

Глава III.

Разработка программ

1. <i>Проектирование программ</i>	2
▣ Этапы разработки программ	2
▣ Программирование «снизу вверх»	4
▣ Структурное программирование	4
2. <i>Построение графиков и работа с ними</i>	7
▣ Структура программы	7
▣ Способы задания функций	8
▣ Системы координат	11
▣ Построение графиков	14
▣ Как найти точки пересечения?	16
▣ Штриховка замкнутой области	21
▣ Площадь замкнутой области	23
3. <i>Вычислительные методы</i>	27
▣ Целочисленные алгоритмы	27
▣ Многозарядные целые числа	29
▣ Многочлены	30
▣ Последовательности и ряды	31
▣ Численное решение уравнений	35
▣ Вычисление определенных интегралов	39
▣ Вычисление длины кривой	39
▣ Оптимизация	40
4. <i>Моделирование</i>	44
▣ Что такое модель?	44
▣ Виды моделей	44
▣ Вращение	45
▣ Использование массивов	46
▣ Математическое моделирование физических процессов	49
5. <i>Сложные проекты</i>	57
▣ Зачем нужны проекты?	57
▣ Как создать проект?	57
▣ Пример проекта	58

1. Проектирование программ

Этапы разработки программ

Постановка задачи

Постановка задачи – наиболее важный этап разработки программы. Обычно сначала она формулируется на естественном языке, причем при выполнении коммерческого проекта все требования к программе необходимо закрепить в договоре в письменной форме.

Различают **хорошо и плохо поставленные задачи**. В хорошо поставленных задачах четко выделены исходные данные и определены взаимосвязи между ними, позволяющие получить однозначный результат (единственное решение).

Задача называется **плохо поставленной**, если для однозначного решения не хватает исходных данных, то есть, неясны связи между исходными данными и результатом.

Предположим, Вася купил компьютер. Постепенно его части выходят из строя и ремонт обходится дороже с каждым годом. В то же время, новые компьютеры дешевеют, и наступает момент, когда выгоднее не ремонтировать старый компьютер, а продать его и купить новый. Когда Васе выгоднее всего продать старый компьютер и купить новый?

Эта задача плохо (некорректно) поставлена, поскольку не заданы исходные данные (стоимость нового компьютера, стоимость ремонта каждый год, и т.д.). Вот еще примеры плохо поставленных задач¹:

- Винни Пух и Пятачок построили ловушку для слонопотама. Удастся ли его поймать?
- Винни Пух и Пятачок пошли в гости к Кролику. Сколько бутербродов с медом может съесть Винни, чтобы не застрять в двери?
- Малыш и Карлсон решили по-братски разделить два орешка – большой и маленький. Как это сделать?
- Аббат Фариа решил бежать из замка Иф. Сколько времени ему понадобится для этого?

Однако, если четко определить все существенные свойства, их численные значения и взаимосвязь между исходными данными и результатом, эти задачи становятся *хорошо поставленными* и могут быть решены.

Разработка модели данных

Одна из основных проблем в практическом программировании – правильно построить *формальную модель* задачи (на языке математики и логики), определить нужную структуру данных (переменные, массивы, структуры) и взаимосвязи между ними. Большинство наиболее серьезных ошибок в программах вызваны именно ошибками в выборе модели данных.

Разработка алгоритма

На этом этапе необходимо выбрать или разработать заново **алгоритм**.

Алгоритм – это четко определенная конечная последовательность действий, которую надо выполнить для решения задачи.

Алгоритм должен учитывать выбранную ранее модель данных. Иногда приходится возвращаться к предыдущему шагу, поскольку для того, чтобы использовать выбранный алгоритм,

¹ А.Г. Гейн и др., Информатика, учебник для 8-9 кл. общеобразоват. учреждений, М.: Просвещение, 1999.

удобнее организовать данные по-другому, например, использовать структуры вместо нескольких отдельных массивов.

📖 **Разработка программы**

Для того, чтобы исполнитель (например, компьютер) понял и выполнил задание, надо составить его на понятном исполнителю языке, то есть написать **программу**.

Программа – это алгоритм, записанный на языке программирования. Часто программой называют также любой набор инструкций для компьютера.

Важным также является выбор языка программирования. Большинство профессиональных программ сейчас составляется на языке *Си* (точнее, на его расширении *Си++*), реже – на *Visual Basic* или *Delphi*.

📖 **Отладка программы**

Отладка – это поиск и исправление ошибок в программе.

На английском языке отладка называется **debugging**, что дословно означает «извлечение жучков». Легенда утверждает, что на заре развития компьютеров (в 1940 г.) в контакты электронных реле компьютера МАРК-II залетела моль (*bug*), из-за чего он перестал работать.

Для отладки программ используются специальные программы, которые называются **отладчиками**. Они позволяют

- выполнять программу в пошаговом режиме (останавливаясь после каждой выполненной команды);
- устанавливать **точки прерывания**, так что программа останавливается каждый раз, когда переходит на заданную строчку;
- просматривать и изменять значения переменных в памяти, а также регистры процессора.

При отладке можно также использовать **профайлер** – программу, которая позволяет определить, сколько времени выполняется каждая часть программы. При этом в первую очередь имеет смысл оптимизировать те процедуры, которые выполняются дольше всех.

📖 **Разработка документации**

Очень важно правильно разработать документацию, связанную с программой. Чаще всего требуется руководство пользователя (*User manual*), иногда (когда поставляются исходные тексты программы) – руководство программиста (*Programmer's manual*), подробное описание алгоритма и особенностей программы. Эти документы надо составлять на простом и понятном языке, так чтобы их мог читать не только автор программы.

📖 **Тестирование программы**

Тестирование – это проверка программы специально обученными людьми – *тестерами*.

Цель тестирования – найти как можно больше ошибок, которые не смог найти разработчик. Различают две ступени тестирования профессиональных программ:

- **альфа-тестирование** – это тестирование сотрудниками той же фирмы, в которой работает команда разработчиков;
- **бета-тестирование** – тестирование предварительной версии в других фирмах и организациях; часто бета-версии программ для тестирования свободно распространяются через Интернет.

Программу условно можно считать правильной, если её запуск для выбранной системы тестовых исходных данных во всех случаях дает правильные результаты.

Но, как справедливо указывал известный теоретик программирования Э. Дейкстра, «*тестирование может показать лишь наличие ошибок, но не их отсутствие*». Нередки случаи, когда новые входные данные вызывают «отказ» или приводят к неверным результатам работы программы, которая считалась полностью отлаженной.

Для тестирования нужно заранее подготовить эталонные примеры с известными результатами. Вычислять их обязательно *до*, а не *после* получения машинных результатов, иначе есть опасность невольной подгонки вычисляемых значений под желаемые, полученные ранее на компьютере.

Сопровождение

Сопровождение – это исправление ошибок в программе после ее сдачи заказчику и консультации для пользователей.

Современный подход к продаже программ сводится к тому, что продается не программа, а техническая поддержка, то есть автор обязуется (за те деньги, которые ему были выплачены) незамедлительно исправлять ошибки, найденные пользователями, и отвечать на их вопросы.

Программирование «снизу вверх»

Раньше этот подход был основным при составлении программ. Программу начинают разрабатывать с самых простых процедур и функций. Таким образом, мы учим компьютер решать все более и более сложные задачи, которые станут частями нашей главной задачи. Построив процедуры нижнего уровня, мы собираем из них, как из кубиков, более крупные процедуры и так далее. Конечная цель – собрать из кубиков всю программу.

Достоинства:

- легко начать составлять программу «с нуля»;
- используя метод «от простого – к сложному» можно написать наиболее эффективные процедуры.

Недостатки:

- при составлении простейших процедур необходимо как-то связывать их с общей задачей;
- может так случиться, что при окончательной сборке не хватит каких-то кубиков, о которых мы не подумали раньше;
- программа часто получается запутанной.

Структурное программирование

Программирование «сверху вниз»

Следующим шагом вперед в программировании стал подход «сверху вниз», при котором программа проектируется по принципу «от общего к частному». Этот метод часто называется **методом последовательной детализации**.

Сначала задача разбивается на более простые подзадачи и составляется основная программа. Вместо тех процедур, которые еще не реализованы, пишутся «*заглушки*» – процедуры и функции, которые ничего не делают, но имеют заданный список параметров. Затем каждая подзадача разбивается на еще более мелкие подзадачи и так далее, пока все процедуры не будут реализованы.

Достоинства:

- поскольку мы начинаем с главного – общей задачи, меньше вероятность принципиальной ошибки;

- структура программы получается простой и понятной.

Недостатки:

- может получиться так, что в разных блоках потребуется реализовать несколько похожих процедур, вместо которых можно было бы обойтись одной

Цели структурного программирования

Структурное программирование появилось потому, что программы становились все сложнее и сложнее, и требовались новые методы, чтобы создавать и отлаживать их в короткие сроки. Надо было решить следующие задачи:

- повысить **надежность программ**; для этого нужно, чтобы программа легко поддавалась тестированию и не создавала проблем при отладке;
- повысить **эффективность программ**; сделать так, чтобы текст любого модуля можно было переделать независимо от других для повышения эффективности его работы;
- уменьшить **время и стоимость разработки** сложных программ;
- улучшить **читабельность программ**; это значит, что необходимо избегать использования запутанных команд и конструкций языка; надо разрабатывать программу так, чтобы ее можно было бы читать от начала до конца без управляющих переходов на другую страницу.

Принципы структурного программирования

Принцип абстракции. Программа должна быть спроектирована так, чтобы ее можно было рассматривать с различной степенью детализации, без лишних подробностей.

Принцип модульности. В соответствии с этим принципом программа разделяется на отдельные законченные простые фрагменты (модули), которые можно отлаживать и тестировать независимо друг от друга. В результате отдельные части программы могут создаваться разными группами программистов.

Принцип подчиненности. Взаимосвязь между частями программы должна носить подчиненный характер. К этому приводит разработка программы «сверху вниз».

Принцип локальности. Надо стремиться к тому, чтобы каждый модуль (процедура или функция) использовал свои (локальные) переменные и команды. Желательно не применять глобальные переменные.

Структурные программы

Программа, разработанная в соответствии с принципами структурного программирования, должна удовлетворять следующим требованиям:

- программа должна разделяться на независимые части, называемые модулями;

Модуль – это независимый блок, код которого физически и логически отделен от кода других модулей; модуль выполняет только одну логическую функцию, иначе говоря, должен решать самостоятельную задачу своего уровня по принципу: один программный модуль – одна функция.

- работа программного модуля не должна зависеть:
 - от того, какому модулю предназначены его выходные данные;
 - от того, сколько раз он вызывался до этого;

- размер программного модуля желательно ограничивать одной-двумя страницами исходного листинга (30-60 строк исходного кода);
- модуль должен иметь только одну входную и одну выходную точку;
- взаимосвязи между модулями устанавливаются по принципу подчиненности;
- каждый модуль должен начинаться с комментария, объясняющего его назначение, назначение переменных, передаваемых в модуль и из него, модулей, которые его вызывают, и модулей, которые вызываются из него;
- оператор безусловного перехода или вообще не используется в модуле, или применяется в исключительных случаях только для перехода на выходную точку модуля;
- в тексте модуля необходимо использовать комментарии, в особенности в сложных местах алгоритма;
- имена переменных и модулей должны быть смысловыми, «говорящими»;
- в одной строке не стоит записывать более одного оператора; если для записи оператора требуется больше, чем одна строка, то все следующие операторы записываются с отступами;
- желательно не допускать вложенности более, чем трех уровней;
- следует избегать использования запутанных языковых конструкций и программистских «трюков».

Все эти вместе взятые меры направлены на повышение качества программного обеспечения.

2. Построение графиков и работа с ними

Структура программы

Чтобы проиллюстрировать методику создания больших программ, мы рассмотрим достаточно сложный проект, в котором необходимо выполнить следующие части задания:

- построить на экране оси координат и сделать их разметку;
- построить графики заданных двух функций;
- найти координаты указанных точек пересечения;
- заштриховать замкнутую область, ограниченную кривыми;
- вычислить площадь заштрихованной части.

Такую работу сложно выполнить сразу, за один день. Разработку большой программы обычно начинают с того, что задача разбивается на несколько более простых подзадач. Для нашей работы такие подзадачи перечислены выше. Будем считать, что каждую их подзадач выполняет специальная процедура. Таким образом, можно сразу определить структуру всей программы и составить основную программу.

```
#include <stdio.h> // стандартный ввод и вывод
#include <graphics.h> // графические функции
#include <math.h> // математические функции

// здесь объявляются константы и глобальные переменные

// здесь записываются все функции и процедуры

//-----
//      Основная программа
//-----
main ()
{
    initwindow ( 800, 600 ); // создать окно для графики
    Axes (); // построение и разметка осей координат
    Plot (); // построение графиков
    Cross (); // поиск точек пересечения
    Hatch (); // штриховка
    Area (); // вычисление площади
    getch (); // ждать нажатия на клавишу
    closegraph (); // закрыть окно для графики
}
```

Таким образом, основная программа состоит только из вызовов процедур. Этих процедур еще нет, мы будем писать их последовательно одну за другой. Чтобы иметь возможность отлаживать программу по частям, прокомментируем строчки с вызовами процедур, поставив в начале каждой строки две наклонные черточки (два слэша) //. После того, как очередная процедура составлена и записана (выше основной программы), надо убрать эти символы в соответствующей строчке основной программы, иначе она не будет вызываться.

Оформление программы

Чтобы легче было разбираться в программе, используют специальные приемы оформления. Они не влияют на выполнение программы, но позволяют легче искать ошибки и вносить исправления. Можно выделить следующие принципы.

- Каждая процедура и функция должна иметь заголовок («шапку»), в котором указывают
 - название функции;
 - описание ее работы;
 - входные и выходные параметры, для функции – возвращаемое значение;
 - процедуры и функции, которые она вызывает;
 - процедуры и функции, которые вызывают ее.

Простейший пример «шапки» показан выше при описании структуры программы.

- Тело каждого цикла **for**, **while**, **do-while**, включая фигурные скобки, смещается вправо на 2-3 символа относительно заголовка, так чтобы начало и конец цикла были хорошо заметны. Открывающая скобка должна быть на том же уровне, что и закрывающая. Это позволяет сразу обнаруживать непарные скобки.

```
while ( a < b )
{
  // тело цикла
}
```

- Аналогичные отступы делаются в условных операторах **if-else** и операторе выбора **switch**.

```
if ( a < b )
{
  // блок «если»
}
else
{
  // блок «иначе»
}
```

```
switch ( k )
{
  case 1: // блок «1»
  case 2: // блок «2»
  default: // по умолчанию
}
```

- Процедуры и функции должны иметь осмысленные имена, так чтобы можно было сразу вспомнить, что они делают.
- Надо давать переменным осмысленные имена, чтобы их роль была сразу понятна. Так счетчик можно назвать **count**, переменную, в которой хранится сумма – **summa** и т.п.
- Чтобы облегчить чтение программы, полезно использовать пробелы в записи операторов, например

```
while ( a < b ) { /* операторы */ }
```

воспринимается легче, чем эта же строка без пробелов.

```
while(a<b){/*операторы*/}
```

- Наиболее важные блоки в процедурах и функциях отделяют пустыми строками или строками-разделителями, состоящими, например, из знаков «минус».



Способы задания функций

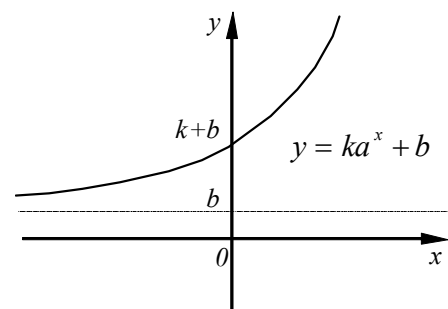
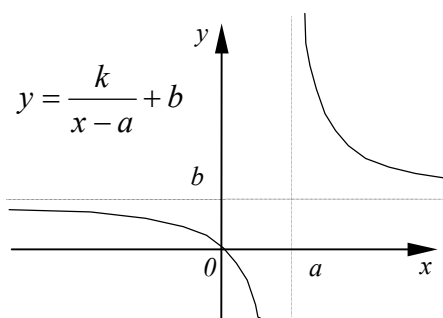
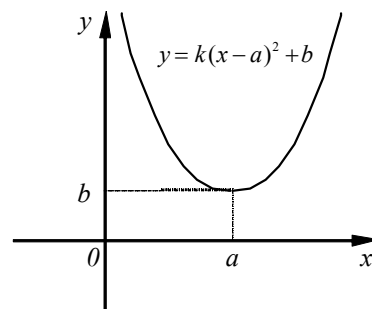
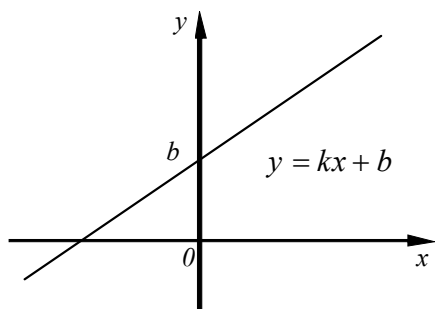
В этом разделе рассматриваются различные способы задания кривых на плоскости, используемые в математике.



Функции, заданные в явном виде

Будем считать, что независимой переменной является **x**, а значением функции — **y**. В простейшем случае известна зависимость **y = f(x)**, то есть, зная **x**, можно однозначно опреде-

литель соответствующее ему значение y (каждому x соответствует только одно значение y). К числу простейших графиков функций относятся прямая, парабола, гипербола, показательная функция.



Функции, заданные в неявном виде

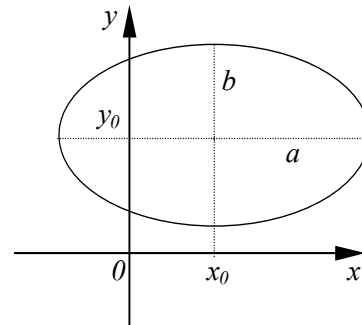
Не все функции могут быть заданы в явном виде, например те, для которых одному значению x соответствует несколько значений y . В этом случае функцию задают в виде

$$f(x, y) = 0$$

Простейшим примером может служить эллипс или его частный случай — окружность. Уравнение эллипса (которое математики называют **каноническим**) записывается так:

$$\frac{(x - x_0)^2}{a^2} + \frac{(y - y_0)^2}{b^2} = 1,$$

где x_0 и y_0 — координаты центра эллипса, а a и b — длины его полуосей. В частном случае, когда $a = b$, эллипс превращается в окружность.



Часто можно представить такую кривую как комбинацию двух или нескольких кривых, каждая из которых задается в явном виде. Например, для окружности можно выразить y в виде

$$y = y_0 \pm b \sqrt{1 - \frac{(x - x_0)^2}{a^2}}$$

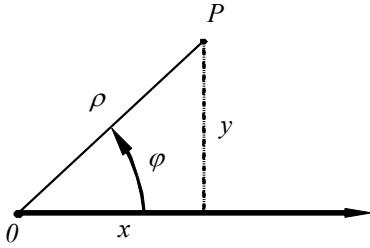
Верхняя часть эллипса соответствует знаку «плюс» в этой формуле, а нижняя — знаку «минус».

При построении эллипсов и окружностей на экране важно проверять область определения (область допустимых значений аргумента). Например, в предыдущей формуле выражение под знаком корня должно быть неотрицательно, поэтому $(x - x_0)^2 \leq a^2$, или $x_0 - a \leq x \leq x_0 + a$.

Функции, заданные в полярных координатах

Для некоторых сложных кривых удастся получить простые выражения, если использовать **полярную систему координат**, в которой для отсчета используется точка O — **полюс** (начало координат) и **полярная ось** — луч, выходящий из полюса горизонтально вправо — от него от-

считывается угол. Любая точка P на плоскости имеет две координаты: расстояние от этой точки до полюса ρ и полярный угол φ между полярной осью и лучом OP . Угол считается положительным при отсчете от полярной оси против часовой стрелки.



Преобразования координат от декартовой системы к полярной и наоборот очевидны:

$$\begin{cases} \rho = \sqrt{x^2 + y^2} \\ \varphi = \arctan \frac{y}{x} \end{cases} \quad \begin{cases} x = \rho \cos \varphi \\ y = \rho \sin \varphi \end{cases}$$

Например, уравнение окружности с центром в начале координат выглядит очень просто в полярной системе координат: $\rho = R$. Как видим, радиус не зависит от угла поворота.

В полярных координатах просто задаются спирали различного типа. Например, спираль Архимеда имеет уравнение $\rho = a\varphi$, а логарифмическая спираль задается формулой $\rho = \frac{a}{\varphi}$. В этих формулах a – некоторая постоянная.

📖 **Функции, заданные в параметрическом виде**

Для функций, заданных в параметрическом виде, соответствующие друг другу x и y выражаются через третью переменную t , которая называется **параметром**:

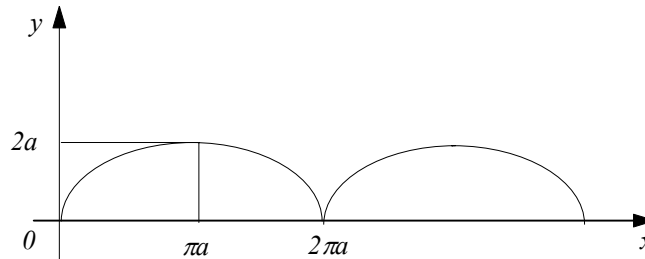
$$\begin{cases} x = f_1(t) \\ y = f_2(t) \end{cases}$$

Например, уравнение эллипса с центром в начале координат записывается в параметрическом виде так

$$\begin{cases} x = a \cos t \\ y = b \sin t \end{cases}$$

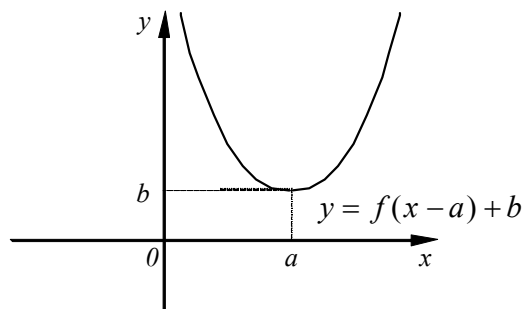
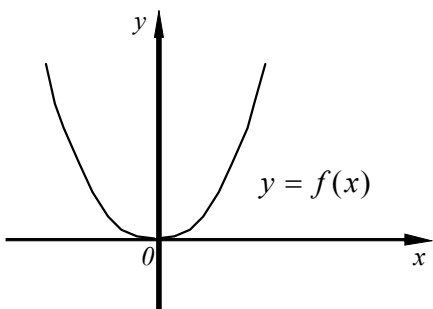
В такой форме очень просто задается, например, **циклоида** — кривая, которую описывает точка катящегося колеса. При $\lambda=1$ получаем классическую циклоиду (на рисунке), при $\lambda < 1$ — укороченную, а при $\lambda > 1$ — удлинненную.

$$\begin{cases} x = a(t - \lambda \sin t) \\ y = a(1 - \lambda \cos t) \end{cases}$$

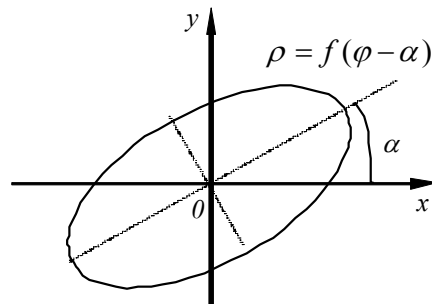
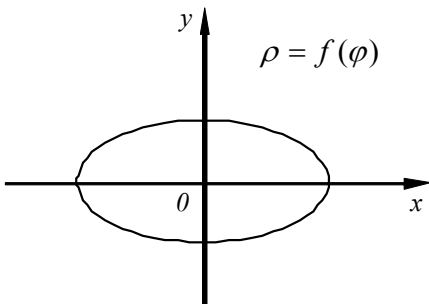


📖 **Преобразование координат**

Простейшим видом преобразования является **параллельный перенос**. Для того, чтобы сместить график на a по оси x и на b по оси y , надо вычислять значение функции в точках $x-a$ и прибавлять к результату b .



Другим простейшим преобразованием является **поворот** на некоторый угол α . Здесь удобнее использовать полярные координаты. Действительно, при повороте против часовой стрелки надо рассчитать значение $\rho = f(\varphi - \alpha)$ и соответствующие координаты \mathbf{x} и \mathbf{y} .



Системы координат

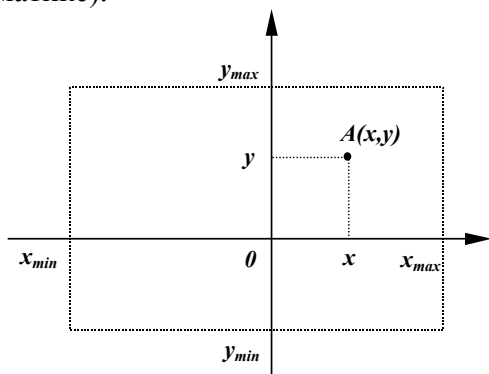


Системы координат и обозначения

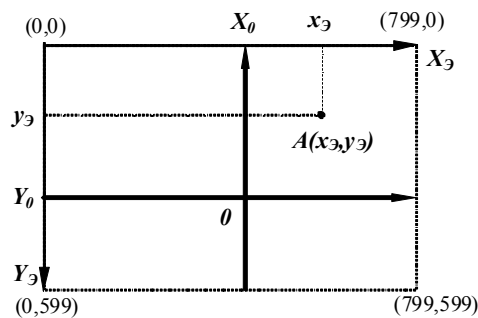
При построении графиков функций на компьютере используются две системы координат:

- **математическая** система координат, в которой задана исходная функция;
- **экранная** система координат.

Между ними есть много принципиальных различий. В математической системе плоскость считается бесконечной, на экране мы можем отобразить только прямоугольную часть плоскости. Кроме того, для большинства систем ось \mathbf{x} в экранной системе координат направлена вдоль верхней границы экрана влево, а ось \mathbf{y} — вдоль левой границы *вниз* (а не вверх, как принято в математике).



математические координаты



экранные координаты

Будем считать, что окно для вывода графики имеет размеры 800 пикселей в ширину и 600 — в высоту. Для других размеров надо будет просто заменить числа в формулах. Можно представить себе, что мы смотрим на плоскость через окно и видим ее прямоугольную часть, ограниченную значениями \mathbf{x}_{\min} и \mathbf{x}_{\max} по оси \mathbf{x} и значениями \mathbf{y}_{\min} и \mathbf{y}_{\max} по оси \mathbf{y} . Далее экранные координаты (в пикселях) точки $\mathbf{A}(\mathbf{x}, \mathbf{y})$ мы будем обозначать через $\mathbf{x}_э$ и $\mathbf{y}_э$.

Масштабы и преобразования координат

Для того, чтобы построить на экране оси координат, выберем значения X_0 и Y_0 , показанные на рисунке – координаты точки $(0,0)$ на экране. Для того, чтобы иметь возможность увеличивать и уменьшать график, надо выбрать **масштабный коэффициент k** , который будет показывать, во сколько раз увеличивается каждый отрезок на экране в сравнении с математической системой координат. Фактически k – это длина единичного отрезка в экранной системе координат.

Зная X_0 , Y_0 и k , можно рассчитать границы видимой области по формулам

$$x_{\min} = -\frac{X_0}{k}, \quad x_{\max} = \frac{(800 - X_0)}{k},$$

$$y_{\min} = -\frac{(600 - Y_0)}{k}, \quad y_{\max} = \frac{Y_0}{k},$$

В программе многие функции будут использовать значения X_0 , Y_0 и k , поэтому их лучше сделать глобальными константами (постоянными). Обратите внимание, что X_0 и Y_0 – целые числа (расстояния в пикселях), а масштаб k может быть дробным, поэтому нужно объявить их так:

```
const int X0 = 100, Y0 = 400;
const float k = 3.5;
```

Теперь надо выяснить, как же преобразовать координаты точки $A(x, y)$ так, чтобы получить соответствующие экранные координаты $(x_э, y_э)$. Каждый отрезок оси x растягивается в k раз и смещается на X_0 относительно левого верхнего угла экрана. Для оси y – все почти так же, но надо учесть еще, что ось направлена в **обратном направлении** — вниз. Поэтому в формуле появляется знак минус. Таким образом, формулы для преобразования координат из математической системы в экранную принимают вид:

```
X = X0 + k * x;
Y = Y0 - k * y;
```

Такой же результат можно получить, если рассмотреть пропорции между соответствующими отрезками на плоскости и на экране.

Дальше нам нужно будет применять эти преобразования координат много раз, поэтому удобно оформить отдельные функции и расположить их выше основной программы:

```
//-----
// SCREENX - перевод X в координаты экрана
//-----
int ScreenX (float x)
{
    return X0+k*x;
}
//-----
// SCREENY - перевод Y в координаты экрана
//-----
int ScreenY (float y)
{
    return Y0-k*y;
}
```

Обратите внимание, что эти функции принимают вещественные параметры (координаты в математической системе), а возвращают целые результаты (экранные координаты в пикселях). Округление (отбрасывание дробной части) в языке Си выполняется автоматически.

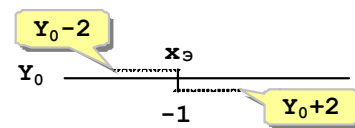
Оси координат

С учетом сказанного выше процедура для рисования осей может выглядеть так:

```
void Axes ()
{
    line ( X0, 0, X0, 599 ); // вертикальная ось
    line ( 0, Y0, 799, Y0 ); // горизонтальная ось
}
```

Теперь осталось добавить в эту процедуру разметку осей координат — надо нанести на осях деления и подписи. Покажем, как это выполняется на примере разметки положительной части оси x с шагом 1. Так как длина единичного отрезка на экране равна k пикселей, то на положительной части оси x будет $(800-X_0)/k$ отметок (не считая нуля). Координата $x_э$ (на экране) отметки, соответствующей значению $x = i$ (в математической системе) равна X_0+i*k .

Теперь надо сделать отметку на оси (черточку) и надпись. Часть оси с одной отметкой и надписью показана слева. При известном x надо провести вертикальный отрезок и вывести значение соответствующего ему x .



Если нарисовать отрезок несложно, для вывода надписи придется потрудиться, поскольку в графическом режиме можно выводить на экран только символьные строки, но никак не целые или вещественные числа. Для того, чтобы сделать то, что мы хотим, надо сначала преобразовать число в строку, а затем эту строку вывести на экран. Первая часть выполняется с помощью функции **sprintf**. Она очень напоминает **printf** и **fprintf**, но выводит информацию не на экран и не в файл, а направляет вывод в указанную строку. Конечно, под нее надо заранее выделить место в памяти. В программе это может выглядеть примерно так:

```
int i;
char s[10]; // строка на 10 символов
sprintf ( s, "%d", i ); // записать результат в строку s
```

Как видим, единственная особенность функции **sprintf** состоит в том, что первым параметром задается строка, в которую записывается результат.

Второй шаг выполняется с помощью функции **outtextxy**, в параметрах которой передаются координаты левого верхнего угла текста и сама символьная строка. Одна цифра занимает ячейку 8×13 пикселей. Будем считать, что надпись занимает два символа, то есть прямоугольник 16×13 . Чтобы надпись не наезжала на отметку, левый верхнюю границу надписи надо немного сместить вниз.

```
outtextxy ( xe-8, Y0+4, s );
```

Теперь мы готовы написать всю процедуру построения и разметки осей до конца. Для разметки с шагом 1 надо перебрать все целые значения i между 0 и $(800-X_0)/k$, и для каждого из них сделать отметку на оси и надпись.

```
void Axes ()
```

```

{
int i, xe;
char s[10];
line ( X0, 0, X0, 599 ); // рисуем оси
line ( 0, Y0, 799, Y0 );

for ( i=0; i<= (800-X0)/k; i++ ) // цикл по всем делениям
{
xe = ScreenX ( i );
line ( xe, Y0-2, xe, Y0+2 ); // рисуем деление
sprintf ( s, "%d", i ); // число - в строку
outtextxy ( xe-8, Y0+4, s ); // вывод числа
}

// а здесь надо написать разметку остальных осей
}

```

Построение графиков

Стандартный способ

Построим на экране график функции, заданной в явном виде $y = f_1(x)$, вернее, ту его часть, которая попадает в выбранный прямоугольник. Вычисление y по известному x мы оформим в виде отдельной функции, которая принимает единственный вещественный параметр и возвращает вещественное число — значение функции в этой точке. Сама функция может быть сколь угодно сложная, например, функция $y = \sqrt{x}$ записывается так:

```

#include <math.h>
float F1 ( float x )
{
return sqrt(x); // необходимо подключить <math.h>
}

```

Чтобы использовать математическую функцию `sqrt` (она вычисляет квадратный корень числа), нужно подключить заголовочный файл `math.h` в начале программы.

Теперь для всех x из интервала $[x_{\min}, x_{\max}]$ надо построить соответствующие точки графика. Однако всего в этом интервале бесконечное множество точек, поэтому на практике придется выбрать конечное множество, например, задать изменение x внутри интервала с шагом h . Каким должен быть шаг?

Поскольку на экране только 800 вертикальных линий, то чаще всего не имеет смысла делать более 800 шагов (для графиков, которые резко уходят вверх или вниз, может понадобиться в 2-3 раза больше). Учитывая, что

$$x_{\min} = -\frac{X_0}{k}, \quad \text{и} \quad x_{\max} = \frac{(800 - X_0)}{k},$$

$$\text{выбираем } h = \frac{x_{\max} - x_{\min}}{800} = \frac{1}{k}.$$

При построении надо проверять область определения функции, чтобы не получить ошибку при делении на нуль, извлечении корня из отрицательного числа и т.п. Для сложных функций лучше оформить свою функцию на языке Си, которая будет возвращать целое число: единицу, если заданное x входит в область определения, и нуль — если не входит. Для нашей функции она может выглядеть так (приводится 2 способа):

```
int ODZF1 ( float x )
{ if ( x != 0 ) return 1;
  else      return 0;
}
```

```
int ODZF1 ( float x )
{
  return (x != 0);
}
```

Справа записан более профессиональный вариант.

Строить точку на экране надо только тогда, когда она попадает на экран. Если пытаться строить точку, координаты которой находятся вне экрана, мы сотрем какую-то ячейку памяти, и результат может быть непредсказуем. Поэтому будем использовать специальную процедуру **Point**, которая переводит переданные ей значения координат **x** и **y** из математической системы в экранную и, если точка попадает на экран, выводит точку заданного цвета **color**. Эту процедуру удобнее всего разместить выше процедур рисования графиков (**PlotF1** и **PlotF2**), чтобы на момент вызова она была уже известна.

```
void Point ( float x, float y, int color )
{
  int xe, ye;
  xe = ScreenX(x);
  ye = ScreenY(y);
  if ( xe >= 0 && xe < 800 && ye >= 0 && ye < 600 )
    putpixel(xe, ye, color);
}
```

Теперь, используя приведенные функции, можно составить процедуру, которая рисует график функции на экране:

```
void Plot()
{
  float x, h, xmin, xmax;
  h = 1 / k;
  xmin = - X0 / k;
  xmax = (800 - X0) / k;
  for ( x = xmin; x <= xmax; x += h )
    if ( ODZF1(x) ) Point(x, F1(x), RED);
}
```

Если надо построить несколько графиков, это можно сделать в одном цикле, вызывая каждый раз соответствующие функции для вычисления значения **y** и проверки области определения.

📄 **Функции, заданные в полярных координатах**

Если функция задана в полярных координатах, независимой переменной в цикле будет не координата **x**, а угол поворота радиус-вектора φ (угол между горизонтальной осью и вектором из начала координат в данную точку). Надо иметь в виду, что угол измеряется в радианах. Угол 90 градусов равен $\pi/2$ радиан (в языке Си константа π обозначается **M_PI**). Диапазон углов от 0 до 2π радиан (360 градусов) соответствует одному обороту вокруг начала координат.

Если требуется, также надо использовать область определения функции (чтобы избежать деления на нуль и извлечения корня из отрицательного числа). Шаг изменения угла подбирается опытным путем так, чтобы не было видно, что линия состоит из отдельных точек.

```
void PlotPolar ()
```

```

{
    float phi, x, y, ro,
           phiMin = 0.,           // минимальный угол
           phiMax = 2*M_PI,      // максимальный угол
           h = 0.001;           // шаг измерения угла

    for ( phi = phiMin; phi <= phiMax; phi += h )
        if ( ODZF1(phi) ) { // проверяем ОДЗ
            ro = F1(phi); // функция в полярных координатах
            x = ro*cos(phi); // пересчет полярных координат
            y = ro*sin(phi); // в декартовы
            Point(x, y, RED); // ставим красную точку на экране
        }
}

```

📄 **Функции, заданные в параметрическом виде**

Если функция задана в параметрическом виде, независимой переменной в цикле будет параметр **t**. Диапазон его изменения (значения переменных **tMin** и **tMax** в программе) надо подбирать по справочнику или экспериментально. Шаг изменения параметра **t** также подбирается опытным путем. Для каждого значения **t** вычисляются **x** и **y** (предположим, что это делают функции **F_x(t)** и **F_y(t)**). Общий порядок действия аналогичен предыдущим случаям.

```

void PlotParameter ()
{
    float t, tMin = -10., tMax = 10.,
          h = 0.001, x, y;

    for ( t = tMin; t <= tMax; t += h )
        if ( ODZF1(t) ) {
            x = Fx(t);
            y = Fy(t);
            Point (x, y, RED);
        }
}

```

📄 **Как найти точки пересечения?**

На следующем этапе работы надо найти точки пересечения графиков. Если даны две функции **f₁(x)** и **f₂(x)**, координаты точек их пересечения по оси **x** удовлетворяют уравнению

$$f_1(x) = f_2(x)$$

К сожалению, точно решить это уравнение аналитически можно только в некоторых простейших случаях. Если функции нелинейные, чаще всего аналитического решения не существует. При этом задачу решать надо, и используются **численные методы**, которые всегда являются **приближенными**. Это означает, что мы всегда получаем решение с некоторой **ошибкой**, хотя она может быть очень и очень маленькой (даже одна миллионная и меньше).

📄 **Метод перебора**

Примитивный метод решения уравнения заключается в следующем. Пусть мы знаем, что на интервале **[a, b]** графики имеют одну и только одну точку пересечения и требуется найти координату точки пересечения **x*** с точность **ε**.

Разобьем интервал $[a, b]$ на кусочки длиной ε и найдем из них такой отрезок $[x^*_1, x^*_2]$, на котором уравнение имеет решение, то есть линии пересекаются. Для этого перебираем все возможные x от a до b и проверяем для каждого значение произведения

$$g_1 = f_1(x) - f_2(x), \quad g_2 = f_1(x+\varepsilon) - f_2(x+\varepsilon).$$

Если функции g_1 и g_2 имеют разные знаки, значит на интервале $[x, x+\varepsilon]$ есть решение уравнения.

Можно поступать: найти такое x , для которого величина функции g_1 (то есть разность двух исходных функций) минимальна, и принять его за приближенное решение.

📖 Метод деления отрезка пополам

Один из самых простых численных методов решения уравнений — **метод деления пополам** или **дихотомия**. Он связан со старинной задачей о поиске льва в пустыне. Надо разделить пустыню на 2 равные части и определить, в которой из них находится лев. Затем эту половину снова делим на две части и т.д. В конце концов, делить станет нечего — лев вот он!

Пусть мы знаем, что на интервале $[a, b]$ графики имеют одну и только одну точку пересечения (это важно!). Идея метода такова: найдем середину отрезка $[a, b]$ и назовем ее c . Теперь проверяем, есть ли пересечение на участке $[a, c]$. Если есть, ищем дальше на этом интервале, а если нет, то на интервале $[c, b]$.

Заметим, что если графики пересекаются на отрезке $[a, c]$, то разность этих функций меняет знак — если в точке a функция $f_1(x)$ проходит выше, чем $f_2(x)$, то в точке c — наоборот. Это можно выразить условием

$$(f_1(a) - f_2(a)) \cdot (f_1(c) - f_2(c)) < 0.$$

Если оно верно, то на участке $[a, c]$ есть пересечение.

Когда же закончить деление отрезка пополам? Видимо тогда, когда мы достигнем заданной точности ε , то есть когда ширина отрезка $[a, b]$ станет меньше ε . Вот функция для вычисления корня с заданной точностью (точность задается параметром **eps**):

```
//-----
// Solve находит точку пересечения на [a,b]
// Вход: a, b - границы интервала, a < b
//       eps - точность решения
// Выход: x - решение уравнения f1(x)=f2(x)
//-----
float Solve( float a, float b, float eps )
{
    float c, fa, fc;

    while( fabs(b-a) > eps ) { // пока не достигли заданной точности
        c = (a + b) / 2.; // середина отрезка [a,b]
        fa = F1(a) - F2(a); // разность функций в точке x=a
        fc = F1(c) - F2(c); // разность функций в точке x=c
        if ( fa*fc < 0 ) b = c; // сужаем область поиска
        else a = c;
    }
}
```

```
return (a + b) / 2.; // результат - середина отрезка
}
```

Она возвращает приближенное значение координаты x точки пересечения. После этого можно рассчитать координату y , подставив полученное x в одну из формул, и вывести обе координаты на экран. Ниже приведена процедура **Cross**, которая это делает.

```
void Cross ()
{
    float y;
    int xe, ye;
    char s[30];
    x1 = Solve(1, 2, 0.001); // найти x-координату точки
    y = F1(x1);             // найти y-координату точки
    xe = ScreenX(x1);       // вычислить экранные координаты
    ye = ScreenY(y);
    sprintf(s, "x1:%5.2f", x1 ); // вывод координат на экран
    outtextxy(xe+5, ye+2, s);
    sprintf(s, "y1:%5.2f", y );
    outtextxy(xe+5, ye+22, s);
}
```

Здесь $x1$ – это *глобальная* переменная, в которую мы записываем координату первой точки пересечения. Если есть еще и вторая точка пересечения, нужно ввести еще одну глобальную переменную, $x2$. Они должны быть объявлены выше всех процедур (после констант) так:

```
float x1, x2;
```

Эти переменные удобно сделать глобальными, потому что они далее будут использоваться в нескольких процедурах.

У функции **Solve** есть один недостаток — она жестко завязана на функции $f_1(x)$ и $f_2(x)$. Хорошо бы сделать так, чтобы в параметрах этой функции можно было передать **адреса функций**, которые надо вызывать.

Этого можно добиться, если объявить новый тип данных — **указатель на функцию**. Единственное ограничение — все функции, которые передаются в процедуру, должны быть одного типа, то есть принимать одинаковые аргументы (в нашем случае — **float x**) и возвращать значения одинаковых типов (в нашем случае — **float**). Объявить новый тип (назовем его **func**) можно в начале программы с помощью команды **typedef**

```
typedef float (*func)( float x );
```

Это достаточно сложное определение говорит о том, что теперь введен новый тип данных — указатель на функцию, которая принимает один вещественный аргумент и возвращает вещественный результат, и тип этот называется **func**. Тогда функция **Solve** может выглядеть так (показаны только изменившиеся строчки):

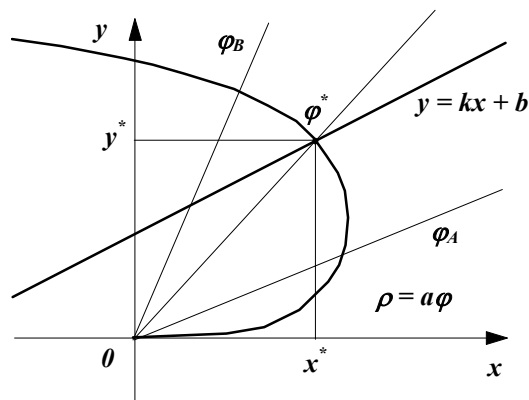
```
float Solve( func ff1, func ff2,
            float a, float b, float eps )
{
    ...
    fa = ff1(a) - ff2(a);
    fb = ff1(b) - ff2(b);
    ...
}
```

При вызове этой функции в первых двух параметрах указывают имена функций, которые надо использовать, например, так

```
x1 = Solve( F1, F2, 1., 2., 0.0001 );
```

📖 **Функции, заданные в полярных координатах**

Возможна такая ситуация, когда одна из кривых задана в полярных координатах, а вторая — в явном виде. В этом случае задача определения координат точек пересечения становится несколько сложнее.



Покажем принцип решения задачи на примере. Пусть требуется найти пересечение спирали $\rho = a\varphi$ и прямой $y = kx + b$. Мы будем искать точку пересечения методом деления отрезка пополам. Сначала построим оба графика и определим углы φ_A и φ_B , между которыми находится точка пересечения графиков. В данном случае можно принять $\varphi_A = 0$ и $\varphi_B = \pi/2$. Наша задача — определить угол φ^* и соответствующие ему декартовы координаты x^* и y^* .

Заметим, что при $\varphi < \varphi^*$ спираль проходит ниже прямой, поэтому если мы рассчитаем для некоторого φ значение ρ и соответствующие ему x_ρ и y_ρ , а также ординату прямой $y_1 = kx_\rho + b$, то выполнится условие $y_\rho < y_1$ или $y_\rho - y_1 < 0$. Аналогично при $\varphi > \varphi^*$ спираль проходит выше прямой, поэтому выполнится условие $y_\rho > y_1$ или $y_\rho - y_1 > 0$.

Таким образом, можно применить метод отрезка деления пополам, поскольку разность $y_\rho - y_1$ меняет знак при переходе φ через φ^* . Общую идею можно записать в виде следующего алгоритма:

- определить интервал углов $[\varphi_A, \varphi_B]$, в котором находится угол φ^* , соответствующий точке пересечения;
- найти середину интервала — угол $\varphi_C = (\varphi_A + \varphi_B) / 2$;
- вычислить $\rho(\varphi_A)$ и соответствующие $y_\rho(\varphi_A)$ и $y_1(\varphi_A)$;
- вычислить $\rho(\varphi_C)$ и соответствующие $y_\rho(\varphi_C)$ и $y_1(\varphi_C)$;
- если значения $y_\rho(\varphi_A) - y_1(\varphi_A)$ и $y_\rho(\varphi_C) - y_1(\varphi_C)$ имеют разные знаки (есть пересечение), повторить поиск на интервале $[\varphi_A, \varphi_C]$, иначе искать на $[\varphi_C, \varphi_B]$;
- поиск заканчивается, когда длина интервала $[\varphi_A, \varphi_B]$ будет меньше заданной точности.

Для вычисления x и y по заданному углу φ для функции, заданной в полярных координатах, удобно использовать процедуру. Она должна вернуть в виде результата два значения, которые можно передать через **изменяемые параметры** (перед именем которых в заголовке функции стоит знак **&**):

```
void Spiral( float phi, float &x, float &y )
{
    float rho, a = 1;
    rho = a * phi; // вычисляем полярный радиус
    x = rho * cos(phi); // вычисляем декартовы координаты
    y = rho * sin(phi);
}
```

Эту процедуру надо вызвать два раза — один раз для угла ϕ_A , второй раз — для ϕ_C . Полностью функция, вычисляющая координаты x и y точки пересечения приведена ниже. Точность ϵ определяется как диапазон углов $[\phi_A, \phi_B]$, при котором процесс прекращается.

```
void SolvePolar( float phiA, float phiB,
                float eps, float &x, float &y )
{
float phiC, xA, yA, xC, yC, fA, fC;
while ( fabs(phiB-phiA) > eps ) {
    phiC = (phiA + phiB) / 2.; // средний угол
    Spiral ( phiA, xA, yA ); // (xA,yA) для угла phiA
    Spiral ( phiC, xC, yC ); // (xC,yC) для угла phiC
    fA = yA - F1(xA); // разности для угла phiA
    fC = yC - F1(xC); // разности для угла phiC
    if ( fA*fC < 0 ) phiB = phiC; // сужаем интервал поиска
    else phiA = phiC;
}
x = xA;
y = yA;
}
```

📄 Функции, заданные в параметрическом виде

Пусть теперь одна кривая задана в параметрической форме, а вторая — в явном виде. Независимой переменной является параметр t . Поэтому, построив графики, надо определить интервал значений параметра $[t_A, t_B]$, внутри которого находится корень уравнения. Далее корень находим методом деления пополам **диапазона изменения параметра**. Пусть функции $F_x(t)$ и $F_y(t)$ вычисляют значения координат параметрической кривой для заданного t . Тогда алгоритм выглядит так:

- найти t_C — середину интервала $[t_A, t_B]$;
- определить координат кривой (x_A, y_A) и (x_C, y_C) для значений параметра t_A и t_C ;
- сравнить значения y_A и y_C со значениями второй функции в точках x_A и x_C ; если разности

$$f_2(x_A) - y_A \quad \text{и} \quad f_2(x_C) - y_C$$

имеют разный знак, то пересечение есть на интервале $[t_A, t_C]$, если они одного знака, тогда пересечение есть на интервале $[t_C, t_B]$.

```
void SolveParameter( float tA, float tB,
                    float eps, float &x, float &y )
{
float tC, xA, yA, xC, yC, fA, fC;
while ( fabs(tB-tA) > eps ) {
    tC = (tA + tB) / 2.;
    xA = Fx(tA); yA = Fy(tA);
    xC = Fx(tC); yC = Fy(tC);
    fA = yA - F2(xA);
    fC = yC - F2(xC);
    if ( fA*fC < 0 ) tB = tC;
    else tA = tC;
}
x = xA;
y = yA;
}
```

Общий случай

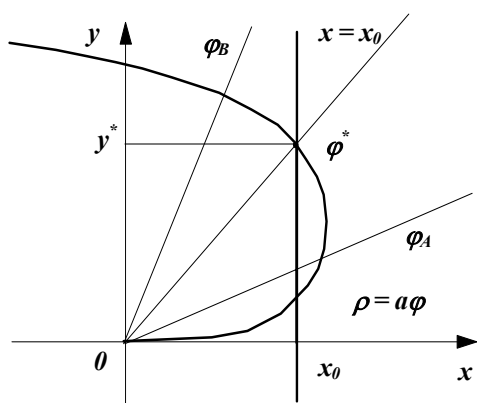
Для общего случая (когда, например, обе кривые заданы в полярных координатах) можно использовать следующий подход:

- выбрать независимую переменную, по которой будет проводиться поиск решения;
- найти функцию, которая меняет знак в точке пересечения кривых (это самый сложный этап, поскольку все зависит от конкретного вида и формы задания этих кривых);
- найти интервал $[A, B]$ изменения независимой переменной, такой, что внутри него содержится точка пересечения, и выбранная функция не меняет знак нигде, кроме точки пересечения;
- применить один из численных методов поиска решения в заданном интервале, например, метод деления отрезка пополам.

Пересечение с вертикальной прямой

Предложенный в предыдущем параграфе алгоритм можно применить и для случая, показанного на рисунке, когда одна из кривых — вертикальная прямая, а вторая задана полярных координатах.

- выберем φ в качестве независимой переменной;
- функция, меняющая знак в точке пересечения: $x_p(\varphi) - x_0$;
- определяем по графику интервал $[\varphi_A, \varphi_B]$;
- применяем метод деления пополам.



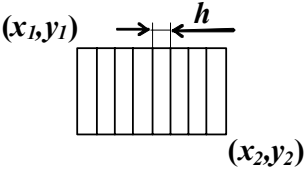
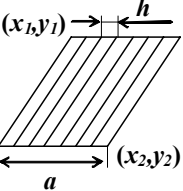
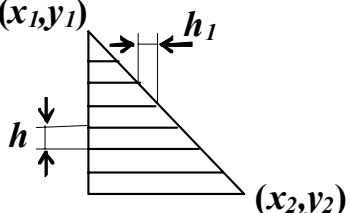
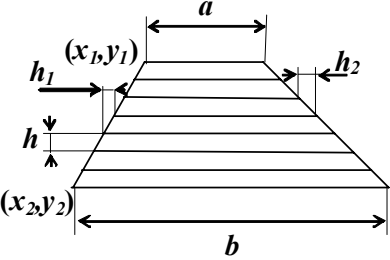
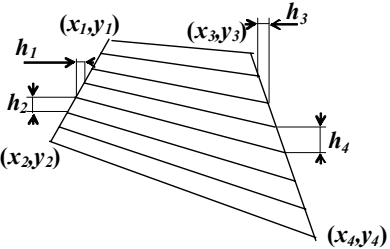
Единственная сложность заключается в проверке наличия корня в некотором интервале изменения угла $[\varphi_A, \varphi_C]$. Для этого поступают следующим образом:

- рассчитывают координаты x_A и x_C точек кривой, соответствующих углам φ_A и φ_C ;
- если разности $x - x_A$ и $x - x_C$ имеют разный знак, то пересечение есть на интервале $[\varphi_A, \varphi_C]$, если они одного знака, тогда пересечение есть на интервале $[\varphi_C, \varphi_B]$.

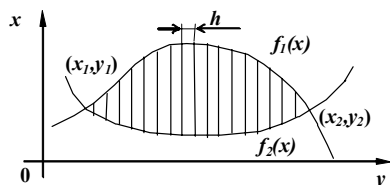
Для кривых, заданных в параметрическом виде, точки пересечения с вертикальной прямой вычисляются почти так же. В этом случае независимой переменной является параметр и координаты x_A и x_C рассчитываются с помощью функции $F_x(t)$.

Штриховка замкнутой области

В данном разделе вы научитесь делать штриховку простейших геометрических фигур на плоскости. Везде предполагается, что надо сделать ровно N линий штриховки (не считая границ фигуры). Для большинства программ потребуется шаг штриховки — расстояние между линиями. Если на интервале длиной L надо провести N линий, не считая границ, то шаг будет равен $\frac{L}{N+1}$, так как интервалов на 1 больше, чем линий.

Фигура	Метод штриховки и программа
<p>прямоугольник</p> 	<p>В цикле меняем x от x_1 до x_2 с шагом h, линии строго вертикальные:</p> <pre data-bbox="531 320 1398 398"> for (x = x1; x <= x2; x += h) line (x, y1, x, y2); </pre>
<p>параллелограмм</p> 	<p>Линии наклонные, параллельные, верхний конец отрезка смещен на a вправо относительно нижнего:</p> <pre data-bbox="515 577 1398 656"> for(x = x1; x <= x1+a; x += h) line (x, y1, x-a, y2); </pre>
<p>треугольник</p> 	<p>Единственная сложность заключается в том, что для каждой следующей линии координата x конца отрезка смещается на $h_1 = \frac{x_2 - x_1}{N + 1}$ по отношению к предыдущей, одновременно с координатой y:</p> <pre data-bbox="515 958 1398 1104"> h1 = (x2 - x1) / (N + 1); xe = x1; for(y = y1; y <= y2; xe += h1, y += h) line (x1, y, xe, y); </pre>
<p>трапеция</p> 	<p>Сразу меняются координаты x для обоих концов отрезка:</p> <pre data-bbox="531 1216 1398 1440"> h1 = (x1 - x2) / (N + 1); h2 = (b - a - x1 + x2) / (N + 1); xs = x1; xe = x1 + a; for(y = y1; y <= y2; xs -= h1, xe += h2, y += h) line (xs, y, xe, y); </pre>
<p>два отрезка</p> 	<p>Надо сделать штриховку так, чтобы каждый отрезок был разделен на $N+1$ равных отрезков: все координаты меняются синхронно на величины соответствующих шагов</p> <pre data-bbox="531 1641 1398 1944"> h1 = (x1 - x2) / (N + 1); h2 = (y2 - y1) / (N + 1); h3 = (x4 - x3) / (N + 1); h4 = (y4 - y3) / (N + 1); xs = x1; xe = x3; ye = y3; for(y = y1; y <= y2; xs -= h1, xe += h3, y += h2, ye += h4) line (xs, y, xe, y); </pre>

криволинейная трапеция



Обычно функции $f_1(x)$ и $f_2(x)$ заданы в математической системе координат, поэтому для рисования отрезка его координаты надо преобразовать к экранным:

```
for( x = x1; x <= x2; x += h )
{
  x0 = ScreenX( x );
  y01 = ScreenY( F1( x ) );
  y02 = ScreenY( F2( x ) );
  line ( x0, y01, x0, y02 );
}
```

Функции $f_1(x)$ и $f_2(x)$, ограничивающие границы фигуры, иногда состоят из нескольких частей, которые имеют разные уравнения. В этом случае для их вычисления можно использовать отдельную функцию языка Си.



Площадь замкнутой области



Общий подход

Существует довольно много разных методов вычисления площади плоской фигуры. Мы рассмотрим четыре самых простых метода и попытаемся сравнить их. Все они являются численными и позволяют найти значение площади только приближенно. Тем не менее, на практике почти всегда можно найти площадь с требуемой точностью (за счет увеличения времени вычислений).

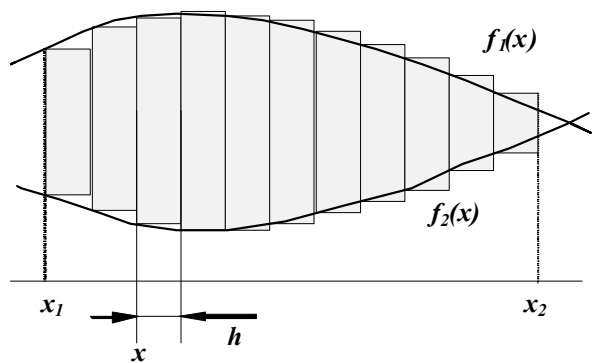
Фактически площадь равна определенному интегралу и часто может быть вычислена аналитически, но эта тема изучается только в конце школьного курса и в высшей математике

Представим себе, что фигура состоит из нескольких простых фигур, для которых мы можем легко подсчитать площадь. В этом случае надо вычислить площади всех этих фигур и сложить их. Но у нашей фигуры границы — кривые линии, поэтому представить ее точно как сумму многоугольников невозможно. Таким образом, используя такой подход, мы вычисляем площадь с некоторой ошибкой, но эту ошибку можно сделать достаточно малой (например, 0,00001).

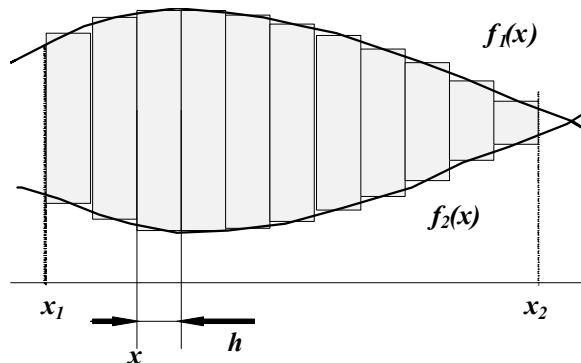


Методы прямоугольников

Простейшими методами вычисления площади являются методы прямоугольников. Из названия ясно, что фигура разбивается на прямоугольники, например так, как показано на рисунках. Обычно ширина h всех прямоугольников выбирается одинаковой, хотя это не обязательно.



Метод левых прямоугольников



Метод средних прямоугольников

Рассмотрим прямоугольник, левая граница которого имеет координату x . Его высоту можно определить разными способами. Наиболее часто используются три способа:

- высота равна $f_1(x) - f_2(x)$ — метод **левых прямоугольников**
- высота равна $f_1(x+h) - f_2(x+h)$ — метод **правых прямоугольников**
- высота равна $f_1(x+h/2) - f_2(x+h/2)$ — метод **средних прямоугольников**

Можно доказать, что самый точный из них — метод средних прямоугольников, он дает наименьшую ошибку.

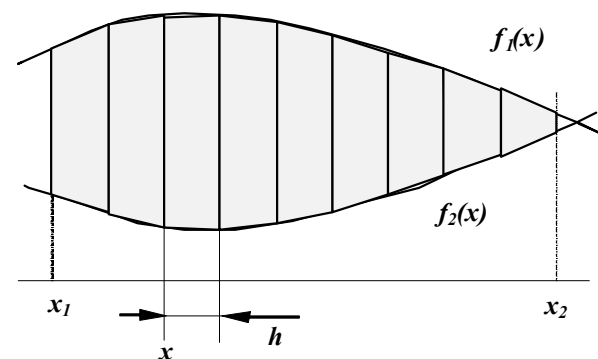
Для того, чтобы вычислить площадь всей фигуры, надо сложить площади всех прямоугольников, имеющих левые границы с координатами от x_1 до x_2-h . Поскольку ширина всех прямоугольников выбирается равной h , можно сложить в цикле все высоты, а затем сумму умножить на h — это будет выполняться быстрее. Ниже приведена программа, вычисляющая площадь S методом средних прямоугольников.

```
float S = 0., x, h = 0.001;
for ( x = x1; x <= x2-h; x += h )
    S += f1(x+h/2) - f2(x+h/2);
S *= h;
```

Если надо повысить точность вычислений, уменьшают шаг h . Методы **левых и правых прямоугольников** имеют погрешность порядка h , что означает, что при уменьшении h в два раза погрешность уменьшается тоже в 2 раза. Метод **средних прямоугольников** имеет погрешность порядка h^2 , то есть при уменьшении h в два раза погрешность уменьшается в 4 раза.

Метод трапеций

Среди простых фигур, которыми можно приближенно заменять «полосы» криволинейной фигуры, удобно использовать **трапеции**, для которых площадь определяется без труда.



Метод трапеций

Площадь одной трапеции равна произведению полусуммы оснований, на высоту. Высота у всех одинакова и равна h , а основания для трапеции, имеющей координату левой границы x , равны, соответственно,

$$f_1(x) - f_2(x) \quad \text{и} \quad f_1(x+h) - f_2(x+h)$$

Заметим также, что правое основание одной трапеции равно левому основанию следующей, что можно учесть при суммировании.

Для вычисления общей площади надо

сложить площади всех трапеций, или, что равносильно, сложить их основания и умножить на $h/2$. Каждое основание входит в сумму два раза (кроме левой границы первой трапеции и правой границы последней). Поэтому программа может выглядеть так:

```
float S, x, h = 0.001;
S = (f1(x1) - f2(x1) + f1(x2) - f2(x2)) / 2.;
for ( x = x1+h; x <= x2-h; x += h )
    S += f1(x) - f2(x);
S *= h;
```

Метод трапеций имеет порядок h^2 , так же, как и метод средних прямоугольников. Удивительно, но его точность ровно в два раза хуже, чем для метода средних прямоугольников.

Метод Монте-Карло

Монте-Карло — всемирный центр игорного бизнеса — дал название большой группе методов, которые позволяют приближенно решать сложные задачи моделирования с помощью случайных чисел. В частности, мы рассмотрим применение метода Монте-Карло (метода статистических испытаний) для вычисления площади фигуры.

Идея метода очень проста. Сначала вся фигура, площадь которой надо определить, заключается внутрь другой фигуры, для которой площадь легко вычисляется. Чаще всего этой фигурой-контейнером является прямоугольник, хотя можно использовать окружность и другие фигуры.

Далее с помощью датчика случайных чисел с **равномерным распределением** генерируются случайные координаты (x, y) точки **внутри прямоугольника** и определяют, попала точка на фигуру или нет. Такие испытания проводятся **N** раз (для увеличения точности **N** должно быть большим, порядка нескольких сотен тысяч или даже нескольких миллионов).

Пусть из **N** точек, полученных таким образом, **M** попали на фигуру. Тогда, считая, что распределение точек в прямоугольнике **равномерное**, площадь фигуры вычисляют как

$$S = \frac{M}{N} (x_2 - x_1)(y_2 - y_1),$$

где произведение двух множителей в скобках — это площадь прямоугольника-контейнера.

```
int i, N = 100000, M = 0;
float S, x, y;
for ( i = 1; i <= N; i ++ ) {
    x = RandFloat (x1, x2); // случайная координата x
    y = RandFloat (y1, y2); // случайная координата y
    if ( InsideFigure(x,y) ) // если точка внутри фигуры,
        M ++;                // то увеличить счетчик
}
S = M * (x2 - x1) * (y2 - y1) / N;
```

Приведенная программа использует функцию **RandFloat**, которая возвращает вещественные числа, равномерно распределенные в заданном интервале:

```
float RandFloat( float min, float max )
{
return (float)rand()*(max - min) / RAND_MAX + min;
}
```

Другая (логическая) функция — **InsideFigure** — возвращает 1 (ответ «да»), если точка (x, y) попала внутрь фигуры, и 0 (ответ «нет»), если не попала:

```
int InsideFigure( float x, float y )
{
return f1(x) >= y && y >= f2(x);
}
```

Метод Монте-Карло дает не очень точные результаты. Его точность сильно зависит от равномерности распределения случайных точек в прямоугольнике.

Конечно, в реальных условиях этот метод следует использовать только тогда, когда других способов решения задачи нет или они чрезвычайно трудоемки. Рассмотрим задачу, которую удобнее всего решать именно методом Монте-Карло.

Задача 1. N треугольников заданы координатами своих вершин. Требуется найти площадь фигуры, образованной объединением всех треугольников.

Сложность. Треугольники могут пересекаться, поэтому фигура может состоять из нескольких частей и иметь границу сложной формы.

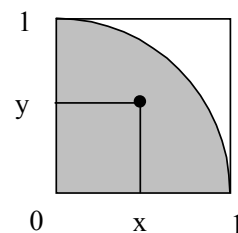
Решение. Решение состоит из двух этапов:

1. Перебором определяем прямоугольник, в котором находятся все заданные треугольники, то есть, находим минимальные и максимальные координаты вершин по каждой оси.
2. Используем метод Монте-Карло, где функция **InsideFigure** возвращает 1 в том случае, если точка попала хотя бы в один треугольник.

Задача 2. Используя метод Монте-Карло, вычислить приближенно число π .

Решение. Известно, что площадь круга равна $S = \pi R^2$, где R — радиус круга. Тогда $\pi = S / R^2$, то есть, зная радиус окружности и определив численно ее площадь, можно приближенно рассчитать значение числа π .

Выберем окружность единичного радиуса с центром в начале координат и рассмотрим ее четверть. Для использования метода Монте-Карло нам надо с помощью датчика случайных чисел, равномерно заполнить точками (пусть их общее количество будет N) квадрат со стороной 1 и сосчитать, сколько из них окажется внутри окружности (это число обозначим через M). Тогда площадь всей окружности примерно равна $4M/N$, а так как ее радиус равен 1, это же число примерно равно π . Таким образом, для решения этой задачи надо N раз выполнить две операции:



1. получить случайные координаты (x, y) точки внутри квадрата — два случайных числа в интервале от 0 до 1;
2. определить, находится ли точка (x, y) внутри окружности, то есть, выполняется ли условие $x^2 + y^2 \leq R^2 = 1$.

3. Вычислительные методы



Целочисленные алгоритмы

К этой группе относятся некоторые классические алгоритмы для работы с целыми числами, которые были известны еще древним математикам.

Алгоритм Евклида (I)

Алгоритм Евклида (3 в. до н.э.) служит для вычисления наибольшего общего делителя (НОД) двух натуральных чисел. Оригинальный алгоритм Евклида основан на равенстве

$$\text{НОД}(a, b) = \text{НОД}(a-b, b) = \text{НОД}(a, b-a).$$

Чтобы не работать с отрицательными числами, можно сформулировать алгоритм так:

Алгоритм Евклида. Заменяем большее из двух чисел разностью большего и меньшего до тех пор, пока они не станут равны — это и есть НОД.

Функция для вычисления НОД может быть записана в двух вариантах:

рекурсивная

```
int NOD ( int a, int b )
{
    if ( a == b ) return a;
    if ( a < b )
        return NOD(a, b-a);
    else return NOD(a-b, b);
}
```

нерекурсивная

```
int NOD ( int a, int b )
{
    while ( a != b ) {
        if ( a > b ) a -= b;
        else      b -= a;
    }
    return a;
}
```

Конечно, в данном случае нерекурсивная форма лучше: она работает быстрее и не расходует стек попусту.

Алгоритм Евклида (II)

Первый вариант алгоритма Евклида медленно работает в том случае, если числа сильно отличаются (например, вычисление **НОД(2, 1998)** потребует 998 шагов цикла). Поэтому чаще используют *модифицированный алгоритм Евклида*, основанный на использовании равенства

$$\text{НОД}(a, b) = \text{НОД}(a, b \% a) = \text{НОД}(a \% b, b)$$

Модифицированный алгоритм Евклида. Заменяем большее из двух чисел остатком от деления большего на меньшего до тех пор, пока меньшее не станет равно нулю. Тогда второе число и есть НОД.

Запишем на языке Си только нерекурсивный вариант:

```
int NOD ( int a, int b )
{
    while ((a != 0) && (b != 0)) {
        if ( a > b ) a = a % b;
        else      b = b % a;
    }
    return a+b;
}
```

Наименьшее общее кратное

Наименьшим общим кратным (НОК) двух чисел называется наименьшее число, которое делится без остатка на оба исходных числа.

Найти наименьшее общее кратное можно очень просто, если знать НОД исходных чисел. Действительно, пусть $x = x_1 \cdot d$ и $y = y_1 \cdot d$ где $d = \text{НОД}(x, y)$. Тогда

$$\text{НОК}(x, y) = xy_1 = x_1y = \frac{xy}{\text{НОД}(x, y)}.$$

Для вычисления НОД можно использовать алгоритм Евклида.

Решето Эратосфена

Другая задача, которую решили математики Древней Греции — нахождение всех простых чисел в интервале от 1 до заданного числа N . Самым быстрым и поныне считается *алгоритм Эратосфена* (275-195 гг. до н.э.). Существует предание, что Эратосфен выписал в ряд на папирусе все натуральные числа и затем прокалывал каждое второе (то есть делящееся на 2), потом — каждое третье, и т.д. В конце этой процедуры остаются «непроколотыми» только простые числа, которые делятся только на себя и на 1.

Вместо папируса мы будем использовать массив A , в котором элемент $A[i]$ для всех i от 1 до N принимает два возможных значения:

- 1, число «не проколото» и является кандидатом на простые;
- 0, число «проколото» и больше не рассматривается.

В целях экономии памяти можно выбрать не целые переменные (занимающие 4 байта), а символьные (например, `unsigned char`, которые занимают 1 байт). Кроме того, для экономии памяти еще в 8 раз можно рассматривать отдельные биты числа, но этот вариант мы не затрагиваем.

Будем перебирать все числа k от 2 до \sqrt{N} (включительно) и в цикле «вычеркивать» числа, кратные k : $2k, 3k, \dots$. Очевидно, что надо использовать только те числа k , которые еще не вычеркнуты. Ниже полностью приведена программа, реализующая этот алгоритм. Заметьте также, что удобно выделить место в памяти под $N+1$ элемент, чтобы использовать элементы с $A[1]$ по $A[N]$. При этом элемент $A[0]$ не используется. Можно этого избежать, но это усложнит программу.

```
#include <stdio.h>
main()
{
    unsigned char *A;
    int i, k, N;
    printf ("Введите максимальное число ");
    scanf ( "%d", &N );

    A = new unsigned char[N+1]; // выделить память под массив
    if ( A == NULL ) return 1; // выход в случае ошибки

    for ( i=1; i<=N; i++ ) A[i] = 1;
        for ( k=2; k*k<=N; k++ )
            if ( A[k] != 0 )
                for ( i=k+k; i<=N; i+=k ) A[i] = 0;

    for ( i=1; i<=N; i++ )
        if ( A[i] == 1 ) printf ( "%d\n", i );
}
```

Главное **преимущество** этого алгоритма — высокая скорость работы, основной **недостаток** — большой объем необходимой памяти. Он демонстрирует одну из принципиальных проблем программирования: как правило, повышение быстродействия алгоритма требует увеличения необходимого объема памяти, а экономия памяти приводит к снижению быстродействия. В реальных ситуациях чаще всего приходится идти на компромисс.



Многоразрядные целые числа

Как работать в языке Си с многоразрядными целыми числами, которые не помещаются в одну ячейку памяти типа `int` (то есть, превышают по модулю 2 147 483 647)? Очевидно, что надо отвести на них несколько ячеек, но при этом придется вручную реализовывать все операции с таким числом. Существует два подхода:

- 1) число хранится в виде символьного массива, в котором каждый элемент обозначает одну цифру от 0 до 9;
- 2) число хранится в виде массива целых чисел, где каждый элемент обозначает одну или несколько цифр.

Мы рассмотрим второй способ. Для экономии места в памяти выгоднее размещать в одной ячейке как можно больше цифр. При этом надо следить, чтобы результаты всех промежуточных операций не превышали максимального значения для выбранного типа данных.

Рассмотрим, например, число 12 345678 901234 567890. Его можно записать в виде

$$12 \cdot (10^6)^3 + 345678 \cdot (10^6)^2 + 901234 \cdot (10^6)^1 + 567890 \cdot (10^6)^0$$

что соответствует записи в системе счисления с основанием **d=1000000**. Цифрами в этой системе служат многоразрядные десятичные числа. Для записи этого конкретного числа нам понадобится всего 4 ячейки типа `int`.

Теперь надо научиться выполнять арифметические операции с такими числами. Рассмотрим в общем виде умножение числа

$$A = a_n d^n + a_{n-1} d^{n-1} + \dots + a_2 d^2 + a_1 d + a_0$$

на некоторое («короткое») число **B**. В результате получим третье число **C**, которое также может быть представлено в системе счисления с основанием **d**:

$$C = c_m d^m + c_{m-1} d^{m-1} + \dots + c_2 d^2 + c_1 d + c_0$$

Очевидно, что $c_0 = (a_0 B) \% d$, при этом в следующий разряд добавляется перенос $r_1 = a_0 B / d$, где остаток от деления отбрасывается. Для следующих разрядов можно составить таблицу

$$\begin{array}{ll} c_0 = (a_0 B) \% d & r_1 = (a_0 B) / d \\ c_1 = (a_1 B + r_1) \% d & r_2 = (a_1 B + r_1) / d \\ c_2 = (a_2 B + r_2) \% d & r_3 = (a_2 B + r_2) / d \end{array}$$

... и т.д.

Вычисления заканчиваются, когда одновременно выполняются два условия:

1. Все цифры числа *A* обработаны.
2. Перенос в следующий разряд равен нулю.

Для удобства фактическая длина числа обычно хранится в отдельной переменной.



Вычисление 100!

Вычислим значение факториала числа 100:

$$100! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot 99 \cdot 100.$$

Можно примерно оценить, что оно меньше, чем 100^{100} , поэтому включает не более 201 цифры. Кроме того, можно сказать, что на конце этого числа будет 24 нуля, так как нули получаются при умножении на число, кратное 10 (10 штук), при умножении чисел, заканчивающихся на 2 и

на 5 (10 штук), при умножении чисел 25, 50, 75 и 100 (содержащих множитель 25) на четные числа (4 дополнительных нуля).

Поскольку $10!$ оканчивается на цифру 8 (если не считать конечные нули), то число $100!$ оканчивается на ту же цифру, что и 8^{10} , то есть на 4 (так как 8^2 заканчивается на 4, 8^5 – на 8).

Проверим правильность выбора основания. Если $d=1000000$, такое значение поместится в ячейку типа `int`. Наибольшее число, которое может получиться в результате умножения, равно $1000000 \cdot 100$, что тоже помещается в разрешенный диапазон ($-2^{31} \dots 2^{31}-1$).

```

const int d = 1000000; // d - основание
int A[40] = {1}; // A[0]=1, остальные A[i]=0
int i, k, len = 1, r, s; // len - длина числа, r - остаток
for ( k = 2; k <= 100; k ++ ) { // умножаем на 2, 3, ..., 100
    i = 0; // начинаем с младшего разряда (0)
    r = 0; // сначала перенос - 0
    while ( i < len // пока не все разряды обработаны
           || r > 0 ) { // или есть перенос
        s = A[i]*k + r; // умножаем разряд, добавляем перенос
        A[i] = s % d; // остается в этом разряде
        r = s / d; // перенос в следующий разряд
        i ++; // переход к следующему разряду
    }
    len = i; // изменили длину числа
}
for ( i = len-1; i >= 0; i -- ) // вывод на экран
    if ( i == len-1 ) printf("%d", A[i]); // 123 -> 123
    else printf("%.6d", A[i]); // 123 -> 000123

```

Обратите внимание, что все разряды длинного числа, кроме самого старшего, выводятся по формату "% .6d". Этот формат означает, что нужно вывести 6 знаков, а если число занимает меньше, дополнить его слева нулями. При этом все лидирующие нули сохраняются.

Многочлены

Как известно, многочленом (полиномом) называют функцию вида

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$$

Часто требуется вычислить значения многочлена в заданной точке $x = x_0$. Если выполнять вычисления «напрямую», каждый раз возводя x в степень (с помощью умножения), требуется

$$n \text{ сложений и } 1 + 2 + \dots + (n-1) + n = \frac{n(n+1)}{2} \text{ умножений.}$$

Однако для решения этой задачи придумали более эффективный способ.

Схема Горнера

Представим многочлен в виде

$$f(x) = (\dots(a_n x + a_{n-1})x + a_{n-2})x + \dots a_2)x + a_1)x + a_0$$

При этом для вычисления значения функции при известном x требуется всего n сложений и n умножений, что значительно быстрее. Ниже приведена функция на языке Си, реализующая схему Горнера:

```
float Gorner ( float x, float a[], int n )
{
    float v = 0.;
    for ( int i = n; i >= 1; i -- )
        v = ( v + a[i] ) * x;
    v += a[0];
    return v;
}
```



Последовательности и ряды



Последовательности

Последовательность — это набор элементов, расположенных в определенном порядке.

Пока будем рассматривать только последовательности чисел. Все элементы последовательности имеют номера, обычно начиная с 1. Для того, чтобы задать последовательность, используют два следующих способа.

1. *Рекуррентную* формулу, которая позволяет вычислить элемент с номером n , зная один или несколько предыдущих. Например, последовательность

$$1, 3, 5, 7, \dots$$

можно задать рекуррентной формулой $a_n = a_{n-1} + 2$.

2. Формулу для n -ого члена последовательности. Для той же последовательности легко получить $a_n = 2n - 1$. Правая часть этой формулы зависит только от номера элемента n . Для хорошо известной **арифметической прогрессии** формула n -ого члена имеет вид

$$a_n = a_1 + (n - 1)d$$

где a_1 — начальный элемент, а d — разность. Для **геометрической прогрессии**, соответственно, $a_n = a_1 q^{(n-1)}$, где q — знаменатель прогрессии.

В таблице ниже приводятся примеры последовательностей.

№	последовательность	a_n
1	1, 3, 5, 7, ...	$2n-1$
2	1, 3, 7, 15, ...	2^n-1
3	2, 9, 28, 65, ...	n^3+1
4	$1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \dots$	$\frac{1}{n}$
5	$1, 1, \frac{5}{7}, \frac{7}{15}, \dots$	$\frac{2n-1}{2^n-1}$

Самые распространенные задачи, связанные с последовательностями:

- определить, сколько членов убывающей последовательности больше некоторого заданного значения;
- найти первый член последовательности, удовлетворяющий заданному условию;
- найти сумму заданного числа элементов последовательности или сумму элементов, удовлетворяющих заданному условию.

Задача. Для последовательности 4 в таблице найти

- 1) сумму элементов, которые больше 10^{-5} ,
- 2) сколько элементов вошло в эту сумму,
- 3) каков первый элемент, меньший 10^{-5} ?

При решении подобных задач особое внимание надо обращать на эффективность вычислений и точность. Если на каждом шаге делать умножение и возведение в степень 2^n , такая программа будет неэффективной.

Можно сделать иначе: ввести две переменные, скажем, **u** и **d**, которые будут обозначать 2^{n-1} и 2^n , соответственно. Первая будет с каждым шагом увеличиваться на 2, а вторая — в 2 раза. Ниже приведены две программы, одна из которых работает в 3 раза быстрее другой. В обоих случаях искомые величины находятся в переменных **s**, **n**, **a**.

```
int n;
float a, s;
n = 0; s = 0;
while ( 1 ) {
    n ++;
    a=(2*n-1) / (pow(2,n) -1);
    if ( a < 1.e-5 ) break;
    s += a;
}
```

```
int n;
float a, s, u, d;
n = 0; s = 0;
u = 1; d = 2;
while ( 1 ) {
    n ++;
    a = u / (d-1.);
    if ( a < 1.e-5 ) break;
    u += 2; d *= 2;
}
```

📖 Рекуррентные последовательности

В 1202 г. итальянский математик Леонардо Пизанский, известный под именем Фибоначчи, предложил такую задачу:

Задача Фибоначчи. Пара кроликов каждый месяц дает приплод – самца и самку, которые через 2 месяца снова дают такой же приплод. Сколько пар кроликов будет через год, если сейчас мы имеем 1 пару молодых кроликов?

Количество кроликов меняется с каждым месяцем в соответствии с последовательностью

1, 1, 2, 3, 5, 8, 13, 21, 34, ...

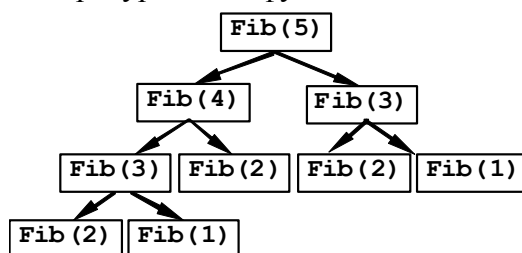
которую называют **последовательностью Фибоначчи**. Она задается не общей формулой **n**-ого члена, а **рекуррентной формулой**, в которой **n**-ый член выражается через предыдущие. При этом надо определить начальные элементы:

$$a_1 = 1, \quad a_2 = 1,$$

$$a_n = a_{n-1} + a_{n-2} \quad \text{для } n > 2$$

Такая последовательность очень просто реализуется с помощью рекурсивной функции

```
int Fib ( int n )
{
    if ( n < 3 ) return 1;
    else
        return Fib(n-1)+Fib(n-2);
}
```



Однако она крайне неэффективна, потому что вычисление, например, **Fib(5)**, приводит к 9 вызовам функции, что в данном случае неоправданно.

Выход из этой ситуации состоит в написании *итеративной* функции, которая использует не рекурсию, а циклы (*итерации*). Заметим, что для вычисления значения a_n нам надо знать (то есть запомнить) значения a_{n-1} и a_{n-2} (в программе они обозначены как **a1** и **a2**). Чтобы не обрабатывать отдельно случаи, когда $n < 3$, последовательность можно «продолжить влево», добавив фиктивные члены $a_{-1} = 1$ и $a_0 = 0$.

```
int Fib ( int n )
{
    int a2 = 1, a1 = 0, a, i;
    for ( i=1; i<=n; i++ )
    {
        a = a1 + a2;      // считаем очередной элемент
        a2 = a1; a1 = a; // продвижение вперед
    }
    return a;
}
```

Родственные задачи

Задача. Вычислить значение выражения справа.

Для решения этой задачи заметим, что считать надо снизу. Обозначим

$$s_{100} = \frac{1}{100}, \quad s_{99} = \frac{1}{99 + \frac{1}{100}} = \frac{1}{99 + s_{100}} \quad \text{и т.д.}$$

Числа $s_{100}, s_{99}, s_{98}, \dots, s_1$ представляют собой последовательность с рекуррентной формулой для n -ого члена $a_n = \frac{1}{101 - n + a_{n-1}}$.

Реализация этого алгоритма на языке Си приводится ниже:

```
float s = 0;
for ( int k = 100; k >= 1; k -- )
    s = 1 / (k + s);
```

Ряды

Ряд – это сумма бесконечного числа элементов последовательности.

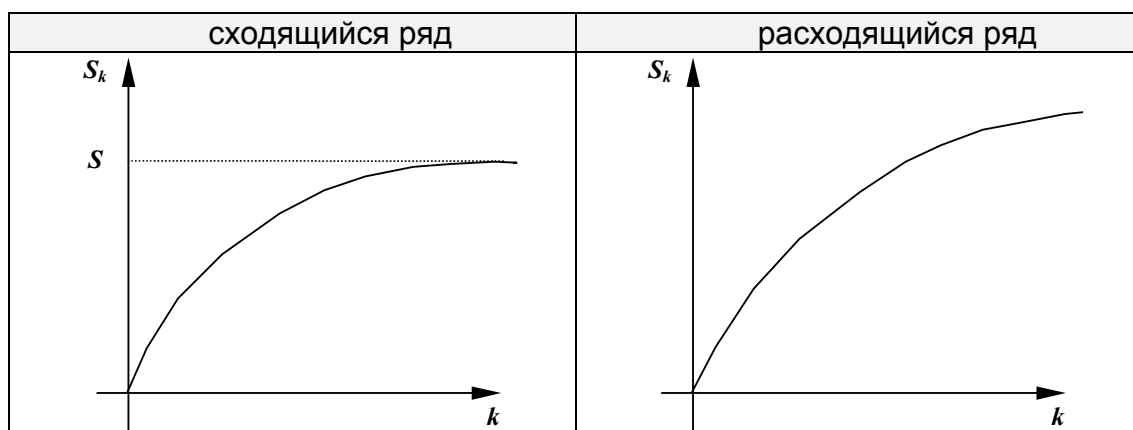
Конечно, на практике невозможно просуммировать бесконечное количество элементов, поэтому бесконечную сумму заменяют конечной — **частичной** суммой ряда, в которую входят первые k элементов:

$$S = \sum_1^{\infty} a_n \approx S_k = \sum_1^k a_n$$

Существует два типа рядов: **расходящиеся** и **сходящиеся**. Для расходящегося ряда частичная сумма S_k может стать больше по модулю, чем любое наперед заданное число. Примером расходящегося ряда является **гармонический ряд**:

$$S = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \dots$$

Для сходящегося ряда частичная сумма стремится, при увеличении количества элементов k , к некоторому фиксированному конечному значению, которое и называется суммой этого ряда.



Для сходящихся рядов часто можно вычислить сумму ряда с заданной точностью ε , так что

$$|S - S_k| = \left| \sum_{k+1}^{\infty} a_n \right| < \varepsilon$$

В общем случае это достаточно сложная математическая задача, однако для рядов особого вида она имеет простое решение.

Если ряд знакопеременный и его элементы убывают, то он сходится, и ошибка в вычислении суммы не превышает модуля последнего отброшенного элемента

Знакопеременным называется ряд, в котором знаки элементов чередуются, например

$$S = 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \frac{1}{5} - \dots + \frac{(-1)^{n+1}}{n} + \dots$$

📖 Вычисление функций

Многие математические функции вычисляются в компьютерах с помощью **функциональных рядов**, то есть рядов, зависящих от переменной. Это позволяет не хранить в памяти таблицы синусов, а непосредственно считать значение функции для любого угла. Как всегда, экономия памяти приводит к снижению быстродействия. Ниже приведены наиболее известные ряды:

$$\begin{aligned}
 e^x &= 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \dots + \frac{x^n}{n!} + \dots \\
 \sin x &= x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots + \frac{(-1)^{n+1} x^{2n-1}}{(2n-1)!} + \dots \\
 \cos x &= 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots + \frac{(-1)^{n+1} x^{2n}}{(2n)!} + \dots
 \end{aligned}$$

Для примера напишем функцию для вычисления **sin x** с точностью 10^{-6} . Поскольку ряд для синуса знакопеременный и его элементы убывают, надо суммировать все элементы, модуль которых больше 10^{-6} . Для того, чтобы учитывать меняющийся знак, нет смысла использовать возведение в степень — достаточно ввести переменную, которая будет принимать значение 1 или -1.

```
double sin1 ( double x )
```

s частичная сумма ряда
a элемент ряда
xx значение x^{2n-1}
fact значение $(2n-1)!$

```

{
double s = 0, a, xx = x,
      fact = 1, n2 = 1;
int z = 1;
do {
    a = z * xx / fact;
    s += a;
    z = -z;
    xx *= x*x;
    n2 += 2;
    fact *= (n2-1)*n2;
}
while ( fabs(a) > 1.e-6 );
return s;
}

```

Интересно определить количество членов ряда, необходимых для вычисления синуса с точностью 10^{-6} , для разных углов (см. таблицу).

диапазон углов (по модулю)	необходимое количество членов ряда
$0^\circ - 90^\circ$	1 – 7
$90^\circ - 180^\circ$	7 – 9
$180^\circ - 270^\circ$	10 – 12
$270^\circ - 360^\circ$	12 – 14

Вспомним, что синус любого угла может быть представлен как синус угла в интервале от 0° до 90° (возможно с переменной знака)

$$\sin \alpha = \sin(180^\circ - \alpha) = -\sin(-\alpha) = -\sin(\alpha - 180^\circ).$$

Таким образом, в любом случае можно использовать не более 7 членов ряда.



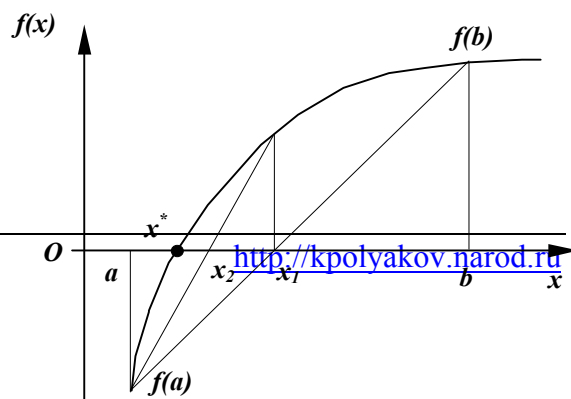
Численное решение уравнений

Многие уравнения невозможно решить аналитически, выписав решение в виде формулы в явном виде. В таких случаях используют **приближенные численные** методы. Приближенными они называются потому, что мы в принципе не можем найти точное решение \mathbf{x}^* , однако можем найти некоторое приближение к нему, которое отличается от точного решения \mathbf{x}^* не более, чем на заданную величину ε .

Мы будем рассматривать уравнения вида $\mathbf{f}(\mathbf{x})=0$, где $\mathbf{f}(\mathbf{x})$ — функция одной переменной. Один из численных методов решения таких уравнений — **метод деления отрезка пополам** или **дихотомии** — мы уже рассматривали. К его преимуществам можно отнести то, что мы можем гарантировать, что ошибка не превышает заданную величину. С другой стороны, надо заранее определить интервал, в котором находится один и только один корень.

Метод хорд

Так же, как и метод дихотомии, этот метод предназначен для уточнения корня на известном интервале $[a, b]$ при условии, что корень существует



и непрерывная функция $f(x)$ имеет на концах отрезка разные знаки.

За следующее приближение к корню принимается не середина отрезка $[a, b]$, как в методе дихотомии, а значение x в точке, где прямая, соединяющая точки $(a, f(a))$ и $(b, f(b))$ пересекает ось Ox . Уравнение прямой, проходящей через эти точки, имеет вид:

$$\frac{x-a}{b-a} = \frac{y-f(a)}{f(b)-f(a)}$$

В точке пересечения имеем $y=0$. Подставляя в уравнение $y=0$, получаем

$$x_1 = a - \frac{(b-a)f(a)}{f(b)-f(a)}$$

Далее все происходит так, как при использовании метода дихотомии: проверяем, на каком из интервалов $[a, x_1]$ и $[x_1, b]$ находится пересечение, и смещаем соответственно точку a или точку b .

Более сложен вопрос о том, когда закончить итерации. Обычно применяется один из двух критериев:

- 1) разность между двумя последовательными приближениями x_k и x_{k-1} стала меньше заданной точности ε , или...
- 2) модуль значения функции $f(x_k)$ в точке x_k стал меньше заданного значения ε_1 .

При этом, к сожалению, нельзя гарантировать, что ошибка составит не более какой-либо величины — она зависит от вида функции.

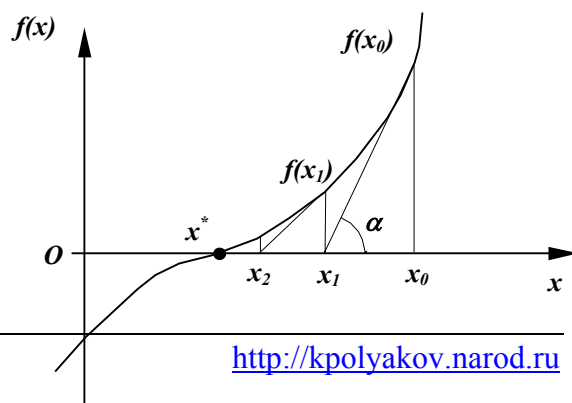
```
float Chord( float a, float b, float eps, float eps1 )
{
    float x, x0 = a, fa, fb, fx;
    while ( 1 ) {
        fa = F(a); fb = F(b);
        x = a - (b - a)*fa / (fb - fa); // следующее приближение
        fx = F(x);
        if ( fabs(fx) < eps1 ) break; // проверка условия 2
        if ( fa*fx < 0 ) b = x;      // сужаем интервал поиска
        else          a = x;
        if ( fabs(x-x0) < eps ) break; // проверка условия 1
        x0 = x;
    }
    return x;
}
```

Параметры **eps** и **eps1** обозначают точность для изменения x и значения функции в этой точке. Эта функция может быть еще оптимизирована, так как можно не вычислять заново каждый раз значения $f(a)$ и $f(b)$ — одно из них мы знаем с предыдущего шага.

📄 Метод Ньютона (метод касательных)

Чаще всего на практике используется метод Ньютона, который обладает быстрой сходимостью и требует знать только начальное приближение x_0 к корню, а не интервал, в котором он находится.

За следующее приближение к корню принимается значение x в точке пересечения касательной, проведенной к кривой в точке $(x_0, f(x_0))$, и оси Ox . Поскольку



$$tg\alpha = f'(x_0) = \frac{f(x_0)}{x_0 - x_1},$$

для k -ого шага итерации получаем

$$x_k = x_{k-1} - \frac{f(x_{k-1})}{f'(x_{k-1})}$$

Метод Ньютона обладает высокой скоростью сходимости. Обычно точность 10^{-6} достигается за 5-6 итераций. Его главный недостаток в том, что надо на каждом вычислять производную функции, а выражение для нее может быть неизвестно, например, если она задана таблицей. Иногда используют модифицированный метод Ньютона, в котором на всех шагах используется производная, вычисленная в точке x_0 . Важно также правильно выбрать начальное приближение к корню, иначе метод Ньютона может «зациклиться».

Приведенная ниже функция использует функции $F(\mathbf{x})$ и $DF(\mathbf{x})$, которые возвращают соответственно значение функции и ее производную в точке \mathbf{x} .

```
float Newton ( float x0, float eps)
{
    float f1;
    do {
        f1 = F(x0) / DF(x0);
        x0 -= f1;
    }
    while ( fabs(f1) > eps );
    return x0;
}
```

Метод Ньютона используется также для решения систем нелинейных уравнений с несколькими переменными, но формулы оказываются более сложными и мы их не рассматриваем.

📖 Метод итераций

От исходного уравнения $\mathbf{f}(\mathbf{x}) = 0$ можно легко перейти к эквивалентному (имеющему те же самые корни)

$$x + bf(x) = x,$$

которое получено сначала умножением обеих частей на константу \mathbf{b} (не равную нулю), а затем добавлением \mathbf{x} . Вводя обозначение $\varphi(\mathbf{x}) = \mathbf{x} + \mathbf{b} \cdot \mathbf{f}(\mathbf{x})$, получаем уравнение

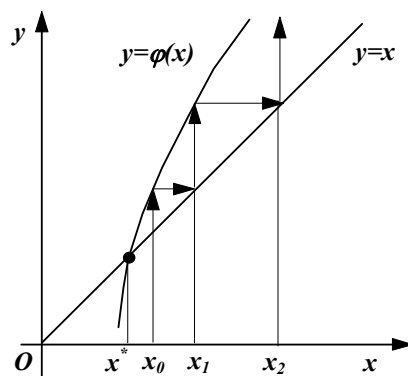
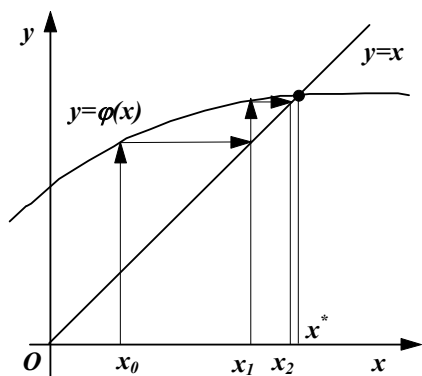
$$x = \varphi(x),$$

которое дает *итерационную формулу* (формулу для повторяющихся вычислений)

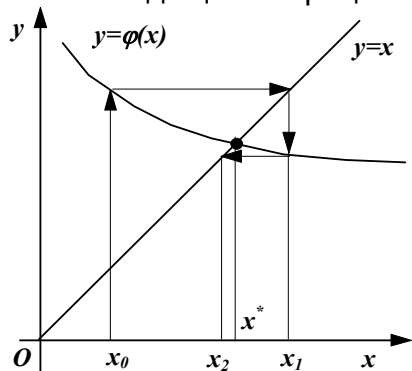
$$x_k = \varphi(x_{k-1}).$$

Теперь надо установить, при каких условиях такая процедура позволяет получить значение \mathbf{x}^* (процесс сходится). Рассмотрим 4 случая.

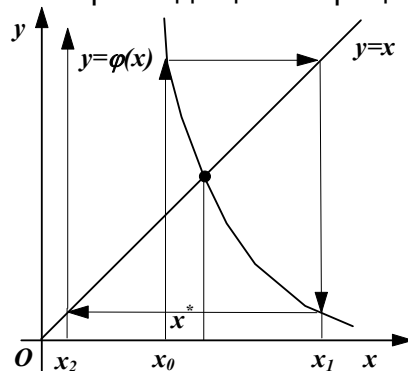
- а) односторонний сходящийся процесс б) односторонний расходящийся процесс



в) двусторонний сходящийся процесс



г) двусторонний расходящийся процесс



Можно показать, что процесс сходится при условии

$$|\varphi'(x)| < 1$$

причем при $0 < \varphi'(x) < 1$ имеет место односторонняя сходимость, а при $-1 < \varphi'(x) < 0$ — двусторонняя. Учитывая, что $\varphi(x) = x + b \cdot f(x)$, можно получить

$$\varphi'(x) = 1 + bf'(x)$$

Наибольшая скорость сходимости наблюдается при $\varphi'(x) = 0$, для этого случая имеем

$$b = -\frac{1}{f'(x)}$$

что дает формулу Ньютона. Таким образом, метод Ньютона имеет наивысшую скорость сходимости среди всех итерационных методов.

Поскольку в общем случае нельзя гарантировать, что метод итераций сходится (это его главный недостаток), в программе надо ограничивать количество шагов некоторым максимальным значением. Функция, реализующая метод итераций, приведена ниже. Выход происходит тогда, когда разность между двумя последовательными приближениями станет меньше заданной точности ϵ , или превышено заданное максимальное число итераций n . Для того, чтобы вызывающая программа могла получить информацию о том, что процесс расходится, параметр n сделан переменным — на выходе он равен числу итераций. Если оно превышает заданное значение, результат неверный и надо менять константу b .

```
float Iter ( float x0, float eps, int &n)
{
  int i = 0;
  float x = x0;
  do {
    x0 = x;          // запомнить предыдущее x
    x = F(x0);      // шаг итерации
    i ++;
  }
}
```

```

        if ( i > n ) break; // процесс расходится, выход
    }
    while ( fabs(x-x0) > eps ); // пока не найдено решение
    n = i;
    return x;
}

```

Вызов этой функции может выглядеть так:

```

int n = 100;
float x;
...
x = Iter ( 1.2, 0.0001, n );
if ( n > 100 )
    printf("Процесс расходится");

```

Использование функции-параметра

В параметры всех процедур можно включать функцию, которая используется в вычислениях. Сначала надо объявить новый тип данных — функция, которая принимает один вещественный параметр и возвращает вещественный результат.

```
typedef float (*func)( float x );
```

Затем этот тип **func** можно использовать в параметрах функций. Например, функция для метода итераций может быть модифицирована так

```

float Iter ( func F, float x0, float eps, int &n)
{
    ...
}

```

Все ее содержимое сохраняется без изменений. При вызове надо передать в качестве первого параметра имя функции, которая вычисляет по заданной формуле.

Вычисление определенных интегралов

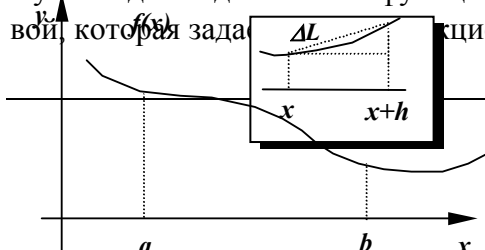
Ставится задача вычисления определенного интеграла

$$J = \int_a^b f(x) dx$$

на конечном интервале в том случае, когда первообразной в аналитическом виде не существует или она очень сложна. Поскольку этот интеграл геометрически представляет собой **площадь** под кривой $y = f(x)$, для его вычисления можно использовать все методы вычисления площадей, рассмотренные в начале главы (методы прямоугольников, трапеций, Монте-Карло, Симпсона).

Вычисление длины кривой

Пусть задана однозначная функция одной переменной $y = f(x)$. Требуется найти длину **L** кривой, которая задана функцией, на известном интервале $[a, b]$.



Эта задача решается в высшей математике с помощью определенного интеграла

$$L = \int_a^b \sqrt{1 + [f'(x)]^2} dx$$

Однако взять такой интеграл аналитически достаточно сложно, поэтому мы будем использовать приближенный метод. Рассмотрим участок кривой на интервале $[x, x+h]$, где h – малая величина шага. Можно приближенно считать, так же, как и при вычислении площади, что длина этой части кривой приближенно равна длине отрезка, соединяющего точки кривой на концах интервала. По теореме Пифагора находим

$$\Delta L = \sqrt{h^2 + [f(x+h) - f(x)]^2}$$

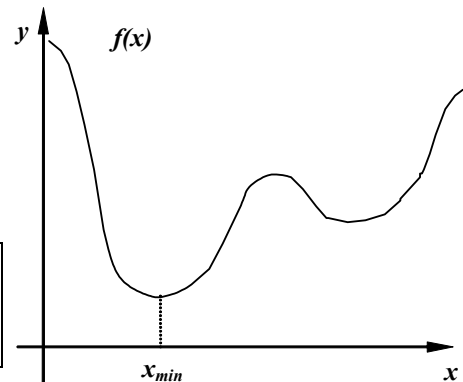
Теперь для определения полной длины кривой на участке $[a, b]$ остается сложить длины всех таких отрезков при изменении x от a до b . Соответствующая функция приведена ниже

```
float CurveLength ( func F, float a, float b )
{
float x, dy, h = 0.0001, h2 = h*h, L = 0;
for ( x = a; x < b; x += h )
{
dy = F(x+h) - F(x);
L += sqrt(h2 + dy*dy);
}
return L;
}
```

Оптимизация

На практике часто существует множество решений задачи, и надо найти оптимальное решение, то есть выбрать параметры из некоторой области так, чтобы заданная функция (например, потери) была минимальной. Если нужен максимум функции (например, прибыли), достаточно сменить ее знак на обратный и искать минимум новой функции.

Задача. Найти такое значение параметра x , при котором функция $f(x)$ достигает минимального значения в некоторой области.



Задача осложняется тем, что в реальных задачах существует множество минимумов функции (они называются **локальными**), тогда как нас интересует **глобальный** минимум — минимальное из всех этих значений. Общих способов поиска глобального минимума функций не существует. Для функций одной переменной задача разбивается на 2 этапа:

- 1) некоторым способом (например, графически) определяются интервалы, на которых функция $f(x)$ имеет один минимум или максимум, то есть является **унимодальной**;
- 2) на каждом интервале уточняются минимальные значения функции, и из них выбирается наименьшее.

Метод золотого сечения

Этот метод предназначен для уточнения минимума функции на интервале $[a, b]$, где она является **унимодальной** (имеет только один минимум). Задача состоит в том, чтобы использо-

вать такую стратегию поиска, которая позволила бы построить последовательность уменьшающихся интервалов неопределенности

$$[a, b], \quad [a_1, b_1], \quad [a_2, b_2], \quad \dots, \quad [a_n, b_n]$$

так чтобы длина последнего интервала стала меньше допустимой точности ε . Доказано, что оптимальной будет стратегия, при которой используются числа Фибоначчи:

$$F_0 = 1, \quad F_1 = 1, \quad F_k = F_{k-1} + F_{k-2} \quad \text{для } k > 1$$

Интервал $[a, b]$ делится на 2 части в отношении двух последовательных чисел Фибоначчи в точке x_1 . Далее, симметрично относительно центра отрезка выбирается точка x_2 . Следующий интервал определяется в зависимости от значений функции в точках x_1 и x_2 так, как показано в таблице (будем считать, что $x_1 < x_2$).

$f(x_1) < f(x_2) \Rightarrow [a, x_2]$	$f(x_1) > f(x_2) \Rightarrow [x_1, b]$	$f(x_1) = f(x_2) \Rightarrow [x_1, x_2]$

Существенный недостаток этого метода состоит в том, что при разбиении отрезка надо учитывать номер шага N , чтобы найти соответствующие числа Фибоначчи и их отношение F_{N-1}/F_N . Установлено, что при увеличении N это отношение стремится к числу ≈ 0.618 . Деление отрезка на две части в таком отношении называется **золотым сечением**. При этом отношение длины большей части отрезка ко всей длине равно отношению длины меньшей части к большей, то есть, обозначив через g длину большей части, получаем

$$g = \frac{1-g}{g},$$

что приводит к квадратному уравнению, положительный корень которого равен

$$g = \frac{-1 + \sqrt{5}}{2} \approx 0.618034$$

Если на каждом шаге отрезок $[a, b]$ делится в этом отношении, после N шагов длина интервала неопределенности уменьшится до $(b-a)g^N$. Это позволяет заранее оценить необходимое количество итерации для достижения заданной точности. Такой метод называется **методом золотого сечения**.

```
typedef float (*func) (float x);
float Gold ( func F, float a, float b, float eps )
{
    float x1, x2, g=0.618034, R = g*(b - a);
    while ( fabs(b-a) > eps ) {
        x1 = b - R;
```

```

    x2 = a + R;
    if ( F(x1) > F(x2) ) a = x1;
    else                b = x2;
    R *= g;
    }
return (a + b) / 2.;
}

```

В приведенном варианте программа ищет минимум функции на заданном интервале. Если надо искать максимум, функцию требуется немного изменить. Недостаток этого способа в том, что надо знать интервал, в котором находится единственный минимум. Кроме того, его сложно применить для поиска минимума функций нескольких переменных.

📖 Градиентный метод

Для использования этого метода надо знать некоторое начальное приближение к точке минимума. Далее используется следующий принцип: смещаемся на шаг h в ту сторону, в которую функция убывает быстрее всего. Для функции многих переменных это направление определяется **градиентом** — направлением наибольшего возрастания функции. Поскольку мы ищем минимум, нужно двигаться в сторону, противоположную градиенту.

Для функции одной переменной нужно двигаться в ту сторону, в которую производная убывает. Если производная положительна (функция возрастает), для поиска минимума надо уменьшать x , если отрицательна, то увеличивать. Тогда итерационную формулу можно записать так

$$x_k = x_{k-1} - h \cdot f'(x_{k-1})$$

При этом шаг h уменьшается до тех пор, пока значение функции в точке x_k не станет меньше, чем $f(x_{k-1})$. Только после этого изменения принимаются, и изменяется x . Итерации заканчиваются, когда разность между двумя последовательными значениями x станет меньше по модулю, чем заданная точность ε .

Поскольку необходимо вычислять не только значение функции, но и ее производную, функция **Grad**, реализующая градиентный метод, принимает в параметрах адреса двух функций, начальное приближение, начальный шаг и требуемую точность.

```

float Grad ( func F, func DF, float x, float h, float eps )
{
float fx = F(x), dx;
while ( 1 ) {
    dx = - h * DF(x);           // очередной шаг
    if ( F(x+dx) >= fx) h /= 2.; // шаг неудачный, уменьшаем h
    else {                      // шаг удачный
        x += dx;                // запомнили x
        if (fabs(dx) < eps) break; // условие выхода
        fx = F(x);
    }
}
return x;
}

```

Приведенная выше функция в принципе не всегда находит минимум (например, если функция бесконечно убывает). Это зависит от выбранного начального приближения x_0 . Кроме того, даже если мы нашли минимум, это не гарантирует, что он глобальный, то есть значение функции в этой точке наименьшее. К ее недостаткам надо отнести и то, что требуется вычислять производную функции, что не всегда легко. Иногда приходится делать это численно, заменяя производную функции на величину

$$\frac{f(x) - f(x + \Delta x)}{\Delta x}$$

где Δx — достаточно малая величина. Градиентный метод используется главным образом для минимизации функций нескольких переменных.

📖 Оптимизация для функции нескольких переменных

На практике функции, для которых надо искать минимум или максимум, зависят от многих переменных (чем их больше, тем сложнее задача).

Задача. Найти такую комбинацию параметров $\{x, y, \dots\}$, при которой функция $f(x, y, \dots)$ достигает минимального значения в окрестности точки (x_0, y_0) .

Главная проблема заключается в том, что обычно функция имеет много локальных минимумов и результат зависит от удачного или неудачного выбора начального приближения.

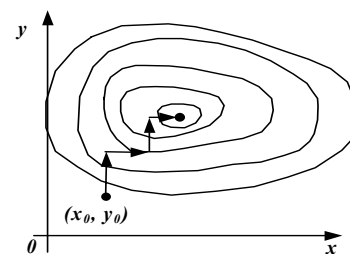
В практических вычислениях используются главным образом следующие три метода оптимизации. Для упрощения мы рассмотрим только случай функции двух переменных без сложных математических выводов.

Метод покоординатного спуска

Выбираем начальное приближение — точку (x_0, y_0) . Сначала фиксируем все координаты, кроме одной. Тогда функция, в которой можно изменять только одну координату, является обычной функцией одной переменной. Ее минимум можно найти, например, с помощью метода золотого сечения.

Изобразим на рисунке *изолинии* — линии равного значения функции двух переменных (как на географических картах). В этом случае минимизация функции при одной изменяемой переменной равносильна движению вдоль одной из осей координат.

Далее сделаем изменяемой другую координату, зафиксировав все остальные. Процедура повторяется до тех пор, пока не будет достигнут минимум, то есть очередной шаг не станет меньше по величине, чем заданная точность.



Градиентный метод

Этот метод связан с математическим понятием *градиента*. **Градиентом** называют вектор, направленный в сторону наискорейшего возрастания функции в данной точке. Следовательно, чтобы найти минимум, надо идти в противоположную сторону. Длина шага часто выбирается из условия минимума функции вдоль направления, противоположного градиенту. Такой вариант называется **методом наискорейшего спуска**.

Метод случайного поиска

Метод случайного поиска показал высокую эффективность в сложных задачах, когда применение других методов затруднительно или неэффективно из-за большого количества параметров и сложности вычисления производной функции.

Его идея заключается в следующем: от исходной точки делаем шаг в случайном направлении. Если в новой точке значение функции меньше прежнего, переходим в нее и продолжаем поиск, если больше, то делаем попытку идти в другом (также случайном) направлении. Если сделано значительное число попыток и нигде значение функции не уменьшается в сравнении с исходной точкой, мы находимся вблизи минимума и надо уменьшить длину шага. Процесс заканчивается, когда величина шага становится меньше заданного значения.

4. Моделирование

Что такое модель?

Модель — это объект, который мы используем для изучения другого объекта (оригинала).

Зачем же использовать какие-то «другие» объекты и почему не исследовать реальный объект? На это есть несколько причин:

- реального объекта может уже (или еще) не быть в действительности, например, при моделировании событий прошлого или при разработке нового технического сооружения;
- проведение экспериментов на реальном объекте очень дорого стоит или может быть опасным и привести к аварии, например, при исследовании подводной лодки или ядерного реактора;
- иногда надо исследовать только какое-то одно свойство оригинала, а реальный объект имеет много взаимосвязанных свойств.

Моделирование — это построение и исследование моделей объектов и процессов.

Перечислим **цели моделирования**, то есть ответим на вопрос, зачем нужны модели.

- **Познание мира.** Человек стремится понять, как устроен мир, выдвигает и проверяет гипотезы о связях и закономерностях в объектах и процессах.
- **Анализ влияния различных воздействий на объект.** Нужно ответить на вопрос, что будет, если изменить некоторым образом исходные данные или условия.
- **Создание объектов с заданными свойствами.** В этом случае основная цель моделирования — определить, как можно создать такой объект.
- **Повышение эффективности управления.** В этом случае модель объекта служит для улучшения характеристик системы, например, для снижения качки судна на волнении.

Виды моделей

Модели можно классифицировать по различным признакам. Один из наиболее важных — способ представления модели. Различают

- **материальные (физические)** модели, которые «можно потрогать», они отражают геометрические и физические свойства оригинала в увеличенном, уменьшенном или просто упрощенном виде; к таким моделям относятся, например, детские игрушки;
- **вербальные (словесные)** модели — информационные модели в мысленной или разговорной форме;
- **структурные** модели — схемы, диаграммы, графики, таблицы;
- **логические** модели, которые описывают выбор одного из вариантов на основе анализа условий;
- **математические** модели — формулы, отображающие связь параметров объектов и процессов.

Нас будет интересовать математическое моделирование. Математические модели бывают **статические** (не учитывающие изменение характеристик объекта во времени) и **динамические**. Наиболее сложны **динамические модели**. В реальных случаях они описываются нелинейными дифференциальными уравнениями, для которых не существует аналитического решения и надо

применять численные методы и моделирование. Моделирование обязательно используется при разработке любых достаточно сложных устройств и в научных исследованиях в двух случаях:

- 1) если нельзя получить аналитическое решение задачи (формулу);
- 2) эксперимент поставить невозможно, или дорого, или опасно.

Иногда применяют моделирование на уменьшенных физических моделях (например, испытание модели корабля в опытовом бассейне), иногда — численное математическое моделирование на компьютере.

Вращение

Рассмотрим самый простой случай: вращение шарика (окружности) вокруг заданной точки.

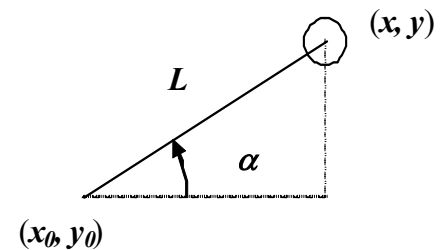
Предварительный анализ

Пусть центр вращения задается координатами x_0 и y_0 , а координаты центра шарика — x и y , радиус орбиты (окружности, по которой вращается шарик) равен L . Тогда при известных x_0 , y_0 , L и угле α , координаты x и y могут быть вычислены из прямоугольного треугольника

$$x = x_0 + L \cos(\alpha)$$

$$y = y_0 - L \sin(\alpha)$$

Независимой переменной в данном случае удобно выбрать угол α , а не координаты объекта. За 1 шаг он изменяется на $\Delta\alpha$. Положительная величина $\Delta\alpha$ означает вращение против часовой стрелки, отрицательная — по часовой.



Такой подход приводит к тому, что положение фигуры фактически задается в полярных координатах с полюсом в центре вращения. Для вращающихся объектов это наиболее естественный и простой способ задания координат.

Программа

```
#include <graphics.h>
#include <math.h>

void Figure ( int x, int y, int color )
{
    const int R = 10; // радиус окружности
    setcolor ( color ); // установить цвет
    circle ( x, y, R ); // рисуем окружность
}

main()
{
    int x0, y0, x, y, L;
    float a, da;
    initwindow ( 400, 400 ); // открыть окно для графики
    x0 = 200; y0 = 200; // координаты центра вращения
    L = 150; // радиус вращения
    a = 0; // начальный угол поворота
    da = 1. * M_PI / 180.; // изменение угла за 1 шаг цикла
    while ( 1 )
```

```

    {
    if ( kbhit() )                // если нажата клавиша
        if ( getch() == 27 ) break; // выход по Esc

    x = x0 + L * cos(a); // считаем координаты
    y = y0 - L * sin(a);

    Figure ( x, y, YELLOW ); // рисуем фигуру
    delay ( 10 );           // задержка
    Figure ( x, y, BLACK ); // стираем фигуру

    a += da;                // вращение: меняем угол поворота
    }
closegraph();
}

```

В программе переменная **a** обозначает угол α , а переменная **da** – изменение угла за 1 шаг цикла. Так как **da** не меняется, то вращение происходит с постоянной угловой скоростью.

Вращение с остановкой

Сделаем вращение замедленным так, чтобы скорость упала до нуля ровно за 5 секунд. Будем считать, что величина задержки при вызове процедуры **delay** равна 10 мс. Тогда за 5 секунд должно выполняться примерно $5000/10=500$ шагов цикла. Мы введем еще одну переменную **dda**, в которой записано изменение скорости вращения, то есть изменение **da**, за 1 шаг цикла. Чтобы скорость упала до нуля ровно за 5 секунд, надо на каждом шаге вычитать из **da** величину, в 500 раз меньшую, чем начальная скорость вращения. Чтобы не началось вращение в обратную сторону, при получении отрицательной скорости будем устанавливать ее в ноль. Основные изменения в программе выглядят так:

```

float a, da, dda;
// без изменения
a = 0;
da = 1. * M_PI / 180.;
dda = - da / 500; // уменьшение скорости за 1 шаг

while ( 1 )
{
// без изменения
a += da;

da += dda; // меняем скорость вращения
if ( da < 0 ) da = 0; // остановка
}

```

Использование массивов

Во многих задачах моделирования и при создании игр удобно использовать массивы. В этом примере мы смоделируем кипение воды в кастрюле. Пузырьки воды начинают подниматься со дна и лопаются, когда доходят до поверхности.

Предварительный анализ

Мы будем считать, что в любой момент в кастрюле (размером с экран) находится 100 пузырьков, и когда какой-то из них лопается (доходит до верхней кромки экрана), вместо него появляется новый пузырек на дне.

Для того, чтобы хранить координаты пузырьков (**x** и **y**), надо использовать два массива по 100 элементов каждый. Перенесем все основные операции в процедуры так, чтобы в основной программе оставить только логику (последовательность действий).

В начале программы надо записать в массивы **X** и **Y** координаты пузырьков так, чтобы они равномерно заполняли весь экран (мы будем считать, что ведем наблюдение через некоторое время после начала кипения). Это можно сделать, например, с помощью случайных чисел.

Основной цикл должен выглядеть так:

- 1) проверяем, не нажата ли клавиша **Esc**, если да, то выходим из цикла;
- 2) рисуем все пузырьки на экране;
- 3) делаем задержку, например, 10 мс;
- 4) стираем пузырьки (то есть, рисуем их черным цветом);
- 5) сдвигаем все пузырьки вверх на заданное число пикселей;
- 6) проверяем, не вышли ли пузырьки за верхнюю границу экрана; если да, то создаем новые пузырьки на дне вместо лопнувших.

Мы будем использовать три процедуры:

Init – начальная расстановка пузырьков;

Draw – рисование всех пузырьков заданным цветом;

Sdvig – сдвиг всех пузырьков вверх;

Zamena – проверяем выход за границу экрана и создаем новые пузырьки вместо лопнувших.

Основная программа

```
#include <graphics.h>
#include <stdlib.h>
const int N = 100;           // количество пузырьков
int X[N], Y[N],             // массивы для хранения координат
    r = 3;                  // радиус пузырька

void Init();                // объявления процедур и функций
void Draw ( int color );
void Sdvig ( int dy );
void Zamena ();
int random(int n) { return rand()%n; }

main()
{
    initwindow ( 800, 600 );
    Init();                 // начальная расстановка
    while ( 1 )
    {
        if ( kbhit() )      // выход по Esc
            if ( getch() == 27 ) break;

        Draw ( YELLOW );   // рисуем пузырьки
        delay ( 10 );       // задержка
        Draw ( BLACK );     // стираем пузырьки
        Sdvig ( 4 );        // вверх на 4 пикселя
        Zamena();           // замена улетевших за пределы экрана
    }

    closegraph();
}
```

Массивы **X** и **Y** необходимо использовать во всех процедурах. Для этого существует два способа: передавать эти массивы в процедуры в качестве параметров или сделать **глобальными**, то есть доступными всем процедурам.

Глобальными называются переменные и массивы, доступные всем процедурам. Глобальные переменные объявляются вне всех процедур и функций, обычно в самом начале файла. При создании они заполняются **нулями**.

В список глобальных переменных мы также включим и радиус пузырьков **r**.

Процедура **Init** до начала основного цикла заполняет массивы координат случайными значениями. Основной цикл содержит вызовы процедур. За 1 шаг цикла пузырьки сдвигаются вверх на 4 пикселя, это число указано в скобках при вызове процедуры **Sdvig**.

📖 Вспомогательные процедуры

Четыре вспомогательные процедуры нужно добавить ниже основной программы.

Процедура **Init** расставляет пузырьки в начальное положение. Нам надо, чтобы все пузырьки находились в границах экрана. Поэтому, например, координаты центра пузырька **x** и **y** должны быть в границах

$$r \leq x \leq 800 - r, \quad r \leq y \leq 600 - r,$$

именно такие диапазоны обеспечиваются приведенными ниже формулами:

```
void Init ()
{ int i;
  for ( i = 0; i < N; i ++ ) // случайная расстановка
  {
    X[i] = random(800 - 2*r) + r;
    Y[i] = random(600 - 2*r) + r;
  }
}
```

С помощью процедуры **Draw** мы можем рисовать или стирать все пузырьки. Их радиус одинаковый и равен **r**, координаты центров находятся в массивах **X** и **Y**, а цвет задается как параметр процедуры. Черный цвет (**BLACK**) означает стирание.

```
void Draw ( int color )
{ int i;
  setcolor ( color );
  for ( i = 0; i < N; i ++ )
    circle ( X[i], Y[i], r );
}
```

Процедура **Sdvig** перемещает все пузырьки вверх на **dy**, уменьшая координату **y**:

```
void Sdvig ( int dy )
{ int i;
  for ( i = 0; i < N; i ++ ) Y[i] -= dy;
}
```

В процедуре **Zamena** проверяем в цикле координату **y** для каждого пузырька и, если она меньше **r**, создается новый пузырек на дне кастрюли.

```
void Zamena ()
{ int i;
  for ( i = 0; i < N; i ++ )
    if ( Y[i] <= r ) {
```



```

        X[i] = random(800 - 2*r) + r;
        Y[i] = 600 - r;
    }
}

```



Математическое моделирование физических процессов



Постановка задачи

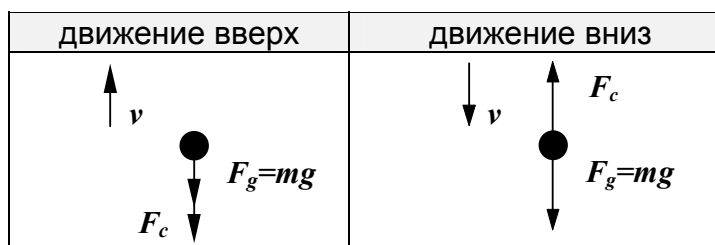
Вертикально вверх со скоростью $v_0 = 60$ м/с вылетает свинцовый шарик диаметром 10 мм (плотность свинца $\rho = 11350$ кг/м³). Определить

1. через какое время и с какой скоростью он вернется обратно;
2. на какую максимальную высоту он поднимется.



Построение полной математической модели

На этом этапе наша задача — построить как можно более полную математическую модель задачи. Основные силы при движении шарика вверх и вниз показаны на рисунках.



Здесь F_g — сила тяжести, которая рассчитывается через массу шарика $m = \rho V$, где ρ — плотность свинца, а V — объем шарика, равный

$$V = \frac{4}{3} \pi r^3$$

где r — радиус шарика. Через F_c обозначена сила сопротивления воздуха. Если скорость шарика не превышает 100 м/с, она может быть рассчитана по формуле Стокса

$$F_c = -6\pi\eta r v,$$

где v — скорость шарика, а η — динамическая вязкость воздуха (Па·с). Вязкость воздуха зависит от высоты, температуры, влажности, ветра и многих других параметров. Обычно в расчетах принимается $\eta \approx 0,022$ Па·с. Знак минус в формуле указывает на то, что направление действия силы сопротивления всегда противоположно направлению вектора скорости.

Кроме того, на шарик действует ветер, но не очень ясно, как его учитывать.



Введение допущений

Полные математические модели очень редко используются при моделировании. Причина в том, что влияние некоторых факторов (например, ветра) просто невозможно точно учесть, и часто ими пренебрегают. Кроме того, учет всех факторов может недопустимо увеличить объем вычислений. В данном примере вводим следующие допущения:

- 1) ускорение свободного падения g постоянно;
- 2) масса и диаметр шарика не меняются;
- 3) влияние ветра не учитывается;
- 4) формула Стокса справедлива, причем вязкость — величина постоянная и равна $\eta \approx 0,022$ Па·с.

Проверим, нельзя ли вообще пренебречь сопротивлением воздуха. Если его нет, то имеем из баланса энергии:

$$\frac{mv_0^2}{2} = mgh_{\max}^0, \quad h_{\max}^0 = \frac{v_0^2}{2g} = \frac{60^2}{2 \cdot 9.81} = 183,5 \text{ м}$$

Общее время полета равно удвоенному времени падения с высоты h_{\max}^0 :

$$t_{\max}^0 = 2\sqrt{\frac{2h_{\max}^0}{g}} = 12,2 \text{ с}$$

Максимальная скорость равна **60 м/с**.

Теперь оценим, что изменится при учете сопротивления воздуха. Явно, что максимальная высота уменьшится. При обратном падении скорость стабилизируется, когда сила сопротивления станет равна силе тяжести, то есть

$$6\pi\eta r v = mg = \rho \frac{4}{3} \pi r^3 g$$

Отсюда следует, что $v_{\max} < \frac{\rho \frac{4}{3} \pi r^3 g}{6\pi\eta r} = 28,11 \text{ м/с}$. Это означает, что при вычислении скорости мы ошиблись по крайней мере в 2 раза. Поэтому сопротивлением воздуха пренебрегать нельзя.

Дискретизация непрерывных процессов

Начальные условия. $h = 0$, $v = v_0 = 60 \text{ м/с}$.

Характер движения. Про движение шарика с учетом сопротивления воздуха нельзя сказать, что оно равномерное или хотя бы равнозамедленное, поскольку скорость меняется, а сила сопротивления воздуха зависит от скорости, таким образом, ускорение не постоянно.

Хотя в каждый момент действующие силы, скорость и ускорение шарика меняются, для моделирования мы считаем, что на каждом очень маленьком интервале времени Δt действующие силы и, следовательно, ускорение, постоянно. Далее, используя формулы для равнозамедленного движения, будем рассчитывать скорость и высоту шарика только в моменты 0 , Δt , $2\Delta t$, $3\Delta t$ и т.д., то есть с шагом Δt . Такой прием называется *дискретизацией*.

Дискретизация – это переход от непрерывной функции к ее дискретным значениям в отдельных точках.

Пусть в момент t известны скорость v и высота h . Положительной будем считать скорость шарика при движении вверх. Тогда ускорение определится разностью сил при данной скорости:

$$a = \frac{-6\pi\eta r v - mg}{m} = \frac{-6\pi\eta r v}{m} - g$$

Тогда, считая, что на интервале $[t, t + \Delta t]$ ускорение постоянно ($a = \text{const}$), по формулам равноускоренного движения рассчитываем скорость и высоту в конце этого интервала:

$$v' = v + a \cdot \Delta t, \quad h' = h + v \cdot \Delta t + \frac{a \cdot \Delta t^2}{2}$$

Далее последовательно рассчитываем все параметры движения шарика до тех пор, пока не получим нулевую (или небольшую отрицательную) высоту — это означает, что шарик вернулся обратно.

Выбор интервала дискретизации Δt . Общих методов решения этой задачи нет. Очевидно, что увеличение интервала дискретизации ускоряет вычисления и снижает точность моделирования и наоборот. Обычно, если из физических соображений неясно, каков должен быть

интервал Δt , проводится серия вычислительных экспериментов, в ходе которой его значение постепенно уменьшается до тех пор, пока результаты моделирования не стабилизируются.

📄 Составление программы

Ниже приводится программа, с помощью которой моделируется движение свинцового шарика, брошенного вертикально вверх. В результате она выведет отрицательную скорость в соответствии с принятым направлением.

```
float ro = 11350., r=5.e-3, eta = 0.022,
      g = 9.81;
float m = ro*4./3.*M_PI*r*r*r, v0 = 60;
float Fc, dt = 0.001, a, v, t, h, hmax = 0;
v = v0; t = 0; h = 0.;
while ( h >= 0 ) {
    Fc = - 6*M_PI*eta*r*v;
    a = Fc/m - g;
    h += v*dt + a*dt*dt;
    v += a*dt;
    t += dt;
    if ( h > hmax ) hmax = h;
}
printf("\nH=%f v=%f t=%f", hmax, v, t );
```

Для этих данных получаем

$$h_{\max} = 79.87 \text{ м}, \quad v = 23.61 \text{ м/с}, \quad t = 8.52 \text{ с.}$$

📄 Особенности вывода на экран

Пусть требуется не только смоделировать полет шарика, но и отобразить его полет на экране в графическом режиме. Для этого будем использовать стандартный прием анимации: рисуем шарик на экране, делаем задержку на 10-20 мс, затем стираем (рисуем в том же месте цветом фона).

Для повышения точности моделирования надо брать маленький шаг Δt , около 0,001 с. Если после каждого шага перерисовывать шарик, он будет лететь очень медленно. Кроме того, скорость его полета будет зависеть от выбранного значения Δt . Чтобы избавиться от этих недостатков, надо ввести новую переменную Δt_{out} – интервал для вывода на экран, который может не совпадать с Δt , причем это никак не скажется на точности результатов. Программа выглядит следующим образом

```
#include <graphics.h>
#include <math.h>
const float K = 5;
main()
{
    float ro = 11350., R = 5.e-3, eta = 0.022, g = 9.81;
    float m = ro*4./3.*M_PI*R*R*R, v0 = 60;
    float Fc, dt = 0.001, a, v, t, h, hmax = 0;
    float dtOut = 0.1, tOut = 0;
    int xe, ye;
    initwindow ( 800, 600 );
    v = v0; t = 0; h = 0.;
```

```

tOut = 0; xe = 400;
while ( h >= 0 ) {
ye = 590 - K*h;
if ( t > tOut ) {
tOut += dtOut;
setcolor ( WHITE ); circle ( xe, ye, 3 );
delay ( 20 );
setcolor ( BLACK ); circle ( xe, ye, 3 );
}

Fc = - 6*M_PI*eta*R*v;
a = Fc/m - g;
h += v*dt + a*dt*dt;
v += a*dt;
t += dt;
if ( h > hmax ) hmax = h;
}
closegraph();
printf("\nH=%6.2f v=%6.2f t=%6.2f", hmax, v, t );
}

```

Обратим внимание на следующие особенности программы:

- 1) введены вспомогательные целые переменные **xe** и **ye**, обозначающие координаты центра шарика на экране (в пикселях);
- 2) для того, чтобы изменять положение шарика на экране, введены целые переменные **dtOut** и **tOut**, обозначающие интервал обновления рисунка и время следующего обновления (при изменении рисунка **tOut** увеличивается на **dtOut**).

Движение на плоскости

Рассмотрим движение шарика в вертикальной плоскости. В некоторый момент известны его координаты (x, y) , скорость v и угол α между вектором скорости и осью **OX**. Требуется рассчитать новые координаты (x', y') , скорость v и угол α через небольшой интервал Δt , в течение которого движение шарика можно считать равноускоренным.

Для решения задачи мы разложим вектора скорости и сил на оси прямоугольной системы координат. На шарик действует две силы: сила тяжести mg и сила сопротивления воздуха

$$F_c = 6\pi\eta r v.$$

На горизонтальную составляющую скорости v_x влияет только сила сопротивления, ее проекция равна

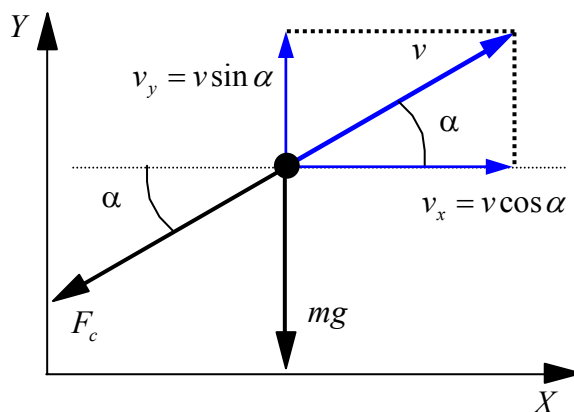
$$F_{cx} = F_c \cos \alpha = 6\pi\eta r v \cos \alpha = 6\pi\eta r v_x.$$

На вертикальную составляющую скорости v_y влияют сила тяжести mg и проекция силы сопротивления

$$F_{cy} = F_c \sin \alpha = 6\pi\eta r v \sin \alpha = 6\pi\eta r v_y.$$

Тогда проекции ускорения на оси координат рассчитываются по формулам

$$a_x = \frac{-F_{cx}}{m}, \quad a_y = \frac{-F_{cy}}{m} - g.$$



Далее мы, как и раньше, предполагаем, что ускорение в течение интервала Δt не меняется, поэтому можно вычислить новые координаты и проекции скорости и по формулам равноускоренного движения

$$x' = x + v \cos \alpha \cdot \Delta t + \frac{a_x \cdot \Delta t^2}{2}, \quad y' = y + v \sin \alpha \cdot \Delta t + \frac{a_y \cdot \Delta t^2}{2}.$$

$$v_x' = v_x + a_x \cdot \Delta t, \quad v_y' = v_y + a_y \cdot \Delta t,$$

После этого вычисляем угол $\alpha' = \arctg \frac{v_y'}{v_x'}$ и новое значение скорости $v' = \sqrt{(v_x')^2 + (v_y')^2}$.

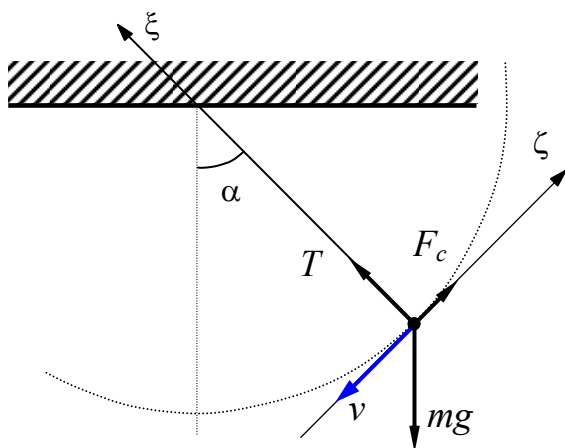
Алгоритм пересчета параметров движения:

Дано: (x, y) , v_x , v_y , α

Найти: (x', y') , v_x' , v_y' , α'

1. Найти проекции силы сопротивления F_{cx} и F_{cy} .
2. Найти проекции ускорения a_x и a_y .
3. Вычислить новые координаты (x', y') .
4. Вычислить новые проекции скорости (v_x', v_y') .
5. Вычислить угол α' и скорость v' .

📖 Движение по окружности



Пусть маятник подвешен на нерастяжимой нити и требуется смоделировать его движение с учетом силы сопротивления среды (шарик будет постепенно останавливаться).

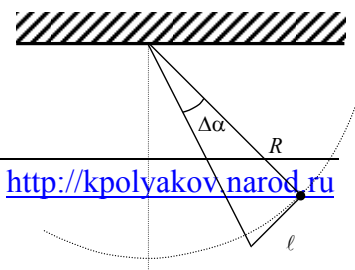
На шарик действует три силы: сила тяжести mg , сила натяжения нити T и сила сопротивления среды F_c . Попытка разложить эти силы с помощью проекций на оси стандартной декартовой системы координат не приводит к успеху – из-за ошибок вычислений (которые неизбежно будут при численном моделировании таким способом) шарик будет «уходить» с окружности.

Шарик в любой момент остается на окружности, поэтому его положение может быть задано одним числом – углом α . Таким образом, в самом деле, эта задача – одномерная. Угловые (полярные) координаты – угол и расстояние от центра (полюса) – являются самым удобным способом для описания любых вращательных движений.

Спроектируем силы на ось ζ , направленную по касательной к окружности, и перпендикулярную ей ось ξ , которая проходит через центр шарика и точку подвеса. При этом проекции сил на ось ξ компенсируются, так как шарик остается на окружности. Изменение скорости шарика определяется только проекцией сил на ось ζ : $F_\zeta = F_c - mg \sin \alpha$, так что ускорение равно

$$a_\zeta = \frac{F_\zeta}{m} = \frac{F_c - mg \sin \alpha}{m} = \frac{F_c}{m} - g \sin \alpha.$$

Как и раньше, мы предполагаем, что ускорение a_ζ постоянно на интервале Δt . Тогда можно, используя формулы для равноускоренного движения, найти скорость и путь, пройденный за это время:



$$v' = v + a_{\zeta} \cdot \Delta t, \quad \ell = v \cdot \Delta t + \frac{a_{\zeta} \cdot \Delta t^2}{2}.$$

Приближенно можно считать, что в течение интервала Δt шарик движется не по окружности, а по прямой. Поэтому из прямоугольного треугольника (см. рисунок) получаем

$$\sin \Delta\alpha \approx \operatorname{tg} \Delta\alpha \approx \frac{\ell}{R}.$$

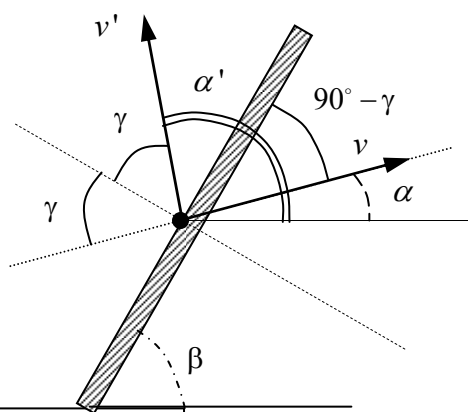
Учитывая, что для малых углов $\sin \Delta\alpha \approx \operatorname{tg} \Delta\alpha \approx \Delta\alpha$, получаем $\Delta\alpha \approx \ell / R$. В итоге алгоритм расчета на каждом шаге моделирования выглядит так:

Алгоритм пересчета параметров движения:

Дано: v, α, R

Найти: v, α

1. Найти проекцию F_{ζ} равнодействующей силы на ось ζ .
2. Найти ускорение a_{ζ} .
3. Вычислить новую скорость v и путь ℓ , пройденный за время Δt .
4. Вычислить изменение угла $\Delta\alpha$.
5. Вычислить $\alpha' = \alpha + \Delta\alpha$



☞ Столкновение

На пути летящего объекта (шарика, снаряда) могут встретиться препятствия. Для моделирования движения в этой ситуации надо уметь, во-первых, определять факт столкновения, и, во-вторых, вычислять угол и скорость после столкновения.

Для того, чтобы определить момент столкновения, нужно найти уравнение прямой, на которой лежит поверхность стенки:

$$y = kx + b,$$

где $k = \operatorname{tg} \beta$ — коэффициент наклона. Если для координат шарика (x, y) выполняется условие $y > kx + b$, то шарик находится «выше» стенки, если $y < kx + b$ — ниже ее. **Момент столкновения** определяется двумя условиями:

$$y = kx + b \quad \text{и} \quad x_0 \leq x \leq x_1.$$

Найдем уравнение нужной прямой. Пусть известны координаты стенки (x_0, y_0) и (x_1, y_1) . Тогда можно определить угол β и коэффициент

$$k : \beta = \operatorname{arctg} \frac{y_1 - y_0}{x_1 - x_0}, \quad k = \frac{y_1 - y_0}{x_1 - x_0}.$$

Постоянная b определяется из условия прохождения прямой через точку (x_0, y_0) :

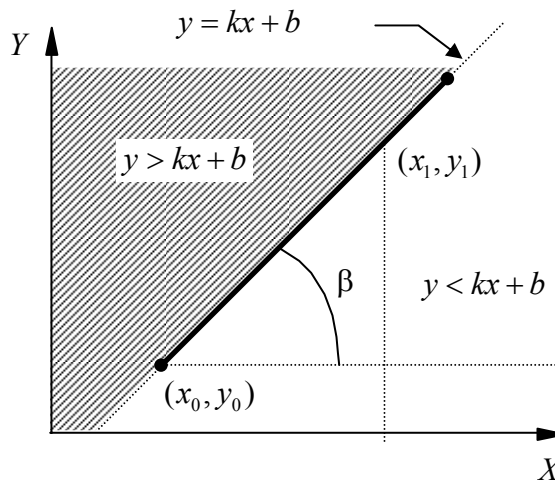
$$y_0 = kx_0 + b \Rightarrow b = y_0 - kx_0.$$

Для этой цели можно также использовать точку (x_1, y_1) :

$$b = y_1 - kx_1.$$

Если известен угол β и одна из точек, сразу можно найти коэффициент наклона $k = \operatorname{tg} \beta$. параметр b вычисляется так же, как и для предыдущего случая, по известной точке.

Для того, чтобы **определить скорость и направление полета после столкновения**, рассмотрим схему справа. Пусть шарик летит под углом α_i к горизонту (это значит, что его вектор



скорости составляет угол α_i с горизонтальным лучом). На его пути находится препятствие в виде плоской стенки, имеющей угол наклона β . Определим угол полета шарика α_{i+1} после отскока от стенки, считая, что угол падения шарика γ (угол между направлением полета и перпендикуляром к плоскости) равен углу отражения. Из элементарных геометрических соображений получаем

$$\beta = \alpha + 90^\circ - \gamma \quad \Rightarrow \quad \gamma = \alpha + 90^\circ - \beta$$

$$\alpha' = \alpha + 2 \cdot (90^\circ - \gamma)$$

Из первого выражения сразу находим γ , а из второго – α' . Все вычисления с углами в программе лучше вести в радианах.

В расчетах надо учесть потерю кинетической энергии при неупругом ударе. Например, если шарик теряет 10% энергии и при ударе его скорость равна v , скорость после отскока определяется из энергетического равенства

$$\frac{m(v')^2}{2} = 0,9 \times \frac{mv^2}{2} \quad \Rightarrow \quad v' = v\sqrt{0,9}$$

Алгоритм пересчета параметров движения:

Дано: v, α, β

Найти: v, α после столкновения.

1. Вычислить угол γ .
2. Вычислить новый угол направления вектора скорости α' .
3. Вычислить модуль вектора скорости v' .

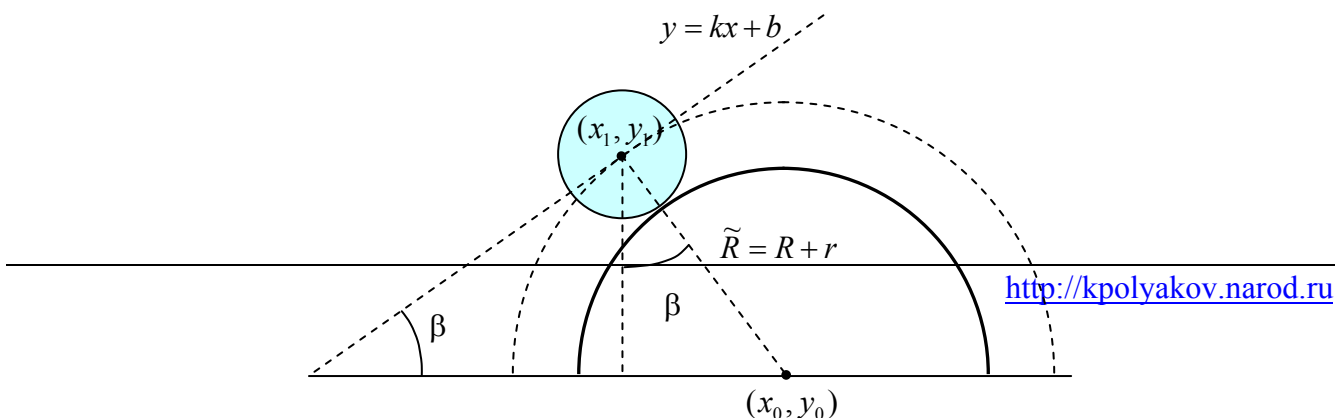
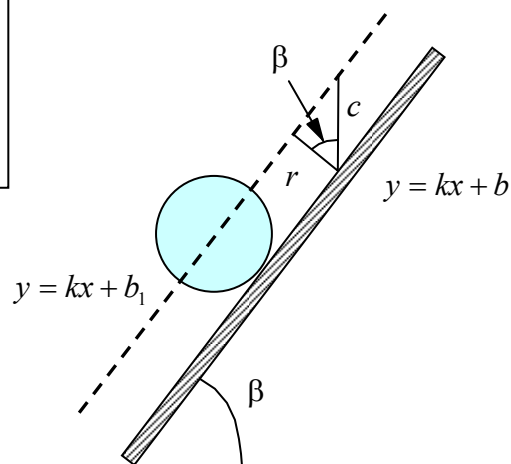
Обратите внимание, что в этих расчетах считается, что изменение скорости и направления происходит мгновенно, т.е., текущее время t и координаты шарика не меняются.

Для того, чтобы **учесть размеры шарика**, надо «сдвинуть» плоскость в сторону шарика на расстояние, равное его радиусу и моделировать отталкивание центра шарика от этой невидимой плоскости (штриховая линия на рисунке). По рисунку видно, что штриховая линия $y = kx + b_1$ «поднимается» относительно исходной на расстояние c , которое можно вычислить из прямоугольного треугольника

$$c = \frac{r}{\cos \beta}.$$

Таким образом, $b_1 = b + c = b + \frac{r}{\cos \beta}$.

Столкновение со сферой рассчитывается так же, как и столкновение с плоскостью, касательной к этой сфере в точке удара. Для того, чтобы учесть размеры шарика, надо моделировать отталкивание центра от невидимой сферы, радиус которой \tilde{R} равен сумме радиусов шарика r и реальной сферы R .



Пусть известны координаты центра сферы (x_0, y_0) и координаты центра шарика (x, y) . При столкновении должно выполняться равенство

$$(x - x_0)^2 + (y - y_0)^2 \approx (R + r)^2.$$

Тогда коэффициент наклона касательной (имеющей уравнение $y = kx + b$) определяется из прямоугольного треугольника (см. рисунок)

$$k = \operatorname{tg} \beta = \frac{x_0 - x_1}{y_0 - y_1},$$

а коэффициент b можно определить из условия прохождения касательной через точку (x, y) :

$$y_1 = kx_1 + b \Rightarrow b = y_1 - kx_1.$$

Дальше расчет изменений скорости и угла при столкновении выполняется так же, как и при отскоке от плоскости.

5. Сложные проекты

Зачем нужны проекты?

Все наши прошлые программы представляли собой один единственный файл. Реальные программы состоят из множества отдельных файлов, которые включают тысячи строк кода и используют дополнительные функции, записанные в библиотеках (в *Dev-C++* библиотеки имеют расширение ***.a**). Для того, чтобы построить из всех этих файлов программу (исполняемый файл с расширением ***.exe**) используют **проекты**.

Проект – это файл, в котором определяется, из каких файлов и как именно (с какими параметрами компилятора и компоновщика) собирается программа.

Проекты служат для того, чтобы можно было

- собрать программу из нескольких модулей, каждый из которых записывается в отдельный файл и может отлаживаться независимо от других;
- подключить функции из библиотек, созданных другими программистами.

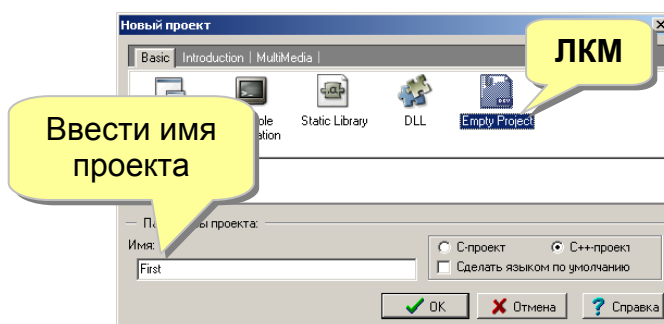
Рекомендуется разбивать программу на модули так, чтобы длина каждого модуля была не более 100-200 строк, иначе становится сложно искать в нем нужную функцию или процедуру.

В *Dev-C++* файлы проектов имеют расширение ***.dev**. Для каждого проекта желательно выделять отдельную папку (каталог), потому что иначе будет очень сложно разбираться в многочисленных файлах (какие к какому проекту относятся).

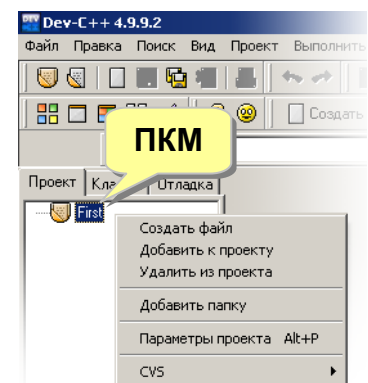
Как создать проект?

Сначала создадим простейший проект, в который будет входить всего один файл. Для этого надо выполнить следующие действия:

- выбрать пункт верхнего меню *Файл – Создать – Проект*;
- в появившемся окне выбрать тип проекта *Empty Project* (пустой проект), ввести имя проекта и щелкнуть по кнопке *OK*:



- появится окно сохранения проекта; здесь нужно выбрать отдельную папку и сохранить в нее файл проекта **First.dev**;
- теперь в левой части окна нужно включить вкладку *Проект*, где появится название проекта (папка); если нажать на правую кнопку мыши, появляется контекстное меню, в котором можно *Создать файл* (новый), *Добавить к проекту* (существующий файл), *Удалить из проекта* (файл, кото-



рый уже есть в проекте).

В простейшем случае нужно выбрать пункт *Создать файл* и писать программу так же, как мы делали раньше. При нажатии клавиши **F9** проект собирается и запускается. Заметим, что хотя формально мы построили проект, но для программы из одного файла он не нужен.

Если проект уже есть, его можно загрузить в память с помощью пункта меню *Файл – Открыть файл или проект*. Кроме того, в меню *Файл – Открыть заново* хранятся имена файлов и проектов, которые открывались последними (для того, чтобы их можно было быстро загрузить).

Пример проекта

Ранее мы написали [программу](#), которая моделирует кипение воды в кастрюле (задача с пузырьками, поднимающимися вверх). Теперь напишем другой вариант этой же программы, в котором функции будут вынесены в отдельный файл. Поскольку теперь программа состоит из нескольких (двух) частей, нужно создавать проект.

Начнем новый проект **Bubble.dev** и запишем его в отдельную папку. Создадим два новых файла:

main.cpp основная программа
func.cpp вспомогательные функции

Основная программа

Файл **main.cpp** практически повторяет старую программу, но не содержит процедур. Тем не менее, в начале программы нужно вставить *объявления* всех используемых процедур (заголовки с точкой с запятой в конце). Этим мы скажем транслятору, что такие процедуры где-то действительно есть (в других файлах или в библиотеках), и определим типы их параметров.

```
#include <graphics.h>
void Init(); // объявления процедур и функций
void Draw ( int color );
void Sdvig ( int dy );
void Zamena ();
int random(int n);

main()
{
    initwindow ( 800, 600 );
    Init(); // начальная расстановка
    while ( 1 ) {
        if ( kbhit() ) // выход по Esc
            if ( getch() == 27 ) break;
        Draw ( YELLOW ); // рисуем пузырьки
        delay ( 10 ); // задержка
        Draw ( BLACK ); // стираем пузырьки
        Sdvig ( 4 ); // вверх на 4 пикселя
        Zamena (); // замена улетевших за пределы экрана
    }
    closegraph ();
}
```

Обратите внимание, что основная программа не использует константу **N**, массивы **X** и **Y**, а также радиус пузырька **r**. Они нужны только в процедурах и будут введены во втором файле.

Процедуры и функции

Если просто поместить в файл `func.cpp` тексты функции `random` и всех процедур, при трансляции мы получим сообщения об ошибках, потому что

- 1) графические функции (`setcolor` и др.) неизвестны;
- 2) константа `N`, массивы `X` и `Y`, а также радиус пузырьков `r` неизвестны.

Чтобы убрать первую группу ошибок, нужно подключить заголовочный файл `graphics.h`. Ошибки второго типа устраняются, если в начале файла `func.cpp` объявить все необходимые константы, переменные и массивы:

```
#include<graphics.h>
const int N = 100;
int X[N], Y[N], r = 3;

// random - случайные числа в интервале [0,n-1]
int random(int n) { return rand() % n; }

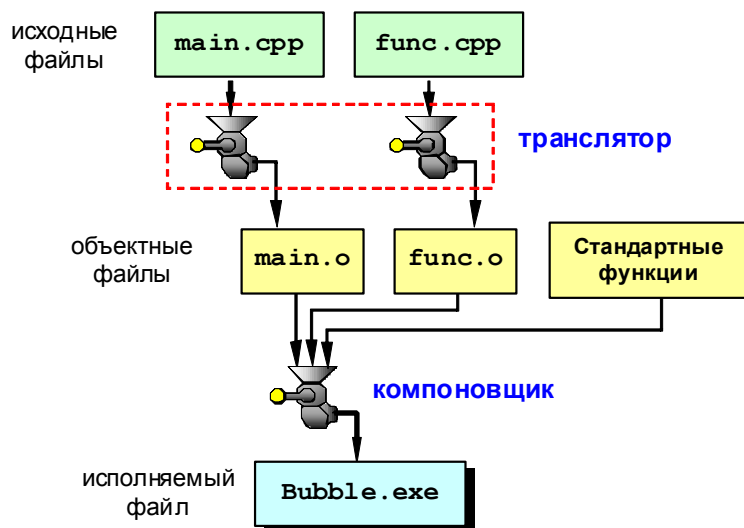
// Init - начальная случайная расстановка
void Init ()
{ int i;
  for ( i = 0; i < N; i ++ ) {
    X[i] = random(800 - 2*r) + r;
    Y[i] = random(600 - 2*r) + r;
  }
}

// Draw - рисование и стирание
void Draw ( int color )
{ int i;
  setcolor ( color );
  for ( i = 0; i < N; i ++ )
    circle ( X[i], Y[i], r );
}

// Sdvig - перемещение вверх
void Sdvig ( int dy )
{ int i;
  for ( i = 0; i < N; i ++ ) Y[i] -= dy;
}

// Zamena - вместо вылетевших появляются новые
void Zamena ()
{ int i;
  for ( i = 0; i < N; i ++ )
    if ( Y[i] <= r ) {
      X[i] = random(800 - 2*r) + r;
      Y[i] = 600 - r;
    }
}
```

Теперь можно нажать кнопку **F9** и запустить проект. Транслятор сначала обработает два исходных файла (`main.cpp` и `func.cpp`) и создаст из них два объектных файла с расширением `*.o` (`main.o` и `func.o`). Затем соберет эти объектные файлы вместе со стандартными функциями и построит исполняемый файл `Bubble.exe` (потому что проект мы назвали `Bubble.dev`).



Глобальные переменные

Мы уже говорили о том, что одна из главных задач проекта — разбить большую программу на несколько частей, каждую из которых можно отлаживать отдельно (этим могут заниматься даже разные люди). При этом возникают некоторые сложности, если функции и процедуры в разных модулях (так называют отдельные файлы, входящие в проект) должны использовать общие глобальные данные или структуры данных. Чтобы в таких ситуациях избежать ошибок, надо помнить два простых правила:

1. В одном модуле (там, где выделяется память под глобальные переменные) они объявляются так же, как и обычно.
2. Во всех остальных модулях, использующих глобальные переменные, перед их объявлением ставится ключевое слово **extern** (от англ. *external* – внешний). Это означает, что они располагаются в каком-то другом модуле.

Например, если в основной программе нужно обращаться к массивам **X** и **Y**, которые объявлены (размещены в памяти) в модуле **func.cpp**, в начало файла **main.cpp** нужно добавить такую строчку:

```
extern int X[], Y[];
```

Размеры массивов указывать не нужно, потому что память тут не выделяется. Начальные значения глобальных переменных можно задавать только в том модуле, где они размещаются в памяти (то есть объявляются без ключевого слова **extern**).

Общие заголовочные файлы

Как вы знаете, любые переменные, которые используются в программе, необходимо объявить. Так же любые процедуры и функции надо объявить в каждом модуле до того, как их вызовы встречаются в тексте модуля.

Конечно, можно в начале каждого модуля добавить вручную объявления всех глобальных переменных с ключевым словом **extern** и объявления всех нужных функций. Однако лучше избавить себя от лишней работы с помощью заголовочных файлов.

Практически любая программа начинается с директивы **#include**, с помощью которой подключаются заголовочные файлы с расширением ***.h**. Стандартные заголовочные файлы, которые поставляются с транслятором, хранятся (по умолчанию) в папке **C:\Dev-Cpp\include**. Такой файл можно посмотреть любым текстовым редактором, при этом вы увидите, что он содержит как раз объявления глобальных констант и функций.

Язык Си позволяет вам создавать свои заголовочные файлы и подключать их к проекту. В заголовочный файл обычно включают объявления всех глобальных переменных (с ключевым словом **extern**) и объявления всех функций, используемых в проекте. Можно, конечно, записать ваш собственный заголовочный файл в каталог **C:\Dev-Cpp\include** и подключать его так же, как и стандартные файлы. Тем не менее, лучше не путать «свою шерсть с государственной» и свои файлы со стандартными. Дополнительные заголовочные файлы обычно записывают в ту же папку, где находятся все остальные файлы проекта. Важно, что в программе их имена записываются не в угловых скобках, а в кавычках, что означает «искать в текущем каталоге».

Например, начало основной программы в проекте с пузырьками могло бы выглядеть так:

```
#include <graphics.h>
#include "bubble.h"    // свой заголовочный файл
main()
{
    ...
}
```

А вот сам заголовочный файл **bubble.h**:

```
void Init ();
void Draw ( int color );
void Sdvig ( int dy );
void Zamena ();
int random(int n);
```

Файл **bubble.h** можно добавить к проекту, но, в принципе, можно этого и не делать – при трансляции программа найдет его в том же каталоге, где лежат исходные файлы.