — постоянный и равен h. Запишем выражения для производной y'_1 при $x = x_1$. В зависимости от способа вычисления конечных разностей получаем разные формулы для вычисления производной в одной и той же точке при помощи:

— левых разностей $\Delta y_1 = y_1 - y_0$, $\Delta x = h$, $y'_1 = (y_1 - y_0)/h$;

— правых разностей $\Delta y_1 = y_2 - y_1$, $\Delta x = h$, $y'_1 = (y_2 - y_1)/h$;

— центральных разностей $\Delta y_1 = y_2 - y_0$, $\Delta x = 2h$, $y'_1 = (y_2 - y_0)/2h$.

Выражения для старших производных:

$$y''_1 = (y'_1)' = (y'_2 - y'_1)/h = ((y_2 - y_1)/h - (y_1 - y_0)/h)/h = (y_2 - 2y_1 + y_0)/h^2.$$

Так можно найти приближенные значения производных любого порядка. Рассмотрим погрешность численного дифференцирования. Аппроксимируем функцию $f(x)$ некоторой функцией $\varphi(x)$:

$$f(x) = \varphi(x) + R(x).$$

Продифференцируем это выражение:

$$f'(x) = \varphi'(x) + R'(x),$$
$$f''(x) = \varphi''(x) + R''(x),$$
$$\ldots\ldots\ldots\ldots\ldots\ldots$$
$$f^{(k)}(x) = \varphi^{(k)}(x) + R^{(k)}(x).$$

Величина $R'(x)$ называется погрешностью аппроксимации производной. Эта погрешность зависит от шага h, и ее записывают в виде $O(h^k)$. Показатель степени k называется *порядком погрешности* аппроксимации производной. Это означает, что погрешность приблизительно пропорциональна h^k. Чем больше k, тем меньше погрешность, т. к. $|h| < 1$.

Оценку погрешности аппроксимации можно произвести при помощи разложения функции в ряд Тейлора

$$f(x + \Delta x) = f(x) + f'(x)\Delta x + f''(x)\Delta x^2/2! + f'''(x)\Delta x^3/3!.$$

Первый из отброшенных членов ряда Тейлора называется главным членом погрешности. Обычно он бывает наибольшим из отброшенных и является мерой погрешности. Пусть функция $f(x)$ задана в виде таблицы $f(x_i) = y_i$ ($i = 0, 1, ..., n$). Запишем ряд Тейлора при $x = x_1$, $\Delta x = -h$ ($h = x_i - x_{i-1}$) с точностью до членов порядка h:

$$y_0 = f(x_0) = f(x_1 + \Delta x) = f(x_1 - h) = y_1 - y'_1 h + O(h^2).$$

Отсюда найдем значение производной в точке $x = x_1$:

$$y'_1 = (y_1 - y_0)/h + O(h^2).$$

Эта формула совпадает с формулой производной на левой границе интервала. Значит, левая производная имеет первый порядок точности. Аналогично, записывая ряд Тейлора при $\Delta x = h$, получим

$$y_2 = y_1 + y'_1 h + O(h^2).$$

Отсюда $y'_1 = (y_2 - y_1)/h + O(h)$.

Получили формулу для правой производной – она тоже имеет первый порядок точности.

Оценим таким же образом центральную и вторую производные. Полагая $\Delta x = -h$ и $\Delta x = h$, получим

$$y_0 = y_1 - y'_1 h + y''_1 h^2/2! - y'''_1 h^3/3! + O(h^4),$$
$$y_2 = y_1 + y'_1 h + y''_1 h^2/2! - y'''_1 h^3/3! + O(h^4).$$

Вычитая из второго равенства первое, получим

$$y_2 - y_0 = 2hy'_1 + O(h^3),$$
$$y'_1 = (y_2 - y_0)/2h + O(h^2).$$

Получили аппроксимацию производной с помощью центральных разностей. Она имеет второй порядок точности. Складывая те же равенства, получим

$$y''_1 = (y_0 - 2y_1 + y_2)/h^2 + O(h^2).$$

Получили выражение для второй производной – она тоже имеет второй порядок точности. Аналогично можно получить аппроксимацию производных более высоких порядков и оценку их погрешностей.

5.1. Аппроксимация производных по формуле Лагранжа

Мы рассмотрели аппроксимацию первой производной, используя две точки, и второй производной, используя три точки. Очевидно, можно получить лучшую точность, используя большее количество точек для аппроксимации. Запишем многочлен Лагранжа $L(x)$ для трех узлов интерполяции ($n = 2$) с равномерным расположением узлов ($x_i - x_{i-1} = h = \text{const}$):

$$L(x) = [(x - x_1)(x - x_2)y_0 - 2(x - x_0)(x - x_2)y_1 + (x - x_0)(x - x_1)y_2]/2h^2,$$
$$R_L(x) = y'''(x - x_0)(x - x_1)(x - x_2)/3!.$$

Продифференцируем эти выражения:

$$L'(x) = [(2x - x_1 - x_2)y_0 - 2(2x - x_0 - x_2)y_1 + (2x - x_0 - x_1)y_2]/2h^2,$$
$$R'_L(x) = y'''[(x - x_1)(x - x_2) + (x - x_0)(x - x_2) + (x - x_0)(x - x_1)]/3!.$$

Пусть $x = x_0$. Тогда в этой точке

$$y'_0 = L'(x_0) + R'_L(x_0) = (-3y_0 + 4y_1 - y_2)/2h + h^2 y'''/3.$$

Аналогично можно получить y'_1 и y'_2 при $x = x_1$ и $x = x_2$:

$$y'_1 = (y_2 - y_0)/2h - h^2y''/6,$$

$$y'_2 = (y_0 - 4y_1 + 3y_2)/2h + h^2y'''/3.$$

Эти формулы имеют второй порядок точности.

5.2. Улучшение аппроксимации

Порядок точности прямо пропорционален числу узлов, используемых при аппроксимации. С увеличением числа узлов аппроксимации формулы становятся все более громоздкими, усложняется оценка точности. Существует простой и эффективный способ уточнения производных – метод Рунге – Ромберга.

Рассчитывают производные с шагом h, затем — с шагом kh (k – целое). Обычно $k = 1$. Формула Рунге – Ромберга:

$$F'(x) = f'(x,h) + [f'(x,h) - f'(x,kh)]/(k^p - 1) + O(h^{p+1}),$$

где $F'(x)$ – уточненное значение производной; $f'(x, h)$ – производная, вычисленная с шагом h, имеющая порядок точности p. Здесь $f'(x, kh)$ – производная, вычисленная с шагом kh. При расчете по этой формуле порядок точности увеличивается на единицу.

6. ЧИСЛЕННОЕ ИНТЕГРИРОВАНИЕ

Пусть на отрезке $[a, b]$ задана функция $y = f(x)$. Разобьем отрезок на n элементарных отрезков с помощью точек $x_0, x_1, ..., x_n$. и $x_0 = a$; $x_n = b$. На каждом отрезке выберем произвольную точку ξ_i ($x_{i-1} < \xi_i < x_i$) и найдем произведение значения функции в этой точке $f(\xi_i)$ на длину отрезка:

$$S_i = f(\xi_i) \cdot \Delta x_{i.}.$$

Составим сумму этих произведений:

$$S_n = \Sigma_{i=1,n} f(\xi_i) \cdot \Delta x_i.$$

Сумма S_n называется интегральной суммой. Определенным интегралом от функции $f(x)$ на отрезке $[a, b]$ называется предел интегральной суммы при неограниченном увеличении числа точек разбиения, при этом длина наибольшего из элементарных участков стремится к нулю:

$$\int_{[a,b]} f(x)\,dx = \lim_{\Delta x \to 0} \Sigma_{i=1,n} f(\xi_i) \cdot \Delta x_i;\ x_{i-1} \le \xi_i \le x_i. \qquad (18)$$

Если функция $f(x)$ непрерывна на $[a, b]$, то предел интегральной суммы существует и не зависит ни от способа разбиения отрезка $[a, b]$ на элементарные отрезки, ни от выбора точек ξ_i.

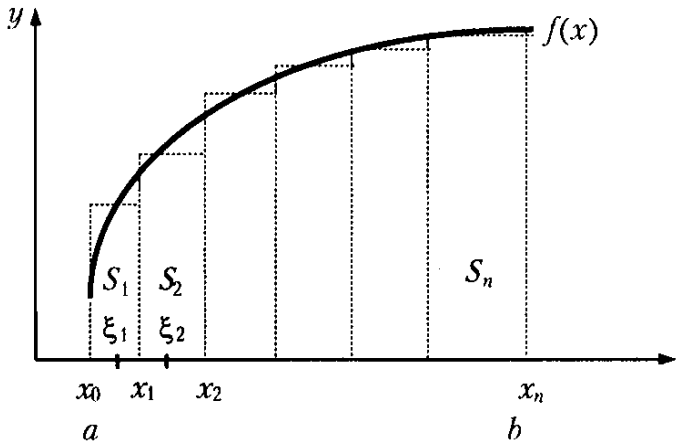


Рис. 9

Выражения S_i при $i = 1, 2, ..., n$ описывают площади элементарных прямоугольников (рис. 9). Интегральная сумма S_n — площадь ступенчатой фигуры, образуемой этими прямоугольниками. При неограниченном увеличении числа точек деления и стремлении к нулю всех элементов верхняя граница фигуры (ломаная) переходит в линию $y = f(x)$. Площадь полученной фигуры (криволинейной трапеции) будет равна определенному интегралу.

Невозможно использовать формулу, когда:

1) первообразную нельзя выразить в элементарных функциях;
2) значения функции $f(x)$ заданы в виде таблицы.

В этих случаях применяют методы численного интегрирования. Они основаны на аппроксимации подынтегральной функции некоторыми более простыми выражениями, например многочленами. Обычно используют локальную интерполяцию. В отличие от численного дифференцирования задача численного интегрирования является устойчивой по исходным данным.

6.1. Метод прямоугольников

В этом методе используют замену определенного интеграла интегральной суммой. В качестве точек могут выбираться левые x_{i-1} или правые x_i границы участков.

Для левых прямоугольников с постоянным шагом формула имеет вид

$$S_- = \Sigma_{i=1,n} h_i f(x_{i-1}) = h\Sigma_{i=1,n} f(x_{i-1});\ \ i = 1, 2, \ldots, n. \qquad (19)$$

Для правых прямоугольников:

$$S_+ = \Sigma_{i=1,n} h_i f(x_i) = h\Sigma_{i=1,n} f(x_i),\ \ i = 1, 2, \ldots, n. \qquad (20)$$

Главный член погрешности этих формул на элементарном участке равен $0{,}25h_i^2 f'(x_i)$ и имеет порядок точности $O(h^2)$. Точней формул (18), (19) является формула прямоугольников, использующая значения функции в средних точках участков:

$$S = \Sigma_{i=1,n} h_i f(x_{i-1} + h_i/2) = h\Sigma_{i=1,n} f(x_{i-1} + h/2),\ i = 1, 2, \ldots, n. \qquad (21)$$

Главный член погрешности формулы (21) на элементарном участке равен $h_i^3 f''(x_{i-1/2})$ и имеет порядок точности $O(h^3)$.

6.2. Метод трапеций

В этом методе используют линейную интерполяцию, т. е. график функции $y = f(x)$ представляется в виде ломаной, соединяющей точки (x_i, y_i). В этом случае площадь всей фигуры складывается из площадей элементарных трапеций (рис. 10).

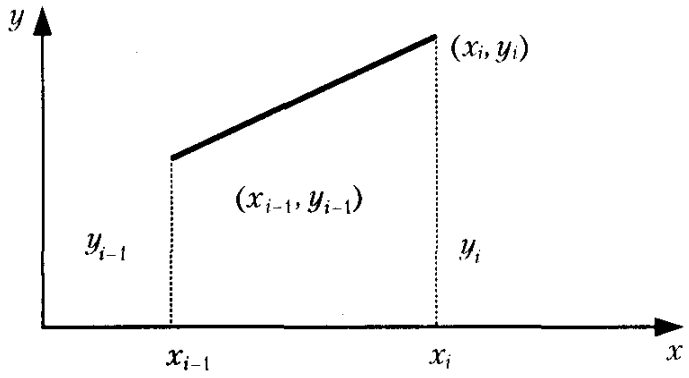


Рис. 10

Площадь каждой такой трапеции равна произведению полусуммы оснований на высоту:

$$S_i = (y_{i-1} + y_i)h_i/2,\quad i = 1, 2, \ldots, n.$$

Складывая все эти равенства, получим формулу трапеций:

$$S = (S_- + S_+)/2,$$

$$S = \Sigma_{i=1,n} h_i(y_{i-1} + y_i)/2 = h(y_0 + y_n)/2 + h\Sigma_{i=1,n-1} y_i.$$

Главный член погрешности на элементарном участке равен $h_i^3 f''(x_i)$.

```
##Интегрирование функции
# -*- coding: utf-8 -*-
from Tkinter import *
from bsc_groups import *
from math import*
##============================================= Function
def f(x): return x**2*sin(x)
##============================================= Calculating
bl=Bl('x','y','Y')
x,xk,dx =2.288930,5.086985,0.1
ss=0.0
yn=f(x)
bl.add(x=x,y=yn,Y=0.0)
while x<=xk:
    x+=dx
    yk=f(x)
    ss+=0.5*(yn+yk)*dx
    bl.add(x=x,y=yk,Y=ss)
    yn=yk
##================================================= Labels
crl=["red","green","#0DF","#C06","#A64","#686","blue","#AC3"]
bg='White'
Lbd =[Bd(500,55,'График',crl[5],bg)]
Lbd+=[Bd(165, 600,"Y= [%6.3f  %6.3f]" % (min(bl.L['Y']),
max(bl.L['Y'])),crl[2],bg)]
Lbd+=[Bd(165, 615,"y = [%6.3f  %6.3f]" % (min(bl.L['y']),
max(bl.L['y'])),crl[0],bg)]
Lbd+=[Bd(165, 630,"x = [%6.3f  %6.3f]" % (min(bl.L['x']),
max(bl.L['x'])),crl[1],bg)]
##============================================= Tkinter
A,B=1000,700
tk=Tk()
tk.title('График')
```

```
fr=Frame(tk)
fr.pack()
c=Canvas(fr,bg=bg,width=A,height=B)
c.pack(expand=1,fill=BOTH)
#============================================== Output
desine_l(60,130,400,400,(255,255,255),crl[1:],bl.L['x'],[bl.L['y']],0,50,50,c)
desine_l(60,130,400,400,(255,255,255),crl[2:],bl.L['x'],[bl.L['Y']],0,50,50,c)
desine_l(510,130,400,400,(255,255,255),crl,bl.L['y'],[bl.L['Y']],0,50,50,c)
for ibd in Lbd: ibd.show(c)
tk.mainloop()
```

Точность интегрирования зависит от степени многочлена, количества участков и расположения точек. Во многих случаях формула центральных прямоугольников дает лучшую точность, чем формула трапеций. Это, на первый взгляд, неожиданно, т. к. формула прямоугольников использует интерполяцию нулевого порядка, а формула трапеций — линейную. Здесь все дело в особом расположении точек, которое повышает точность.

6.3. Метод парабол (метод Симпсона)

Разобьем отрезок интегрирования [*a, b*] на четное число *n* равных частей с шагом *h.* На каждом отрезке подынтегральную функцию заменим многочленом второй степени:

$$f_i(x) = a_i x^2 + b_i x + c_i; \quad x_{i-1} \le x < x_{i+1}.$$

Коэффициенты этих квадратных трехчленов могут быть найдены из условий равенства многочлена в точках x_i соответствующим значениям функции y_i. В качестве многочлена можно принять многочлен Лагранжа второй степени, проходящий через точки (x_{i-1}, y_{i-1}), (x_i, y_i), (x_{i+1}, y_{i+1}). Затем на каждом участке взять определенный интеграл и их просуммировать. В результате получим формулу

$$S = h[y_0 + 4(y_1 + y_3 + \ldots + y_{n-1}) + 2(y_2 + y_4 + \ldots + y_{n-2}) + y_n]/3.$$

Главный член погрешности метода Симпсона на элементарном участке равен $h^5 f^{(IV)}(x_i)/480$.

В формулы погрешности *h* входит в некоторой степени. При $h \to 0$ погрешность стремится к нулю. Но при уменьшении *h* увеличивается

число участков, а погрешности на участках складываются. Например, в формулах для центральных прямоугольников и трапеций при уменьшении h в два раза погрешность на участке уменьшится в 8 раз, но число участков удвоится, значит, общая погрешность уменьшится в 4 раза. Коэффициент уменьшения ошибки не равен 4, т. к. вторая производная также изменяется и сказывается влияние членов более высокого порядка. Но можно считать, что для функций, имеющих непрерывные и ограниченные вторые производные, удвоение участков приблизительно уменьшает погрешность в 4 раза. Аналогично можно оценить влияние изменения шага на погрешность для других формул.

Алгоритм метода Симпсона

В программе:

a, b — соответственно нижний и верхний пределы интегрирования;
n — количество интервалов разбиения (должно быть четным);
$f(x)$ — подынтегральная функция.

```
# -*- coding: utf-8 -*-
##Интегрирование функции методом Симпсона
from math import *
##=============================================
Function
def f(x):
    return x**2*sin(x)
##=============================================
Calculating
a=2.288930
b=5.086985
n=100.0
h=(b-a)/n
x=a
s=f(x)
while x+h<b:
    x+=h
    s+=4.0*f(x)
    x+=h
    s+=2.0*f(x)
x+=h
```

```
s+=f(x)
s*=h/3
print s
```

Для вычисления интеграла с заданной точностью применяется следующий алгоритм. Задается требуемая погрешность вычисления интеграла e и количество элементарных участков. Вычисляется шаг h. При этом шаге находится интеграл I_1 по одной из рассмотренных ранее формул. Затем число участков удваивается и вычисляется интеграл I_2 при удвоенном числе участков. Условие окончания счета:

$$|(I_1 - I_2)/I_2| \le e. \qquad (22)$$

Если условие не выполняется, то количество участков опять удваивается и происходит новое вычисление интеграла. И так до тех пор, пока не выполнится условие окончания счета.

6.4. Метод Эйткена

Этот метод является способом уточнения расчета, выполненного одним из рассмотренных ранее методов. Расчет проводится три раза при трех различных шагах – h_1, h_2, h_3, причем их отношения постоянны $h_2/h_1 = h_3/h_2 = q$. Обычно $q = 0{,}5$. Если в результате численного интегрирования получены соответственно значения интеграла I_1, I_2, I_3, тогда уточненное значение при шаге h_1 будет определяться по формуле

$$I = I_1 - (I_1 - I_2)^2/(I_1 - 2I_2 + I_3).$$

Порядок точности уточненного значения интеграла рассчитывается по формуле

$$p = \ln(|I_3 - I_1|/|I_2 - I_1|)/\ln q.$$

6.5. Метод сплайнов

Рассмотренные выше методы можно применять, если подынтегральная функция задана аналитически, т. е. можно вычислить значение функции в любой точке. Если функция задана в виде таблицы, то лучше всего использовать сплайны. На каждом участке представим функцию в виде

$$\varphi_i(x) = a_i + b_i(x - x_{i-1}) + c_i(x - x_{i-1})^2 + d_i(x - x_{i-1})^3;$$
$$x_{i-1} < x < x_i;\ \ i = 1, 2, 3, \ldots, n;\ h_i = x_i - x_{i-1};$$
$$I = \int_{[a,\,b]} f(x)\,dx = \Sigma_{i=1,n} \int_{[\Delta xi]} \varphi_i(x)\,dx. \qquad (23)$$

Подставив (22) в (23) и проинтегрировав, получим

$$I = \Sigma_{i=1,n}(a_i h_i + b_i h_i^2/2 + c_i h_i^3/3 + d_i h_i^4/4).$$

Первый член правой части формулы совпадает с формулой трапеций. Значит, второй член характеризует поправку к методу трапеций, которую дает использование сплайнов.

7. ИНТЕГРИРОВАНИЕ ДИФФЕРЕНЦИАЛЬНОГО УРАВНЕНИЯ

Обыкновенными дифференциальными уравнениями называются уравнения, которые содержат одну или несколько производных от искомой функции $y = \varphi(x)$. Их можно записать в виде

$$F(x, y, y', \ldots, y^{(n)}) = 0,$$

где x — независимая переменная.
Наивысший порядок n входящей в уравнение производной определяет порядок дифференциального уравнения. Уравнение первого порядка можно записать в виде

$$F(x, y, y') = 0.$$

В ряде случаев из общей записи уравнения удается выразить старшую производную в явном виде:

$$y' = f(x, y);$$
$$y'' = f(x, y, y'). \qquad (22)$$

Такая форма называется уравнениями, разрешенными относительно старшей производной.

Решением дифференциального уравнения называется всякая функция $y = \varphi(x)$, которая после ее подстановки в уравнение превращает его в тождество.

Общее решение обыкновенного дифференциального уравнения n-го порядка содержит n произвольных постоянных C_1, C_2, ..., C_n, т. е. общее решение уравнения имеет вид

$$Y = \varphi(x, C_1, C_2, \ldots, C_n).$$

Частное решение дифференциального уравнения получается из общего, если произвольным постоянным придать определенные значения.

Для уравнения первого порядка общее решение зависит от одной произвольной постоянной:

$$y = \varphi(x, C).$$

Если постоянная принимает определенное значение $C = C_0$, то получится частное решение $y = \varphi(x, C_0)$.

Теорема Коши. Если правая часть $f(x, y)$ уравнения (22) и ее частная производная $f'(x, y)$ определены и непрерывны в некоторой области G изменения переменных x, y, то для всякой внутренней точки (x_0, y_0) этой области данное уравнение имеет единственное решение, принимающее заданное значение $y = y_0$ при $x = x_0$.

Для уравнений высших порядков геометрическая интерпретация более сложная. Через каждую точку в области решения при $n > 0$ проходит не одна интегральная кривая. Поэтому для выделения частного решения уравнения первого порядка надо задать координаты (x_0, y_0) произвольной точки на данной интегральной кривой. Для уравнений высших порядков этого недостаточно. Здесь правило следующее: для выделения частного решения из общего нужно задать столько дополнительных условий, сколько произвольных постоянных в общем решении, т. е. каков порядок уравнения.

В зависимости от способа задания дополнительных условий выделяют два вида задач:

1. Если условия задаются в единственной точке, то такая задача называется *задачей Коши.* Дополнительные условия в задаче Коши называются *начальными условиями.*

2. Если дополнительные условия задаются в разных точках, то такая задача называется *краевой.* Сами дополнительные условия называются при этом *граничными*, или *краевыми*, *условиями.* Обычно граничные условия задаются в двух точках: $x = a$ и $x = d$, являющихся границами области решения дифференциального уравнения.

Наиболее распространенными численными методами решения дифференциальных уравнений являются методы *конечных разностей.* Сущность этих методов состоит в следующем. Область непрерывного изменения аргумента и функции заменяется дискретным множеством точек, называемых узлами. Эти узлы составляют *разностную сетку.* Искомая функция непрерывного аргумента приближенно заменяется функцией дискретного аргумента на заданной сетке. Эта функция называется *сеточной.* Исходное дифференциальное уравнение заменяется разностным уравнением, т. е. производные заменяются конечно-разностными отношениями. Такая замена дифференциального уравнения разностным называется его *аппроксимацией на сетке или разностной аппроксимацией.*

Если все члены уравнения перенести в левую часть, то правая часть в этом случае не будет равна нулю, а будет равна невязке, которая в данном случае называется *погрешностью аппроксимации* $\varepsilon(x)$.

Как известно, погрешность аппроксимации может быть представлена в виде $\varepsilon = O(h^k)$. При этом говорят, что в данной точке x имеет место аппроксимация k-го порядка.

При решении дифференциального уравнения обычно требуется оценить погрешность аппроксимации не в одной точке, а на всей сетке. В качестве погрешности аппроксимации на сетке можно принять

$$\varepsilon_h = \max|\varepsilon(x_i)|.$$

Необходимо также аппроксимировать дополнительные условия на границах. Совокупность разностных уравнений, аппроксимирующих исходное уравнение, и дополнительных условий на границах называется *разностной схемой.*

Однако не всякая разностная схема дает удовлетворительное решение. Разностная схема *аппроксимирует* исходную дифференциальную задачу, если при $h \to 0$ $\varepsilon_h(x_i) \to 0$.

Под *устойчивостью* схемы понимается непрерывная зависимость ее решения от входных данных (коэффициентов уравнений, правых частей, начальных и граничных условий), или другими словами, малой погрешности исходных данных соответствует малая погрешность решения. Чувствительность разностной схемы характеризуется устойчивостью к различным погрешностям, которая является внутренним свойством разностной задачи, и это свойство не связано непосредственно с исходным дифференциальным уравнением. Естественно, для практических расчетов должны использоваться устойчивые схемы, т. к. входные данные обычно содержат погрешности, которые в случае неустойчивых схем приводят к неверному решению. Кроме того, в расчетах на компьютере погрешности возникают из-за округлений, а использование неустойчивых схем приводит к недопустимому накоплению этих погрешностей.

Обозначим искомое точное значение функции в узлах Y_i, а значение сеточной функции y_i ($i = 1, 2, ..., n$). Тогда погрешности решения в узлах будут таковы:

$$\delta y_i = y_i - Y_i.$$

Введем некоторое характерное значение этих погрешностей, например их максимальное по модулю значение на сетке:

$$\delta y = \max|\delta y_i|.$$

Разностная схема называется *сходящейся,* если выполняется условие $\lim\limits_{h\to 0}\max|\delta y_i| = 0$.

Теорема. Если решение исходной дифференциальной задачи существует, а разностная схема устойчива и аппроксимирует задачу на данном решении, то разностное решение сходится к точному. В этом случае узлы разностной схемы представляют объекты одного класса. Их удобно клонировать и обрабатывать одним списком.

7.1. Задача Коши

Требуется найти функцию $y = y(x)$, удовлетворяющую уравнению $\mathrm{d}Y/\mathrm{d}x = = f(x, Y)$.

Решение существует, и оно единственно. Используются разностные методы. Значения функции в узлах x_0, x_1, x_i, ..., x_n, заменим сеточной функцией y_0, y_1, y_i, ..., y_n. Заменяя значение производной отношением конечных разностей, осуществим переход к разностной задаче относительно сеточной функции:

$$y_{i+1} = F(x_i, h_i, y_{i-1}, y_i, \ldots).$$

Конкретное выражение правой части зависит от способа аппроксимации производной. Для каждого численного метода получается свой вид уравнения. Если в правой части уравнения отсутствует y_{i+1}, т. е. значение y_{i+1} вычисляется по предыдущим значениям y_i, y_{i-1}, ..., то разностная схема называется *явной.* Если для вычисления y_{i+1} требуется одна точка y_i, то метод одношаговый. Если требуется k точек, то метод k – шаговый. Если в правую часть уравнения входит искомое значение y_{i+1}, то решение уравнения усложняется. Такой метод называется *неявным.* В этом случае уравнение приходится решать итерационными методами.

7.2. Метод Эйлера

Задано уравнение $\mathrm{d}Y/\mathrm{d}x = f(x, Y)$. Начальные условия в задаче Коши имеют вид $x = x_0$, $Y(x_0) = Y_0$, а решение ищем для значений $x > x_0$.

Значения функции в узлах x_0, x_1, ..., x_n (рис. 11) заменим сеточной функцией $y_0, y_1, \ldots, y_n$. Для простоты примем постоянным шаг $h = \Delta x_i = x_{i+1} - x_i = \mathrm{const}$. Заменим производную конечно-разностным отношением

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

Отсюда $y_i = y_{i\text{-}1} + hf(x_{i\text{-}1}, y_{i\text{-}1})$.

Зная значение функции в начальной точке y_0, последовательно можно найти значения функции во всех точках сетки. Метод явный, одношаговый. Рассмотрим его геометрическую интерпретацию.

Точку $b(x_1, y_1)$ получим, проведя касательную к кривой, проходящей через начальную точку $a(x_0, y_0)$, до пересечения касательной с ординатой в точке x_1. Далее, проведя касательную в точке b до пересечения со следующей ординатой, получим точку $c(x_2, y_2)$ и т. д.

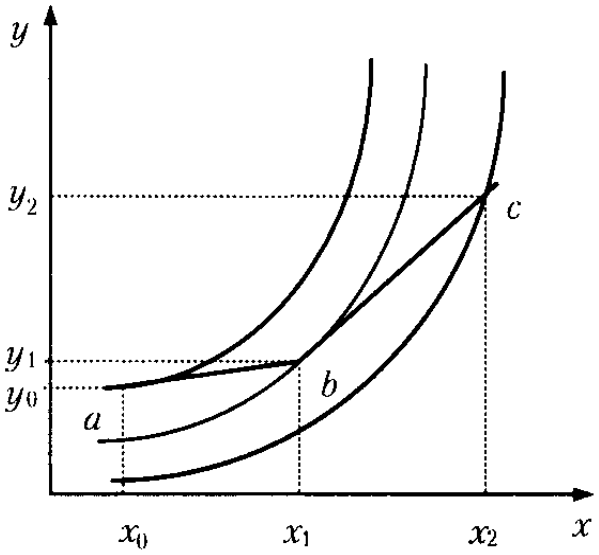


Рис. 11

Точным решением уравнения является интегральная кривая, которая проходит через точку $a(x_0, y_0)$. Значит, вместо кривой, проходящей через точку a, получили ломаную abc.

Оценим погрешность метода. Формулу y_i можно рассматривать как полученную разложением функции Y в ряд Тейлора:

$$Y(x_i + \Delta x_i) = Y(x_i) + Y'(x_i)\Delta x_i + \mathrm{O}(\Delta x_i^2).$$

Заменяя значения Y в узлах на значения сеточной функции y и учитывая, что производная $Y'(x_i) = f(x_i, y_i)$, получим $y_i = y_{i-1} + hf(x_{i-1}, y_{i-1}) + \mathrm{O}(h^2)$.

На каждом шаге погрешность имеет порядок $\mathrm{O}(h^2)$. При нахождении решения в точке x_n, отстоящей от x_0 на расстояние L, погрешность суммируется. Суммарная (глобальная) погрешность на n-м шаге равна $n\mathrm{O}(h^2)$. Так как $h = L/n$, то $n\mathrm{O}(h^2) = L\mathrm{O}(h^2)/h = \mathrm{O}(h)$.

Метод Эйлера имеет первый порядок точности. Он используется сравнительно редко.

Алгоритм метода Эйлера

В программе:

a, b — концы интервала; *h* — шаг интегрирования;
f(*x, y*) — правая часть уравнения.

```
# -*- coding: utf-8 -*-
##Интегрирование диф. уравнения методом Эйлера
from math import *
##============================ Function
def f(x,y):
   return y**2*sin(x)
##============================ Calculating
a=2.288930
b=5.086985
n=100.0
h=(b-a)/n
x,y=a,a
while x<=b:
   y+=h*f(x,y)
   x+=h
   print x,y
```

Рассмотрим в качестве примера систему четырех уравнений:

$$y'_1 = f_1(x, y_1, y_2, y_3, y_4),$$
$$y'_2 = f_2(x, y_1, y_2, y_3, y_4),$$
$$y'_3 = f_3(x, y_1, y_2, y_3, y_4),$$
$$y'_4 = f_4(x, y_1, y_2, y_3, y_4).$$

Начальные условия: $x = x_0$, $y_1 = y_{10}$, $y_2 = y_{20}$, $y_3 = y_{30}$, $y_4 = y_{40}$. Для решения системы дифференциальных уравнений формула запишется в виде

$$y_{j,i} = y_{j,i-1} + hf_j(x_{i-1}, y_{1,i-1}, y_{2,i-1}, y_{3,i-1}, y_{4,i-1}),$$

где j — номер неизвестного $j = 1, 2, ..., n$; i — номер точки.

Алгоритм метода Эйлера для системы уравнений

Производим интегрирование на интервале [*a, b*] с шагом интегрирования *h* и начальными условиями $X = X_0$, $Y = Y_0$.

```
# -*- coding: utf-8 -*-
##Интегрирование системы дифференциальных уравнений методом Эйлера
from math import *
##============================ Function
def f1(x,y):
    pass
  return dy1
def f2(x,y):
    pass
  return dy2
def f3(x,y):
    pass
  return dy3
def f4(x,y):
    pass
  return dy4

##============================ Calculating

class CF:
  def __init__(self,n,f):
    self.n,self.f=n,f
  def step(self,x,h,y):
    y[self.n]+=h*self.f(x,y)

a=2.288930
b=5.086985
n=100.0
```

```
h=(b-a)/n
x=a
l=4
y=[0.0,0.0,0.0,0.0]
F=[f1,f2,f3,f4]
Y=[]
for i in range(l): Y+=[CF(i,F[i])]
while x<=b:
    for i in Y: i.step(x,h,y)
    x+=h
    print x,y
```

7.3. Повышение точности. Метод Рунге—Кутты

Метод Рунге—Кутты является наиболее распространенным методом решения обыкновенных дифференциальных уравнений. Существуют разностные схемы разного порядка точности, построенные на основе этого метода. Приведем метод Рунге—Кутты четвертого порядка:

$$y_{i+1} = y_i + (k_0 + 2k_1 + 2k_2 + k_3)/6,$$
$$k_0 = hf(x_i, y_i),$$
$$k_1 = hf(x_i + h/2, y_i + k_0/2),$$
$$k_2 = hf(x_i + h/2, y_i + k_1/2),$$
$$k_3 = hf(x_i + h, y_i + k_2).$$

Метод явный, одношаговый. Имеет погрешность на шаге O (h^5), а значит, глобальную погрешность O(h^4). Метод Рунге—Кутты требует на каждом шаге четырехкратного вычисления правой части уравнения. Но это окупается повышенной точностью, что дает возможность проводить расчет с более крупным шагом.

```
# -*- coding: utf-8 -*-
##Интегрирование дифференциального уравнения методом Рунге—
Кутты
from math import *
##========================== Function
```

```
def f(x,y):
    return y**2*sin(x)
##============================ Calculating
class CF:
    def __init__(self,f):
      self.f=f
    def step(self,x,y,h):
      k0=h*self.f(x,y)
      k1=h*self.f(x+h/2.0,y+k0/2.0)
      k2=h*self.f(x+h/2.0,y+k1/2.0)
      k3=h*self.f(x+h,y+k2)
      return (k0+2.0*k1+2.0*k2+k3)/6.0


a=2.288930
b=5.086985
n=100.0
h=(b-a)/n
x,y=a,a
Y=CF(f)
while x<=b:
    y+=Y.step(x,y,h)
    x+=h
    print x, y
```

Если мы уменьшим шаг h в два раза, то, поскольку метод Рунге—Кутты имеет порядок точности $O(h^4)$, погрешность уменьшится примерно в 16 раз. Так как на каждом шаге правая часть уравнения вычисляется 4 раза, то количество вычислений увеличится в 8 раз. Чтобы уменьшить погрешность вычислений по методу Эйлера в 16 раз, надо шаг уменьшить в 16 раз, а значит, во столько же раз увеличится количество вычислений правой части. Следовательно, для метода Рунге—Кутты потребуется в два раза меньше времени на вычисление. Кроме того, при вычислении с шагом h точность решения методом Рунге—Кутты была выше. При уменьшении шага в большее число раз получим еще больший выигрыш во времени на вычисления.

Особенность одношаговых методов заключается в том, что для получения решения в каждом новом расчетном узле достаточно иметь значение сеточной функции лишь в предыдущем узле. Это допускает изменение шага в любой точке в процессе расчета. Недостатком одношаговых методов является трудность выбора шага, т. к. трудно вычислить погрешность на каждом шаге.

Алгоритм расчета по методу Рунге—Кутты подобен алгоритму расчета методом Эйлера. Разница в том, что внутри цикла сначала вычисляются k_0, k_1, k_2, k_3, а затем — значение y в новом расчетном узле.

Алгоритм решения аналогичен алгоритму решения системы уравнений методом Эйлера.

К решению систем уравнений сводятся также задачи Коши для уравнений высших порядков. Например, рассмотрим задачу Коши для уравнения второго порядка.

$$d^2Z/dx^2 = \varphi(x, Z, Z'), Z(x_0) = z_0, Z'(x_0) = y_0.$$

Обозначим $Z' = Y_1$, $Z = Y_2$. Тогда задача заменяется следующей:

$$dY_1/dx = \varphi(x, Y_1, Y_2);$$
$$dY_2/dx = Y_1;$$
$$Y_1(x_0) = y_0;$$
$$Y_2(x_0) = z_0.$$

7.4. Многошаговые методы

При аппроксимации производных лучшая точность получается, если использовать информацию с большим количеством точек. Для вычисления y_{i+1} используются результаты не одного, а k предыдущих шагов, т. е. значения y_{i-k+1}, y_{i-k+2}, ..., y_i. В этом случае получается k-шаговый метод.

Существуют явные и неявные методы. Хороший результат можно получить, используя комбинацию явного и неявного метода. Эти комбинации называются методами *прогноза и коррекции.* Каждый шаг состоит из двух этапов и рассчитывается многошаговыми методами.

С помощью явного метода (прогноза) по известным значениям функции в предыдущих узлах находится начальное приближение в новом узле. Используя неявный метод (коррекции), в результате итераций уточняется это значение.

```
# -*- coding: utf-8 -*-
##Интегрирование дифференциального уравнения двушаговым методом ##(прогноз - коррекция)
from math import *
##============================ Function
def f(x,y):
    return y**2*sin(x)
##============================ Calculating
class CF:
    def __init__(self,f):
      self.f=f
    def step(self,x,y,h):
      y1=y+h*self.f(x,y)            # прогноз
      return 0.5*(f(x,y)+f(x+h,y1))*h   # коррекция

a=2.288930
b=5.086985
n=100.0
h=(b-a)/n
x,y=a,a
print '----------------'
print x, y
Y=CF(f)
while x<=b:
    y+=Y.step(x,y,h)
    x+=h
    print x, y
```

8. ЗАДАЧИ ВЫБОРА

Рассмотрим задачи поиска экстремума для функций из класса дважды дифференцируемых. Считаем, что проведены предварительные исследования, показавшие наличие на выбранном интервале [*a,b*] одного максимума. Этот интервал делится на *n* отрезков длиной

$h = (b–a)/n$. В каждом узле, ограничивающем отрезок, вычисляется значение функции и создается их список, в котором определяется максимальное значение. Выделяем новый интервал длиной $2h$, серединой его будет узел с максимальным значением функции. Повторяем с этим интервалом операции такие же, как и с [*a, b*]. Операции повторяются до тех пор, пока длина отрезка не станет меньше заданной точности *e*.

```
## Поиск максимума функции  одной переменной
# -*- coding: utf-8 -*-
from Tkinter import *
from bsc_groups import *
from math import*

def f(x): return x**2*sin(x)

x1=0.0
x2=2.0*pi
n=500.0
dx=(x2-x1)/n
e=0.00001
bl=Bl('x','y')
while 1:
   x=x1
   bl.L['x']=[]
   bl.L['y']=[]
   while x<=x2:
      y=f(x)
      bl.add(x=x,y=y)
      x+=dx
   ymax=max(bl.L['y'])
   i=bl.L['y'].index(ymax)
   if (i==0) or (i==len(bl.L['y'])+1):
```

```
        print 'Граница'
        break
    xmax=bl.L['x'][i]
    x1=xmax-dx
    x2=xmax+dx
    dx=(x2-x1)/n
    print 'x1=%10.6f  x2=%10.6f' % (x1,x2)
    if x2-x1<e:
        print 'max:'
        break
print 'x=%10.6f y=%10.6f' % (xmax,ymax)


## Метод покоординатного спуска
from Tkinter import *
from bsc_groups import *
from math import*

L1,L2=[0.0,0.0],[2.0*pi,2.0*pi]
n,e,bl,Ln,Lx=100.0,0.000001,Bl('x','y'),L1[:],L1[:]
print 'x1n=%11.8f  x1k=%11.8f  x2n=%11.8f  x2k=%11.8f' %
(L1[0],L2[0],L1[1],L2[1])
print '_________'*8

def f(x): return x[0]**2*sin(x[0])+x[1]**3*cos(x[1])

def ff(j,x1,x2,Lx):
    bl.L['x'],bl.L['y'],x,dx=[],[],x1,(x2-x1)/n
    while x<=x2:
        Lx[j]=x
        bl.add(x=x,y=f(Lx))
        x+=dx
    ymax=max(bl.L['y'])
    i=bl.L['y'].index(ymax)
```

```
    xmax=bl.L['x'][i]
    if   i==0:              x1,x2=xmax,   xmax+dx
    elif i==len(bl.L['y'])+1: x1,x2=xmax-dx,xmax
    else:                   x1,x2=xmax-dx,xmax+dx
    return xmax,x1,x2

while 1:
    ll=len(L1)
    for j in xrange(ll): Lx[j],L1[j],L2[j]=ff(j,L1[j],L2[j],Lx)
    v2=0.0
    for j in xrange(ll): v2+=(Lx[j]-Ln[j])**2
    v=sqrt(v2)
    print 'xx1=%11.8f  xx2=%11.8f  v=%11.8f' % (Lx[0],Lx[1],v)
    if v<e: break
    Ln=Lx[:]

print '_________'*8
print 'x1=%11.8f  x2=%11.8f  v=%11.8f  e=%11.8f' % (Lx[0],Lx[1],v,e)
```

9. ПРОГРАММИРОВАНИЕ НА ГРАФАХ

Практически все задачи программирования численных методов представимы в виде программ на ориентированных замкнутых связных графах. Наиболее простой способ описания графа — это метод перечисления (например, ребер). Из этого метода получим метод описания в виде матрицы инциденций:

```
## Определение матрицы инциденций по списку ребер
G={1:[1,2],2:[1,5],3:[1,3],4:[2,3],5:[4,5],6:[2,5]}
print G
print 'Матрица инциденций'
sp=[]
for i in G.values(): sp+=[max(i)]
ll=max(sp)
```

```
S=[0]*ll
for i in G.values():
    SS=S[:]
    i1=i[0]-1
    i2=i[1]-1
    SS[i1]=-1
    SS[i2]=1
    st=""
    for j in SS:
        st+="%3i" % j
    print st
```

или способ матрицы смежности:

```
## Определение матрицы смежности по списку ребер
G={1:[1,2],2:[1,5],3:[1,3],4:[2,3],5:[5,4],6:[2,5]}
sp=[]
for i in G.values(): sp+=[max(i)]
lm=max(sp)
print sp
print 'Матрица смежности'
ss=[]
for i in range(lm):
    s=[]
    for j in range(lm): s+=[0]
    ss+=[s]
for i in G.values():
    i1=i[0]-1
    i2=i[1]-1
    ss[i1][i1]+=1
    ss[i2][i2]+=1
    ss[i1][i2]=-1
    ss[i2][i1]=-1
```

```
for i in ss:
    st=""
    for j in i:
        st+="%3i" % j
    print st
```

Для задач, формализованных как поиск путей в графе, следующая программа является одной из основных.

```
“””” Поиск всех возможных путей в ориентированных графах от назначенной первой  вершины к пятой “””“
# -*- coding: utf-8 -*-
E={1:[1,2],
   2:[1,3],
   3:[1,5],
   4:[2,3],
   5:[2,4],
   6:[3,4],
   7:[4,5]}
sp=[]
i=1
ik=5
while i<ik:
    for j in E.values():
        if i==j[0]:
            if i==1: sp+=[j]
            else:
                for k in sp:
                    if i==k[-1]: k+=[j[1]]
                    elif (i in k) and (i<>k[-1]) and j[1] not in k:
                        sr=k[:i]+[j[1]]
                        sp+=[sr]
    i+=1
for k in sp: print k
```

Библиотека вспомогательных программ bsc_groups.py

В библиотеке находятся программы, необходимые для сохранения результатов расчета и представления их в графическом виде.

```
# -*- coding: utf-8 -*-
import sys
from Tkinter import *
from math import *
import shelve
#-----------------------------------------------
def LayImagine(L=[(1,1),(10,10)], m=1.0, nx=100, ny=100,c=None,col=
"black"):
    line=[]
    for (i,j) in L:
        ix,iy=nx+int(i/m),ny-int(j/m)
        line+=[(ix,iy)]
    c.create_line(line,fill=col)

class Bl:
    def __init__(self,*L):
        self.k,self.L,self.LL=L[0],{},{}
        for i in L:
            self.L[i]=[]
            self.LL[i]=[]
    def add(self,**L):              #L={}):
        for i,j in L.items():
            if i in self.L.keys(): self.L[i]+=[j]
    def adf(self,**L):
        for i,j in L.items(): self.L[i].insert(0,j)
    def addL(self,**L):
        for i,j in L.items():
            if i in self.L.keys(): self.LL[i]+=[j]
    def show(self,name='',*L):
```

```
        sf,s="""""""",""
        for i in L:
            if i in self.L.keys(): s+="%18s;" % i
        sf+= s+"\n"
        for k in xrange(len(self.L[self.k])):
            s=""
            for i in L:
                if i in self.L.keys(): s+="%18.6f;" % self.L[i][k]
            sf+=s+"\n"
        f=open(name,'w')
        f.write(sf)
        f.close()
    def showL(self,m,x,y,c,col='black'):
        for i in self.LL.values():
            for k in i: LayImagine(k,m,x,y,c,col)
    def showLk(self,k,m,x,y,c):
        for i in self.LL[k]: LayImagine(i,m,x,y,c)
#---------------------------------------
class Bd:
    def __init__(self,x,y,s,fg,bg):
        self.x,self.y,self.s,self.fg,self.bg=x,y,s,fg,bg
    def show(self,c):
        l=Label(c,text=self.s,fg=self.fg,bg=self.bg, justify='left')
        l.pack()
        c.create_window(self.x,self.y,window=l)
#---------------------------------------
def take(name):
    f=open(name,'r')
    sf=f.read()
    f.close()
    ls=sf.split('\n')
    #---------------
    s0=ls[0]
```

```
    ls0=s0.split(';')
    l0=[]
    for i in ls0:
        j=i.strip()
        l0+=[j]
    #---------------
    d={}
    len0=len(l0)
    for i in range(len0): d[i]=[]
    for i in ls[1:]:
        lsi=i.split(';')
        k=0
        for j in lsi:
            jj=j.strip()
            try:
                ff=float(jj)
                d[k]+=[ff]
            except: print k,'--->',jj
            k+=1
    bl=Bl(l0[0])
    bl.L[l0[0]]=d[0]
    for i in range(len0)[1:]:
        bl.L[l0[i]]=d[i]
    return bl
#-----------------------------------------------------grafik
def maker(a=580, b=380,nx=50,ny=50,dx=10,dy=10):
    ixn,iyn,ixk,iyk=nx,ny,a+nx,b+ny
    ix,iy=ixn,iyn
    line=[ix,iy]
    while ix<ixk:
        line+=[ix,iyk,ix,iy]
        ix+=dx
        line+=[ix,iy]
```

```
    ix,iy=ixk,iyk
    line+=[ix,iy]
    while iy>iyn:
        line+=[ixn,iy,ix,iy]
        iy-=dy
        line+=[ix,iy]
    return line

def makel(a=580.0, b=380.0,x=[0.0, 580.0],y=[0.0,380.0],nx=5,ny=10):
    xmin, ymin, xmax, ymax = min(x), min(y), max(x), max(y)
    dx, dy = (xmax-xmin)/a, (ymax-ymin)/b
    if dx == 0.0 or dy == 0.0: return [nx,ny,x,y]
    line = []
    for i in xrange(len(x)):
        ix, iy = nx+int((x[i]-xmin)/dx), ny+int((ymax-y[i])/dy)
        line+=[ix,iy]
    return line

def
makepl(a=400,b=400,fil=["black"],lt=[],lx=[],nx=20,ny=450,c=None):
    k,kl,mi=0,len(fil)-1,mashtab1(lt,lx,a,b,nx,ny)
    for i in mi:
        c.create_line(i,fill=fil[k])
        if k<kl: k+=1
        else: k=0

def mashtab1(mx,mmy,x,y,nx,ny):
    max_x,min_x=max(mx),min(mx)
    dx=(max_x-min_x)/x
    max_y,min_y=None,None
    for my in mmy:
        max_ay,min_ay=max(my),min(my)
        if max_y==None or max_y<max_ay: max_y=max_ay
        if min_y==None or min_y>min_ay: min_y=min_ay
    dy=(max_y-min_y)/y
```

```
    mmi=[]
    for my in mmy:
        mi=[]
        for i in range(len(mx)):
            ix,iy=int((mx[i]-min_x)/dx)+nx,y+ny-int((my[i]-min_y)/dy)
            kort=(ix,iy)
            mi.append(kort)
        mmi+=[mi]
    return mmi
def mashtab2(mx,mmy,x,y,by=0.0):
    max_x,min_x=max(mx),min(mx)
    dx=(max_x-min_x)/x
    if by==0:
        max_y,min_y=None,None
        for my in mmy:
            max_ay,min_ay=max(my),min(my)
            if max_y==None or max_y<max_ay: max_y=max_ay
            if min_y==None or min_y>min_ay: min_y=min_ay
        dy=(max_y-min_y)/y
    mmi=[]
    for my in mmy:
        mi=[]
        if by:
            max_y,min_y=max(my),min(my)
            dy=(max_y-min_y)/y*by
        for i in range(len(mx)):
            ix,iy=int((mx[i]-min_x)/dx),y-int((my[i]-min_y)/dy)
            if ix<1: ix=1
            if iy<1: iy=1
            if ix>x-1: ix=x-1
            if iy>y-1: iy=y-1
            kort=(ix,iy)
            mi.append(kort)
```

```
        mmi+=[mi]
    return mmi,max_x,min_x,max_y,min_y
def desine_f(name,a=400,b=300,rgb=(255,255,255),fil=[(0,0,0)],
                    lt=[],lx=[],by=0,mes='proba',dix=50,diy=50):
    fill=(0,0,0)
    ab=[(1,1),(a-1,1),(a-1,b-1),(1,b-1),(1,1)]
    imp = Image.new('RGB',(a,b),rgb)
    draw = ImageDraw.Draw(imp, mode='RGB')
    x,y,ix,iy=a,b,0,0
    if x and y:
        mi,max_x,min_x,max_y,min_y=mashtab2(lt,lx,x,y,by)
        l=len(lx)
        dl=255/l
        (r,g,bl)=fill
        k=0
        kl=len(fil)-1
        for i in mi:
            draw.line(i,fil[k])
            if k<kl:
                k+=1
            else:
                k=0
        if mes:
            s='%8.3f;%8.3f' % (min_y,max_y)
            draw.text((10,5),mes+s, fill=(1,100,250))
        draw.line(ab,fill)
        while iy<y:
            iy+=diy
            draw.line([(1,iy),(x,iy)],fill)
        while ix<x:
            ix+=dix
            draw.line([(ix,1),(ix,y)],fill)
    imp.save(name,'gif')
```

```
def mashtab_l(mx,mmy,x,y,by=0.0,nx=1,ny=1):
    max_x,min_x=max(mx),min(mx)
    dx=(max_x-min_x)/x
    if by==0:
        max_y,min_y=None,None
        for my in mmy:
            max_ay,min_ay=max(my),min(my)
            if max_y==None or max_y<max_ay: max_y=max_ay
            if min_y==None or min_y>min_ay: min_y=min_ay
        dy=(max_y-min_y)/y
    mmi=[]
    for my in mmy:
        mi=[]
        if by:
            max_y,min_y=max(my),min(my)
            dy=(max_y-min_y)/y*by
        for i in range(len(mx)):
            ix,iy=int((mx[i]-min_x)/dx),y-int((my[i]-min_y)/dy)
            if ix<1: ix=1
            if iy<1: iy=1
            if ix>x-1: ix=x-1
            if iy>y-1: iy=y-1
            kort=(ix+nx,iy+ny)
            mi.append(kort)
        mmi+=[mi]
    return max_x,min_x,max_y,min_y,dy,dx,mmi
def desine_l(nx=1,ny=1,a=400,b=300,rgb=(255,255,255),fil=[(0,0,0)],
                 lt=[],lx=[],by=0,dix=50,diy=50,c=None):
    if len(lt)<2: return
    x,y,ix,iy=a,b,0,0
    if x and y:
        max_x,min_x,max_y,min_y,dy,dx,mi=mashtab_l(lt,lx,x,y,by,nx,ny)
        if max_x==min_x or max_y==min_y or mi==[]: return
```

```
        l=len(lx)
        dl=255/l
        k,kl=0,len(fil)-1
        for i in mi:
            c.create_line(i,fill=fil[k])
            if k<kl: k+=1
            else:    k=0
    rline=maker(a,b,nx,ny,dix,diy)
    c.create_line(rline, fill="black")
    if max_y*min_y<0 and by==0:
        c.create_line([nx,ny+b+int(min_y/dy),nx+a,ny+b+int(min_y/dy)],
fill="gray")
    if max_x*min_x<0 and by==0:
        c.create_line([nx+a-int(max_x/dx),ny,nx+a-int(max_x/dx),ny+b],
fill="gray")

#-------------------------------------------------class
def F0(x,y,r):
    df=0.05*pi
    fk,f,L=2.0*pi+df,0.0,[]
    while f<fk:
        xc,yc=r*cos(f)+x,r*sin(f)+y
        L+=[(xc,yc)]
        f+=df
    return L

def F4(x,y,a,b,u=0.0):
    alf=atan(b/a)
    r=sqrt(a*a+b*b)
    dx1=r*cos(alf+u)
    dy1=r*sin(alf+u)
    dx2=r*cos(u-alf)
    dy2=r*sin(u-alf)
    L=[(x+dx1,y+dy1),(x+dx2,y+dy2),(x-dx1,y-dy1),(x-
dx2,ydy2),(x+dx1,y+dy1)]
    return L
```

```
def F3(x,y,a,b):
    L=[(x,y),(x-a,y-b),(x+a,y-b),(x,y)]
    return L

def put_shelve(key,data,fname):
    d = shelve.open(fname)
    d[key] = data
    d.sync()
    d.close()
    return 1
def get_shelve(key,fname):
    d = shelve.open(fname)
    if d.has_key(key):
        data=d[key]
        d.close()
        return data
    else:
        d.close()
        return 0
def del_shelve(key,fname):
    d = shelve.open(fname)
    if d.has_key(key):
        del d[key]
        d.sync()
        d.close()
        return 1
    else:
        d.close()
        return 0
```

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

Бахвалов Н. С. Численные методы / Н. С. Бахвалов, Н. П. Жидков, Г. М. Кобельков. М.: Наука, 1987. 597 с.

Бут Э. Д. Численные методы / Э. Д. Бут. М.: Физматгиз, 1959. 239 с.

Воробьева Г. Н. Практикум по вычислительной математике / Г. Н. Воробьева, А. Н. Данилова. М.: Высшая школа, 1990. 207 с.

Мак-Кракен Д. Численные методы и программирование на Фортране / Д. Мак-Кракен, У. Дорн. М. : Мир, 1977.

Турчак Л. И. Основы численных методов / Л. И. Турчак. М.: Наука,1987. 318 с.

Хемминг Р. В. Численные методы / Р. В. Хемминг. М.: Наука, 1968. 400 с.

Шуп Т. Решение инженерных задач на ЭВМ / Т. Шуп. М.: Мир, 1982. 235 с.

ОГЛАВЛЕНИЕ

Учебное издание

Буйначев Сергей Константинович

Применение численных методов в математическом моделировании

Редактор *И. В. Коршунова*
Компьютерная верстка – *Я. П. Бояршинов*

Подписано в печать 27.05.2014. Формат 60×90 1/16.
Бумага писчая. Плоская печать. Усл. печ. л. 4,5.
Уч.-изд. л. 2,7. Тираж 50 экз. Заказ № 1158.

Издательство Уральского университета
Редакционно-издательский отдел ИПЦ УрФУ
620049, Екатеринбург, ул. С. Ковалевской, 5
Тел.: 8(343)375-48-25, 375-46-85, 374-19-41
E-mail: rio@urfu.ru

Отпечатано в Издательско-полиграфическом центре УрФУ
620075, Екатеринбург, ул. Тургенева, 4
Тел.: 8(343) 350-56-64, 350-90-13
Факс: 8(343) 358-93-06
E-mail: press-urfu@mail.ru

Для заметок

Для заметок

ОБ АВТОРЕ

БУЙНАЧЕВ
Сергей Константинович

Кандидат технических наук, доцент кафедры «Детали машин». Специалист в области теории механизмов и машин, деталей машин, прикладной механики, математического моделирования и оптимизации, а также математического моделирования процессов и оборудования производства.