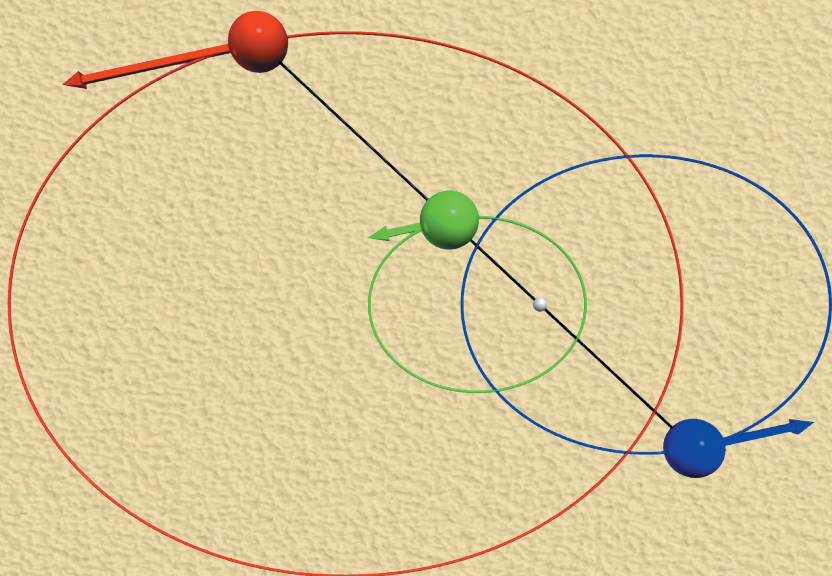




М. И. Карякин

**ВИЗУАЛИЗАЦИЯ  
МЕХАНИЧЕСКИХ СИСТЕМ,  
ПРОЦЕССОВ И ЯВЛЕНИЙ:  
проектные задания  
с использованием VPython**



МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ  
РОССИЙСКОЙ ФЕДЕРАЦИИ  
Федеральное государственное автономное образовательное  
учреждение высшего образования  
«ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

**М. И. Карякин**

**Визуализация механических систем,  
процессов и явлений:  
проектные задания с использованием VPython**

*Учебное пособие*

Ростов-на-Дону – Таганрог  
Издательство Южного федерального университета  
2021

УДК [532.5+539.3]:004.424(075.8)

ББК 22.251+22.253+32.973 я73

К27

*Печатается по решению кафедры теории упругости  
Института математики, механики и компьютерных наук им. И. И. Воровича  
Южного федерального университета (протокол № 9 от 26 апреля 2021 г.)*

**Рецензенты:**

заведующий кафедрой «Теоретическая и прикладная механика»

Донского государственного технического университета,

доктор физико-математических наук, профессор *А. Н. Соловьев*;

профессор теоретической и компьютерной гидроаэродинамики

Южного федерального университета,

доктор физико-математических наук, профессор *М. А. Сумбатян*

Карякин, М. И.

К27

Визуализация механических систем, процессов и явлений:  
проектные задания с использованием Python : учебное пособие /  
М. И. Карякин ; Южный федеральный университет. – Ростов-на-  
Дону ; Таганрог : Издательство Южного федерального универси-  
тета, 2021. – 244 с.

ISBN 978-5-9275-3827-0

Учебное пособие предназначено для студентов вузов, изучающих современные курсы или разделы курсов, связанные с моделированием явлений, происходящих в деформируемых телах и средах, с использованием математических и компьютерных методов, основанных на динамике частиц и ее приложениях. Оно может также быть использовано в качестве дополнительной литературы при изучении бакалаврских и магистерских курсов по основам алгоритмизации, программированию, теоретической механике, математическому моделированию, современным концепциям естествознания и аналогичным дисциплинам.

УДК [532.5+539.3]:004.424(075.8)

ББК 22.251+22.253+32.973 я73

ISBN 978-5-9275-3827-0

© Южный федеральный университет, 2021

© Карякин М. И., 2021

## ПРЕДИСЛОВИЕ

Учебное пособие было написано для слушателей курса «Динамика частиц и ее приложения», реализуемого в Южном федеральном университете в рамках магистерской программы «Фундаментальная математика, механика и математическое моделирование», с целью познакомить их с базовыми возможностями пакета VPython для визуализации динамических процессов, происходящих в механических системах. Однако в процессе написания стало понятно, что круг его потенциальных читателей может быть значительно расширен.

Представленный материал будет полезен изучающим современные курсы или разделы курсов, связанные с моделированием явлений, происходящих в деформируемых телах и средах, с использованием математических и компьютерных методов, основанных на динамике частиц и ее приложениях. Пособие может быть использовано в качестве дополнительной литературы при изучении бакалаврских и магистерских курсов по основам алгоритмизации, программированию, теоретической механике, современным концепциям естествознания и аналогичных дисциплин. Изложенный материал позволит обучающимся познакомиться с широким спектром современных задач математического моделирования, относящихся к совершенно разным областям человеческого знания – от объектов микромира до астрономических систем. Объединяющим столь разнородные сферы науки как раз и являются модели, основанные на динамике частиц – классическом и подробно изученном разделе механики. И хотя в данном пособии не будет детально представлен богатый математический аппарат этой теории, за кадром останутся многие тонкие и интересные механические факты и явления, автор надеется, что ему удастся показать, как достаточно традиционные и устоявшиеся идеи и методы одной науки могут оказаться востребованными и даже привести к прорывным результатам в другой.

Две главы этого пособия можно считать в каком-то смысле информационными. Глава 1 «Использование пакета VPython для создания 3D-анимаций» содержит подробное описание основных функций графической библиотеки и демонстрирует их использование для создания простейших 3D-изображений и анимаций, в том числе интерактивных.

Глава 3 «Моделирование движения механических систем» посвящена краткому описанию вычислительных алгоритмов, рекомендованных для использования при выполнении проектных заданий. Алгоритмы проиллюстрированы их применением к моделированию динамики нескольких механических систем. Визуализация процессов при этом осуществляется с использованием VPython.

Остальные главы содержат постановки проектных заданий, которые могут выполняться как индивидуально, так и небольшими группами обучающихся. Таких заданий – четыре: «Визуализация молекулы», «Пружинная модель твердого тела», «Небесная механика», «Знакомство с молекулярной динамикой». Все главы с проектными заданиями имеют похожую структуру. В разделе «Формулировка задания» содержится постановка задачи, набор требований к методам решения и ожидаемым результатам, параметры отчета и т. п. В этом же разделе приводятся несколько (порядка двадцати) вариантов проектных заданий. Предполагается, что номер задания для каждого обучающегося или группы определяется преподавателем. Следующий раздел, «Базовые сведения», содержит общую информацию по теме проектного задания, в ряде случаев выходящую за рамки конкретной задачи. Первичная цель этого раздела – снабдить обучающегося основными методиками и алгоритмами для выполнения задания. Более общая, и не менее важная цель состоит в том, чтобы очертить предметную область, связанную с конкретной задачей, познакомить с существующими в этой области проблемами и актуальными технологиями. Размер этого раздела может варьироваться в зависимости от типа проектного задания. Третий раздел – «Пример выполнения задания» – содержит текст или фрагменты текста программы, демонстрирующий решение основных типовых проблем, возникающих при реализации проекта, а также комментарии к приведенному коду.

Завершая краткое предисловие, нужно еще ответить на естественный вопрос: почему в качестве средства реализации вычислительных алгоритмов и визуализации механических процессов выбран язык Питон?

Причины достаточно просты и лежат на поверхности. Питон сегодня де факто является своеобразным *эсперанто* современной междисциплинарной науки. Специалисты в области математического моделирования, физики, астрономии, химии, биологии, биоинформатики, лингвистики, машинного обучения, анализа данных, статистики «говорят»

на этом языке, в том смысле, что количество пакетов и библиотек Питона, ориентированных на эти и многие другие сферы исследований, продолжает постоянно увеличиваться. В различных частях этого пособия будут упомянуты астрономические пакеты, пакеты молекулярной динамики, библиотеки визуализации структуры химических веществ, полностью разработанные на Питоне, или использующие Питон в качестве интерфейсного средства. Не случайно поэтому, что практически во всех известных «рейтингах» современных языков программирования Питон уверенно входит в тройку лидеров. Например, июньский рейтинг Tiobe 2021 года<sup>1</sup> озаглавлен так: «Питон как никогда оказался близок к первому месту» (*Python has never been so close to position #1 before*). В процессе выполнения разнообразных проектных заданий, представленных в данном пособии, обучающиеся могут достаточно быстро познакомиться с мощностью и гибкостью этого языка, что позволит им впоследствии успешнее ориентироваться и погружаться в самые различные области современной прикладной математики и информатики. Завершая ответ, приведем цитату из статьи в Википедии, посвященной этому языку<sup>2</sup>: *«За счёт читабельности, простого синтаксиса и отсутствия необходимости в компиляции язык хорошо подходит для обучения программированию, позволяя концентрироваться на изучении алгоритмов, концептов и парадигм.»*

Все приведенные в данном учебном пособии программы носят исключительно демонстрационный характер. Так, в них практически отсутствуют операторы ввода (все необходимы параметры задаются прямо в тексте программ), не проводится никакой проверки на возможные ошибки, отсутствует анализ исключений. Это связано и с желанием сэкономить место, и с тем, что приведенный код, как правило, иллюстрирует какую-то одну вычислительную идею или схему визуализации. Ну и кроме того, в данном случае пользователь и программист являются одним и тем же лицом. И если этот программист/пользователь снабдил самого себя неправильным входным файлом, или ошибся в константах и т. п., то, будем надеяться, он сможет осознать это, проанализировав реакцию интерпретатора языка или сообщения об ошибках времени выполнения. Чтобы исключить

---

<sup>1</sup> <https://www.tiobe.com/tiobe-index/>

<sup>2</sup> <https://ru.wikipedia.org/wiki/Python>

дублирование и уменьшить объем пособия, из текстов программ удалена часть комментариев, поскольку они присутствуют в тексте, сопровождающем фрагменты кода.

Большая часть приведенных литературных источников находится в открытом доступе, ряд статей, учебников и монографий доступен в рамках вузовской подписки. Списки цитируемой литературы сгруппированы отдельно по каждой главе; их нумерация, поэтому, является составной: номер источника включает и номер главы, отделенный точкой. Для остальных же объектов – формул, таблиц, рисунков и фрагментов кода принята сквозная нумерация.

# ГЛАВА 1. ИСПОЛЬЗОВАНИЕ ПАКЕТА VPYTHON ДЛЯ СОЗДАНИЯ 3D-АНИМАЦИЙ

## ВВЕДЕНИЕ

Библиотека VPython предназначена для существенного облегчения труда программиста, разрабатывающего простейшие 3D-модели и демонстрационные ролики. Не претендуя на сверхвысокую эффективность и мощь графического движка, она позволяет быстро создавать динамические иллюстрации к учебным курсам, визуализировать механические, физические и химические явления и процессы, наглядно представлять бизнес-схемы, обеспечивая при этом достаточный уровень интерактивности. Использование языка Питон в качестве основы позволяет пользователю, даже не являющемуся профессиональным программистом, достаточно быстро освоить основные функции и оперативно приступить к работе.

Проект VPython был начат Давидом Шерером (David Scherer) в 2000 году. В 2011 году вместе с Брюсом Шервудом (Bruce Sherwood) они начали разрабатывать среду GlowScript – программную оболочку для VPython, работающую в окне браузера. С 2014 года в среде GlowScript стало возможно использовать язык программирования RapidScript, очень близкий к стандартному Питону. Одновременно с этим группой программистов под руководством Джона Коуди (John Coady) началось продолжающееся и по сей день развитие пакета VPython 7, являющегося классической библиотекой для стандартного Питона, позволяющего тем самым использовать любые другие библиотеки Питона, в том числе и сторонние.

VPython 7 поначалу был ориентирован на применение в известной среде Jupyter notebook. С 2017 года использующие этот пакет программы могут запускаться из стандартной среды IDLE или из Spyder, при этом окно с 3D-анимацией отображается во вкладке системного браузера.

В настоящее время разработчики развивают и поддерживают обе версии своего продукта: и GlowScript VPython, и VPython 7. Обе реализации используют одну и ту же библиотеку трехмерной графики WebGL, которая позволяет решать достаточно сложные задачи построения



изображений с использования современного графического оборудования в среде браузера. Сами разработчики в качестве основных преимуществ браузерной графики отмечают отсутствие кода, специфического для операционной системы, и исчезновение необходимости в инсталляторах. Кроме того, по их оценкам, многие библиотеки, ориентированные на использование в браузерах, являются заметно лучше поддерживаемыми их авторами.

Что касается выбора одной из двух версий, то в последнее время новые пользователи VPython (а это, как правило, студенты) чаще выбирают браузерную реализацию, т.е. GlowScript. Основной причиной такого выбора является, по-видимому, отсутствие необходимости установки какого бы то ни было программного обеспечения, даже языка программирования Питон. Коды программ хранятся в облаке, ими можно легко обмениваться через гиперссылки, а также встраивать в другие веб-страницы. Библиотекой VPython 7, по мнению ее авторов, чаще пользуются профессионалы-исследователи, преподаватели и разработчики научного ПО, поскольку им для работы часто необходима вся мощь питоновских библиотек, которые не могут быть доступны<sup>3</sup> в среде GlowScript, основанной на JavaScript.

В современной зарубежной учебной литературе по механике и физике VPython используется очень широко; примером этому могут служить школьные курсы физики [1.7], целые университетские курсы [1.6, 1.10] или отдельные лабораторные практикумы [1.9]. Что касается русскоязычной литературы по этому пакету, то ее количество достаточно ограничено; безусловного упоминания в этой связи заслуживают работы Е. Е. Германовой [1.1, 1.2]. Данная глава настоящего пособия, призванная в какой-то мере исправить ситуацию, основывается, прежде всего, на официальном руководстве по пакету [1.5] и некотором опыте автора.

## ПЕРВОЕ ЗНАКОМСТВО

**Установка.** Чтобы начать работать с браузерной версией VPython, необходимо зарегистрироваться на сайте [glowscript.org](http://glowscript.org), а затем в разделе **MyPrograms** выбрать ссылку на ваши программы, а затем щелкнуть на

---

<sup>3</sup> Исключением является модуль `random`, реализованный и поддерживаемый компилятором RapidScript-NG. Кроме того, распространенные математические функции интегрированы в VPython, поэтому для их вызова не обязательно использовать модуль `math`.

ссылку **Create New Program** (создать новую программу). В окне браузера появится пустая страница создания новой программы с единственной строчкой вида

```
GlowScript 3.0 VPython.
```

Код программы вводится, начиная со второй строчки; для ее выполнения используется пункт меню **Run this program**.

Для использования VPython как модуля Питона его сначала следует установить. Этот процесс может оказаться не совсем простым, он реализован по-разному для разных дистрибутивов Питона. Далее в этом пособии будем полагать, что VPython устанавливается на стандартный дистрибутив Питона 3.xx, загруженный с сайта [www.python.org](http://www.python.org). В этом случае установка модуля может быть осуществлена командой

```
> pip install vpython
```

Процесс инсталляции гарантированно пройдет успешно<sup>4</sup> при условии, что в системе установлен компилятор Visual Studio языка C++. Подробный обзор остальных вариантов установки дан в официальной документации по пакету <https://vpython.org/presentation2018/install.html>

Подключение модуля VPython к программе на языке Питон осуществляется стандартным образом, например

```
from vpython import *
```

Именно такой вариант подключения, обеспечивающий использование всех функций пакета с их прямыми именами без использования префиксов, будет принят по умолчанию во многих примерах этого учебного пособия. Пакет GlowScript неявно использует именно такой же вариант подключения модуля VPython.

**Холст.** Как и большинство графических библиотек, пакет VPython при отображении трехмерных объектов на экране использует концепцию «холста» (canvas). Его центр имеет координаты (0, 0, 0), ось x направлена вправо, ось y – вертикально вверх (как в учебниках по математике), а ось z направлена перпендикулярно экрану в сторону зрителя (рис. 1).

---

<sup>4</sup> Начиная с версии 7.6 (январь 2020 года) в большинстве случаев установка для операционных систем Windows 7 и Windows 10 осуществляется успешно и при отсутствии этого компилятора

По умолчанию холст масштабируется автоматически так, чтобы были видны все имеющиеся на нем объекты. Это облегчает задачу отображения – можно пользоваться любыми размерными единицами длины, при условии, что величины всех длин и расстояний используют одинаковую размерность.

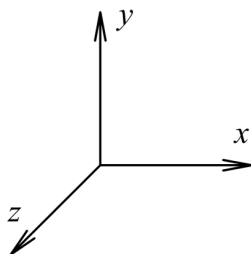


Рис. 1. Система координат VPython

В качестве примера отображения графики на рисунке 2 приведен результат выполнения следующего кода<sup>5</sup>, создающего холст белого цвета разрешением 500x400 пикселей (строчки 1–2) и изображающего на нем красный параллелепипед (строчки 3–4) и зеленый шар (строчки 5–6).

Фрагмент кода 1

```
1 scene = canvas(width=500,height=400,  
2             background=vector(1,1,1))  
3 redbox = box(pos=vector(0,0,0),  
4             size=vector(8,4,6), color=color.red)  
5 ball = sphere(pos=vector(0,6,0),  
6             radius=2, color=color.green)
```

Получившееся изображение выглядит не очень трехмерным. Для перемещения «камеры», то есть позиции наблюдателя, необходимо нажать правую кнопку мыши в пределах холста и, удерживая ее, перемещать указатель. Аналогичного эффекта можно добиться при удерживании клавиши Ctrl и перемещении указателя при нажатой левой кнопки мыши. Перетаскивание левой кнопкой мыши при зажатой клавише Shift приводит к перемещению камеры в одной плоскости (изображение при этом движется «параллельно» экрану). Удалять или приближать изображение

---

<sup>5</sup> Как было отмечено выше, во всех примерах кода опущена строчка `from vpython import *`

можно вращением колесика мыши или перемещением мыши при одновременно нажатых левой и правой кнопках.

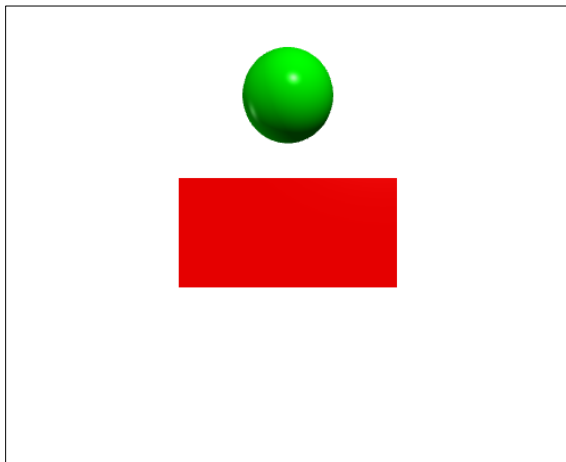


Рис. 2. Пример графического окна

На рисунке 3 показана другая проекция того же изображения, полученная в результате некоторых из указанных манипуляций. Оси и надписи на эту проекцию нанесены для иллюстрации параметров функций `box` и `sphere` из примера кода 1, связанных с положением и размером.

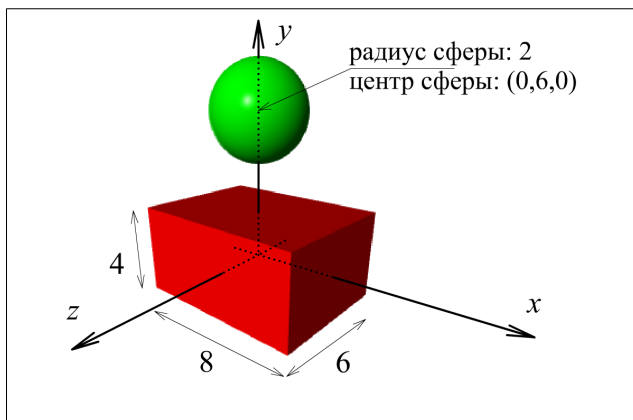


Рис. 3. 3D-объекты и их параметры

**Анимация.** Чтобы заставить шар в приведенном выше примере двигаться, например, «падать вниз», в направлении параллелепипеда, необходимо поочередно выводить на экран «кадры», соответствующие новому положению объекта. Для этого в рассмотрение следует ввести две величины – скорость движения и время. В качестве примера во фрагменте кода 2 (который будет работоспособным, если вставить его непосредственно за кодом из фрагмента 1) скорость принята постоянной величиной, направленной вниз, т.е. противоположной направлению оси  $Oy$ ; в качестве величины вектора скорости выбрано единичное значение (строка 1). Чтобы задать интервал времени между «кадрами» выберем некоторое достаточно малое значение  $\Delta t$ , например, 0.01 (строка 2). Дальнейшая схема очевидна – задаем начальное и конечное время движения (строки 3, 4), а затем в цикле увеличиваем текущее значение времени на  $\Delta t$  и корректируем положение шара: *пройденное расстояние = скорость × время*.

Особого упоминания в приведенном ниже примере заслуживает последняя строка. Функция `rate(частота)` останавливает вычисления так, чтобы с момента ее предыдущего вызова прошло не менее, чем  $1/\text{частота}$  секунд. В данном случае `rate(100)` приостановит вычисления до тех пор, пока с момента предыдущего вызова не пройдет как минимум  $1/100 = 0.01$  секунды. Если это время уже будет потрачено (например, на прорисовку графики), то дополнительной задержки не произойдет. Таким образом, наш цикл будет выполняться не чаще, чем 100 раз в секунду, даже на самом мощном компьютере. Отсутствие же вызова этой функции приведет к тому, что пользователь увидит на экране лишь конечное положение шара – потому что таковы скорости современных компьютеров.

Альтернативой является использование функции `sleep`: `sleep(0.01)` означает "ничего не делать в течение 0.01 секунды" и поэтому эквивалентна вызову `rate(100)`. Нужно отметить, что во время «простоя», вызванного функцией `sleep`, программа все-таки периодически занимается отрисовкой сцены и обработкой событий мыши, позволяя, в частности, масштабировать и вращать изображение.

Нужно отметить, что в текущей версии пакета `VPython 7` функция `rate` работает не всегда надежно: ей требуется некоторое время –

несколько вызовов – прежде чем таймер начнет работать корректно. При использовании GlowScript таких проблем замечено не было.

Фрагмент кода 2

```
1 | ball.velocity = vector(0,-1,0)
2 | delta_t = 0.01
3 | t = 0
4 | t_final = 3
5 | while t < t_final:
6 |     t = t + delta_t
7 |     ball.pos += ball.velocity*delta_t
8 |     t += delta_t
9 |     rate(100)
```

## МАТЕМАТИЧЕСКИЙ АППАРАТ VPYTHON

**Математические функции.** Для обеспечения независимости программ, использующих модуль `vpython`, от остальных библиотек (что особо актуально для GlowScript-реализации), в нем реализованы основные математические функции, частично заменяющие их аналоги из стандартной библиотеки `math`. Их описания приведены ниже в Таблице 1.

Таблица 1. Математические функции и константы VPython

Функция	Возвращаемое значение
<code>sqrt(x)</code>	$\sqrt{x}$
<code>sin(x)</code>	$\sin x$
<code>cos(x)</code>	$\cos x$
<code>tan(x)</code>	$\operatorname{tg} x$
<code>asin(x)</code>	$\arcsin x$
<code>acos(x)</code>	$\arccos x$
<code>atan(x)</code>	$\operatorname{arctg} x$ , возвращает значения в диапазоне $(-\pi/2, \pi/2)$
<code>atan2(y, x)</code>	Угол, между положительным направлением оси $Ox$ и вектором, имеющим координаты $(x, y)$ . Возвращает значение в диапазоне $[-\pi, \pi]$ . Для положительных значений аргумента $x$ эта функция

	эквивалентна $\text{atan}(y/x)$ , при $x=0$ возвращает приближенное значение $\pi/2$ .
<code>exp(x)</code>	$e^x$
<code>log(x)</code>	$\ln x$ Для вычисления десятичного логарифма можно пользоваться формулой $\log x = \ln x / \ln 10$ .
<code>pow(x, y)</code>	$x^y$
<code>pi</code>	3.141592653589793
<code>ceil(x)</code>	Наименьшее целое число, превосходящее $x$
<code>floor(x)</code>	Наибольшее целое число, не превосходящее $x$
<code>sign(x)</code>	Знак числа <sup>6</sup> $x$ (1 для $x > 0$ , -1 для $x < 0$ , 0 для $x = 0$ )
<code>factorial(n)</code>	$n!$ , аргумент – целое неотрицательное число
<code>random()</code>	Возвращает (псевдо) случайное число в диапазоне $[0, 1)$
<code>combin(n, k)</code>	Биномиальный коэффициент, $C_n^k = \frac{n!}{k!(n-k)!}$
<code>radians(x)</code>	$\pi x / 180$ (перевод градусов в радианы)
<code>degrees(x)</code>	$180x / \pi$ (перевод радианов в градусы)

Для пользователей GlowScript версии VPython предоставляет возможность подключения модуля `random` с ограниченным функционалом: в нем доступны функции `seed_state()`, `get_random_byte()`, `seed()`, `random()`, `randrange()`, `randint()`, `uniform()`, `choice()`, `shuffle()` и `sample()`. Для вызова упомянутой в Таблице 1 функции `random()` подключение модуля `random` не требуется.

В этом разделе следует упомянуть также функции, которые хотя и не являются в строгом смысле математическими, но так же, как и функции из Таблицы 1, служат для замены стандартной библиотеки Питон. На этот раз речь идет о модуле `time`.

---

<sup>6</sup> Несмотря на то, что функция `sign` вычисления знака числа давно упоминается в документации пакета как реализованная, на самом деле, на момент написания этого пособия (VPython версии 7.6.1), эта функция поддерживается только в GlowScript версии.

Функция `sleep(t)` останавливает выполнение программы на  $t$  секунд, а функция `clock()` возвращает количество секунд, прошедших до вызова этой функции. Строго говоря, с какого момента функция `clock()` начинает свой отсчет, в стандартном Питоне зависело от операционной системы, поэтому, видимо, в версии языка 3.8 она была удалена. Тем не менее, при отсутствии доступа к библиотеке `time` эта функция позволяет в VPython измерять интервалы времени. В GlowScript реализации доступна также функция `msclock()`, которая возвращает время не в секундах, а в миллисекундах.

**Операции с векторами.** Класс `vector` служит для определения векторов, которые являются базовыми в пакете VPython для создания 3D-объектов и операций с ними. Сами по себе объекты этого класса не являются отображаемыми, но их использование позволяет эффективно решать различные задачи вычислительной геометрии.

Конструктор `vector`, или его синоним `vec`, используется для создания векторов достаточно естественным образом. Оператор

$$v = \text{vector}(x, y, z)$$

создает вектор с координатами  $x$ ,  $y$  и  $z$ . Доступ к отдельным компонентам вектора (на чтение и на запись) обеспечивают свойства `v.x`, `v.y`, `v.z`. Для экземпляров класса `vector` определены операции сложения, вычитания и умножения на числа, имеющие смысл обычных математических операций с векторами.

Ниже приведен полный список функций<sup>7</sup> VPython для работы с векторами.

`mag(a)`, или `a.mag` – длина вектора,  $|\vec{a}|$ .

`mag2(a)`, или `a.mag2` – квадрат длины вектора,  $\vec{a} \cdot \vec{a}$ .

`norm(a)`, или `a.norm()` – единичный вектор, сонаправленный с заданным,  $|\vec{a}|^{-1} \vec{a}$ .

`hat(a)`, или `a.hat` – альтернативный способ вычисления  $|\vec{a}|^{-1} \vec{a}$ . Слово *hat* связано с часто используемой в физике «шапочкой»  $\hat{\vec{a}}$  для обозначения такого вектора.

---

<sup>7</sup> Часть этих функций реализована также в виде атрибутов и методов класса `vector`



Для удобства по определению полагается, что `norm(vec(0,0,0))`, или `vec(0,0,0).hat` представляют собой нулевой вектор: `vec(0,0,0)`.

`dot(a,b)`, или `a.dot(b)`, или `a dot b` – скалярное произведение двух векторов.

`cross(a,b)`, или `a.cross(b)` – векторное произведение двух векторов.

`diff_angle(a,b)`, или `a.diff_angle(a)` – угол между двумя векторами, в радианах.

`proj(a,b)`, или `a.proj(b)` –  $(\vec{a} \cdot \hat{b}) \hat{b}$ , векторная проекция вектора  $\vec{a}$  на направление вектора  $\vec{b}$ .

`comp(a,b)`, или `a.comp(b)` –  $\vec{a} \cdot \hat{b}$ , скалярная проекция вектора  $\vec{a}$  на направление вектора  $\vec{b}$ .

`a.equals(b)` принимает значение **True**, если векторы  $\vec{a}$  и  $\vec{b}$  имеют одинаковые компоненты, т.е. их длины и направления совпадают.

`vector.random()` генерирует вектор, каждая компонента которого является случайным числом из интервала (-1, +1).

Характеристики `mag`, `mag2` и `hat` являются, на самом деле т. н. вычисляемыми свойствами объекта класса `vector`. Присваивание нового значения параметрам `mag` и `mag2` меняет все компоненты вектора, так чтобы его длина изменилась в соответствии с присвоенным значением, а направление осталось прежним. Например, после выполнения последовательности команд `x = norm(y); y.mag = 1` значения компонент векторов  $\vec{x}$  и  $\vec{y}$  станут одинаковыми, и результатом вызова функции `x.equals(y)` будет **True**. Присваивание вида `x.hat = y` приведет к тому, что длина вектора  $\vec{x}$  не изменится, а его направление будет совпадать с направлением вектора  $\vec{y}$ .

**Построение графиков.** Пакет `VPython` содержит набор средств для построения графиков различного вида, в том числе в логарифмическом масштабе. Построенные графики являются интерактивными, в том смысле, что перемещение курсора мыши в области построения графика сопровождается выводом различной вспомогательной информации.

Для отображения графиков служит класс `graph`, позволяющий создавать и отображать двумерные объекты – точки, кривые, вертикальные или горизонтальные столбики (см. рис. 4).

Для построения графиков используется один из двух графических пакетов. Первый, основанный на `Flot` [1.4], считается более быстрым, зато второй, основанный на `Plotly` [1.8], предлагает более широкий набор интерактивных возможностей, таких как масштабирование и прокрутку изображения. По умолчанию используется первый вариант, соответствующий значению параметра `fast=True` в списке аргументов функций `graph`, `gcurve`, `gdots`, `gvbars`, или `ghbars`. Во многих случаях «медленная» версия не уступает в скорости; отличие становится заметным лишь при построении графиков на основе очень большого количества данных.

Изучить возможности построения графиков лучше на примерах. На рис. 4 на отрезке  $[-30, 75]$  с шагом 1 по параметру  $t$  построены графики следующих функций:

$$f_1(t) = 5 + 5 \cos(-0.2t)e^{0.015t} \text{ (сплошная линия с маркерами)}$$

$$f_2(t) = 2 + 5 \cos(-0.1t)e^{0.015t} \text{ (столбики)}$$

$$f_3(t) = 5 \cos(-0.03t)e^{0.015t} \text{ (точечная линия)}$$

Код для их построения выглядит, соответственно, так:

Фрагмент кода 3

```
1 f1 = gcurve(width=4, markers=True,
2           marker_color=color.orange,
3           label='сплошная')
4 f2 = gvbars(delta=0.4, color=color.green,
5           label='столбики')
6 f3 = gdots(color=color.red, size=6,
7           label='пунктир')
8 for t in range(-30, 76):
9     f1.plot(t, 5.0+5.0*cos(-0.2*t)*exp(0.015*t))
10    f2.plot(t, 2.0+5.0*cos(-0.1*t)*exp(0.015*t))
11    f3.plot(t, 5.0*cos(-0.03*t)*exp(0.015*t))
```

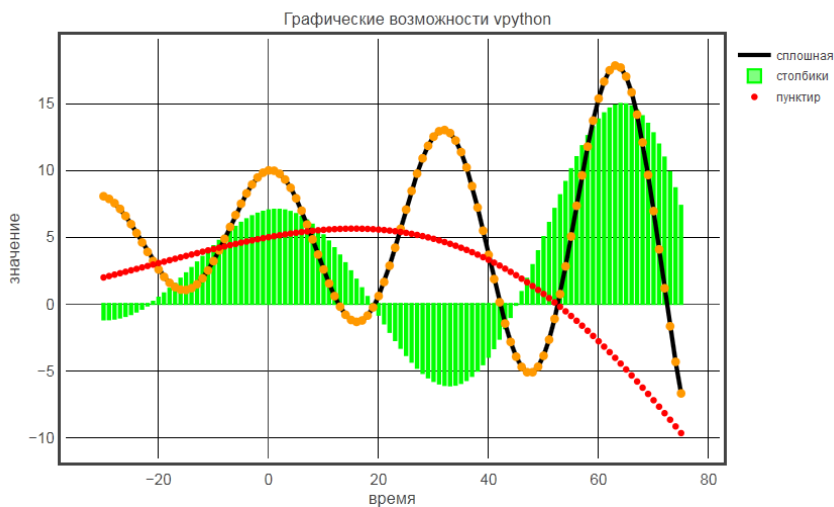


Рис. 4. Пример графика

В примере приведены три из четырех возможных типов объектов построения графика – сплошная кривая (`gcurve`), пунктир, т. е. линия, составленная из отдельных точек (`gdots`), и вертикальные столбики (`gvbars`). Четвертый тип – это горизонтальные прямоугольники (`ghbars`). Как видно из примера и результатов его выполнения, на одном чертеже могут располагаться сразу несколько графиков.

Каждый из этих объектов для построения графика использует список точек – пар «абсцисса-ордината». Этот список может быть задан параметром `data` при задании графического объекта, например

```
f1 = gcurve(data=[ [1,2], [2,-2], [3,1], [4,3] ])
```

а также может пополняться в процессе работы программы, любым из вариантов:

```
f1.plot(1,2)
```

```
f1.plot([1,2])
```

```
f1.plot([1,2], [3,4], [5,6])
```

```
f1.plot([ [1,2], [3,4], [5,6] ])
```

Обратите внимание, что имя параметра `data` можно опускать. В процессе работы можно полностью поменять все точки графика присваиванием нового значения атрибуту `data`, например так

```
f1.data = [ [10,20], [30,40], [50,60] ]
```

Цвет графического объекта может быть выбран в момент его создания (по умолчанию он черный); присваивание нового значения атрибуту цвета в ходе работы приведет к изменению не только новых точек, но и всех уже отображенных.

Для объекта `gcurve` можно указать ширину линии в пикселах. Значение параметра `markers=True` означает, что для каждой точки данных<sup>8</sup> графика на экране в соответствующей позиции будет изображен круг. По умолчанию он слегка толще ширины кривой; его радиус или диаметр может быть при необходимости задан с помощью параметров `radius` или `size`, соответственно. Параметр `marker_color`, как и следует из его названия, отвечает за цвет маркера, по умолчанию совпадающего с цветом кривой.

Задав параметр `dot=True`, можно сделать текущую точку графика отображаемой в виде кружка. Это оказывается полезным, например, если график представляет собой отрисовку траектории движения некоторого объекта. Параметр `dot_radius` задает размер этого кружка (радиус по умолчанию равен трем). Цвет кружка по умолчанию совпадает с цветом объекта `gcurve`, но при необходимости можно задать другой цвет, используя атрибут `dot_color`.

Для кружков пунктирной линии `gdots` также можно задать атрибуты `radius` или `size`; по умолчанию радиус равен трем пикселах.

Для объектов `gvbars` и `ghbars` могут быть задан атрибут `delta`, означающий ширину столбика; значение этой характеристики по умолчанию равно единице.

В некоторых случаях уменьшение количества точек на графике не приводит к заметному ухудшению его качества, но может существенно

---

<sup>8</sup> Речь идет только о тех точках, которые входят в набор `data`. При построении объекта `gcurve` эти точки соединяются сплошной линией, поэтому формально на графике изображено больше точек (см. рис. 4)

ускорить построение фигуры. Для уменьшения количества отображаемых точек при задании всех четырех перечисленных выше объектов используется параметр `interval`. Если он равен, например, десяти, то точка реально добавляется в список точек графика лишь при каждом десятом вызове операции добавления. Если интервал равен нулю, добавление точек к графику прекращается. Если интервал равен -1 (значение по умолчанию), то точки не пропускаются.

Перед тем как создавать объекты отображения графиков можно предварительно создать объект `graph`, для того чтобы определить размеры области отображения и ее расположение, заголовок графического окна, подписи координатных осей и диапазон изменения этих осей, фоновый цвет и т. п. Большинство из этих характеристик присутствует и на рис. 4, но мы рассмотрим новый фрагмент кода, результат работы которого представлен на рис 5.

Фрагмент кода 4

```
1 import numpy as np
2 gd = graph(width=400, height=400, fast=False,
3           title='<b>Кубическая<br>парабола</b>',
4           xtitle='<i>x</i>',
5           ytitle='<i>x</i><sup>3</sup>',
6           foreground=color.black,
7           background=color.white,
8           xmin=-5, xmax=5,
9           ymin=-100, ymax=100)
10 x3 = gcurve(width=3)
11 x = np.linspace(gd.xmin, gd.xmax, 200)
12 y = x**3
13 x3.data = zip(x,y)
```

Фрагмент кода 4 создает графическое окно размером 400 на 400 пикселей, над которым по центру размещается заголовок «Кубическая парабола». Оси абсцисс и ординат графика имеют метки  $x$  и  $x^3$ , соответственно. Цвет фона – белый, а цвет рисования осей и линий сетки – черный (что совпадает со значениями, установленными по умолчанию).

В отличие от принятого по умолчанию режима автоматического масштабирования области отображения, т. е. установки пределов изменения аргумента и значений функции так, чтобы в окне помещался

весь график целиком, здесь использованы фиксированные границы и для аргумента, и для значений функции. Если указать только одно из значений в паре ( $x_{\min}$ ,  $x_{\max}$ ) или ( $y_{\min}$ ,  $y_{\max}$ ), то второму автоматически присвоится ноль. При использовании графического движка на основе библиотеки Plotly, что соответствует установке параметра `fast` в **False**, рекомендуется указывать оба значения в паре.

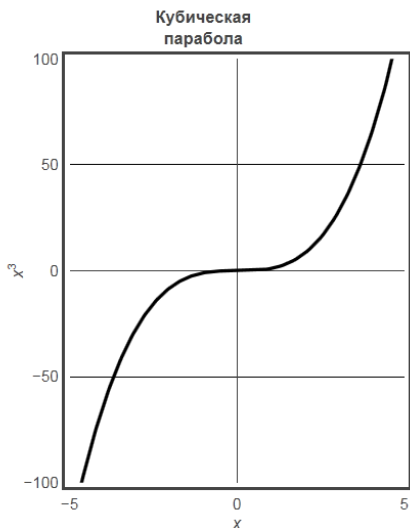


Рис. 5. Использование параметров настройки области отображения графика

В качестве параметров `title`, `xtitle`, и `ytitle` могут быть использованы любые строковые, а также числовые выражения, которые будут преобразованы в строковый формат. Эти параметры поддерживают теги HTML для наклонного (`<i>` или `<em>`) и полужирного (`<b>` или `<strong>`) начертания символов и для отображения верхних (`<sup>`) и нижних (`<sub>`) индексов. Параметр `title` поддерживает также многострочность на основе тега `<br>` или `<br/>`.

Вызов функции `graph()` без параметров создает область рисования с автоматическим масштабированием размера 640x400, без заголовка и подписей.

В окне браузера может быть создано несколько областей отображения графиков. Их взаимное расположение частично регулируется

параметром выравнивания: `align="left"` означает, что область графиков будет придвинута к левой границе окна браузера, `align="right"` – к правой. По умолчанию `align="none"`, объекты размещаются в окне браузера последовательно, как любые элементы HTML документа.

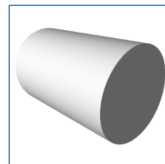
По умолчанию, очередной графический объект принадлежит той области рисования, которая создана последней. Для изменения этого порядка при создании любого из четырех графических объектов можно явно указать требуемую область, например, так:

```
energy = gdots(graph=graph2, color=color.blue)
```

Для создания графиков в логарифмическом масштабе используются атрибуты `logx=True` и/или `logy=True`. Необходимо помнить, что при этом все значения соответствующей координаты должны быть неотрицательными.

## БАЗОВЫЕ 3D-ОБЪЕКТЫ

**Цилиндр.** На примере объекта «Цилиндр» будут продемонстрированы основные черты, присущие основным 3D-объектам библиотеки VPython, таким как параллелограмм, сфера, пирамида и т. д.



Для создания (и отображения на сцене) цилиндра используется функция `cylinder()`. В примере ниже она служит для создания цилиндра, точка привязки которого – центр нижнего основания – лежит в плоскости  $yz$  и имеет там координаты  $(2,1)$ , ось параллельна оси  $Ox$ , радиус основания равен единице, а высота равна пяти.

```
rod = cylinder(pos=vector(0,2,1),  
              axis=vector(5,0,0), radius=1)
```

Высота созданного таким образом цилиндра совпадает с длиной вектора `axis`.

Цилиндр, созданный выше, связан с переменной `rod`. С ее использованием положение цилиндра может быть изменено в любой момент после его создания, что сразу же приводит к отображению его в новой позиции. Для параллельного переноса достаточно изменить положение точки привязки – как вектора целиком

```
rod.pos = vector(3,4,5)
```

так и одной из его координат

```
rod.pos.x = 10
```

Вращение цилиндра, как и любого другого объекта, осуществляется с помощью функции `rotate()`:

```
rod.rotate(angle=a,  
           axis=vec(x,y,z),  
           origin=vector(x0,y0,z0))
```

Параметры этой функции имеют следующий смысл:

`angle` – угол поворота в радианах,

`axis` – вектор, характеризующий направление оси вращения,

`origin` – точка, лежащая на этой оси.

При создании трехмерных объектов (не только цилиндров) может быть использован целый ряд ключевых (или именованных, т. е. не позиционных, а используемых в формате `имя=значение`) аргументов из следующего набора:

`pos`, точка привязки объекта. Для цилиндра это центральная точка одного из торцов. Для стрелки, конуса и пирамиды ситуация аналогична. А вот для параллелепипеда, сферы и кольца эта точка соответствует геометрическому центру объекта. Значением точки привязки по умолчанию является `vector(0,0,0)`.

`axis`, ось – вектор, направленный из точки привязки к другой «стороне» объекта; в случае цилиндра его длина равна, поэтому, высоте. Значением оси по умолчанию является вектор единичной длины `vector(1,0,0)`. Изменяя значение этого вектора, можно откорректировать размер объекта, в том числе динамически.

`up`, вектор, определяющий «верхнюю» сторону 3D-объекта. С точки зрения отображения эффект данного параметра может быть заметен, например, при использовании текстур определенного вида, или когда сечение цилиндра представляет собой не круг, а овал. Значением по умолчанию является вектор `vector(0,1,0)`. Векторы `axis` и `up` всегда перпендикулярны друг другу, изменение направления одного из них влечет за собой автоматическую корректировку другого.



`length`, длина вектора `axis` и, одновременно,  $x$ -компонента вектора `size`; по умолчанию равна единице. Изменение этого параметра влечет за собой корректировку длины вектора `axis`.

`radius`, радиус объекта, по умолчанию равен единице.

`size`, вектор вида `vector(length, height, width)`, который может быть использован для задания размера объекта вместо параметров `length` и `radius`. Его использование как раз и означает, в частности, что поперечное сечение цилиндра может быть эллиптическим. Задание значения параметру `size` приводит к изменению длины вектора `axis`, которая становится равным его  $x$ -компоненте (`length`).

`color`, вектор цвета объекта в формате RGB. Значением по умолчанию является `vector(1, 1, 1)`, что соответствует белому цвету

`red`, `green`, `blue`, вещественные цветовые параметры, позволяющие задать значение каждой составляющей цвета индивидуально, в пределах от нуля до единицы; дефолтные их значения, естественно, равны единице.

`opacity`, вещественный параметр прозрачности, принимает значения от нуля (полностью прозрачный) до единицы (полностью непрозрачный, значение по умолчанию).

`visible`, параметр отображения, установка значения которого в **False** приводит к тому, что объект перестает отображаться, например, `rod.visible = False`. Присвоение атрибуту `visible` значения **True** снова делает объект видимым.

`shininess`, вещественный параметр блеска, или коэффициент зеркального отражения, меняется в пределах от 0 до 1, и имеет значение по умолчанию, равное 0.6.

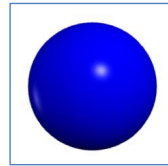
`emissive`, логический параметр, показывающий, является ли объект излучающим свет. Если он установлен в **True**, то яркость объекта определяется исключительно его собственным цветом, остальные источники света при ее расчете игнорируются. Примером использования этого параметра может служить излучающая сфера, помещенная в то же место, что и источник локального света; результат будет напоминать светящуюся лампу. В данном пособии эффект излучения использован для

визуализации Солнца в планетарной модели (см. главу 5). По умолчанию излучение отключено (**False**).

`texture`, позволяет задать текстуру поверхности объекта с использованием либо встроенной библиотеки изображений, либо графических файлов пользователя. Более подробно использование этого параметра и возникающие при этом трудности описаны ниже в разделе «Текстуры».

`make_trail`, логический параметр, установка которого в **True** позволяет создать видимый след за движущимся объектом типа стрелка, параллелепипед, конус, цилиндр, эллипсоид, кольцо, пирамида или сфера. Настройки отображения этого следа описаны ниже в разделе «Дополнительные атрибуты и методы».

**Сфера/Эллипсоид.** Для создания сферы используется функция `sphere()`. Например, для создания сферы синего цвета, имеющей радиус 0.5, центр которой расположен в точке с координатами (1, 2, -1) эта функция вызывается со следующим набором параметров:



```
ball = sphere(pos=vector(1, 2, -1), radius=0.5,  
              color=color.blue)
```

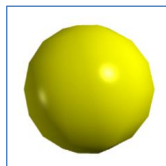
Заметим, что точкой привязки сферы, в отличие от цилиндра, является ее геометрический центр.

Все возможные параметры объекта *сфера* аналогичны перечисленным при описании объекта *цилиндр*, однако имеются незначительные отличия.

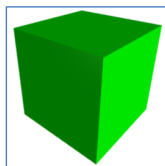
Длина вектора `axis` для сферы не имеет никакого значения, ее размер определяется параметром `radius`, или параметром `length` вектора `size=vector(length, height, width)`, который имеет смысл диаметра сферы. Дефолтное значение вектора `size` равно (2,2,2), т. е. по умолчанию сфера является единичной. Как и в случае цилиндра, делая значения `length`, `height` и `width` неодинаковыми, можно вместо сферы получить эллипсоид. В отличие от других 3D объектов изменение вектора `size` не влияет на вектор `axis` и наоборот.

Хотя эллипсоид можно создать на основе объекта сфера, в VPython существует и специальный класс для создания эллипсоидов. Объекты этого класса создаются с помощью функции `ellipsoid`. В отличие от сферы параметр `axis` играет важную роль, прежде всего, отвечая за ориентацию эллипсоида в пространстве. Кроме того, если параметр `length` эллипсоида не задан, он принимается равным длине вектора его оси.

**«Простая» сфера.** Для того, чтобы изображение сферы на экране выглядело гладким даже при большом увеличении, этот объект состоит из очень большого количества треугольников. В тех случаях, когда сфер на экране должно быть много, а качество их прорисовки не так существенно, разработчики VPython рекомендуют использовать объект `simple_sphere`. Он имеет те же самые параметры, отличие становится заметным лишь при рендеринге – его поверхность состоит из небольшого количества треугольников. Именно простые сферы используются при отображении следа движущихся тел. Данный объект рекомендуется использовать для визуализации системы частиц, положения которых рассчитываются методами молекулярной динамики (см. главу 6).



**Параллелепипед.** Для создания прямоугольного параллелепипеда в VPython используется функция `box()`. В отличие от цилиндра точкой привязки прямоугольника служит его центр. Размеры могут быть заданы указанием характеристик `length` (длина, или размер вдоль оси  $x$ ), `height` (высота, или размер вдоль оси  $y$ ) и `width` (ширина, или размер вдоль оси  $z$ ), например так:



```
b = box(length=10, height=1, width=2)
```

Этого же результата можно достигнуть с использованием параметра `size`:

```
b = box(size=vector(10, 1, 2))
```

Осью созданного таким образом прямоугольника будет вектор  $(10, 0, 0)$ : хотя точка привязки прямоугольника не такая как у цилиндра, вектор `axis` аналогичен случаю цилиндра и соединяет одно «основание» с другим.

Вектор `axis` может быть использован при создании параллелепипеда, грани которого не параллельны координатным плоскостям. При этом нужно учесть следующие обстоятельства: длиной параллелепипеда теперь называется его размер в направлении этого вектора (см. рис. 6); если параметр `length` или вектор `size` не заданы, то эта длина совпадает с длиной вектора `axis`. Вращать прямоугольник вокруг его оси можно с помощью изменения вектора `up` (который по умолчанию, как и для цилиндра, есть вектор  $(0,1,0)$ ). С учетом сказанного конструкция вида

```
mybox = box(axis=vector(a_x, a_y, a_z),  
            length=L, height=H, width=W,  
            up=vector(u_x, u_y, u_z))
```

приведет к созданию параллелепипеда, высоты  $H$ , основание которого с размерами  $L \times W$  будет перпендикулярно вектору  $(u_x, u_y, u_z)$ . Ясно, что поскольку все упомянутые параметры параллелепипеда не являются независимыми, то при создании такого объекта с их одновременным использованием важно учитывать порядок применения этих параметров. А он такой: сначала задается `axis` (тем самым определяя  $x$ -составляющую вектора `size`), затем задается вектор `size` (возможно, корректирующий длину «оси»), и после этого задается вектор `up` (что может привести к еще одной корректировке оси, чтобы ее направление было перпендикулярно этому вектору).

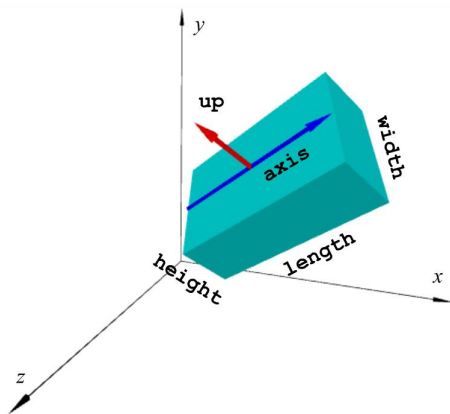
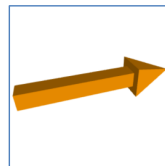


Рис. 6. Параметры объекта «Параллелепипед»

**Стрелка.** Объект «стрелка» представляет собой «древко» в форме вытянутого вдоль одной оси параллелепипеда с квадратным поперечным сечением и «наконечника», имеющего форму правильной четырехугольной пирамиды. Для ее создания используется функция `arrow()`.



Базовые атрибуты данного объекта и их дефолтные значения совпадают с аналогичными характеристиками цилиндра: `pos` (0,0,0), `axis` (1,0,0), `length` (1), `color` (1,1,1) (белый цвет), `opacity`(1), `shininess` (0.6), `emissive` (**False**), и `up` (0,1,0). Параметр `up` является существенным, потому что и древко, и наконечник имеют квадратные поперечные сечения, поэтому изменение вектора `up` позволяет вращать стрелку вокруг ее оси. Важно отметить также, что атрибут `pos` для стрелки соответствует точке, находящейся в центре основания древка.

Специфическими атрибутами объекта «стрелка» являются:

- `shaftwidth` (ширина древка),  
по умолчанию `shaftwidth = 0.1 * (длина стрелки)`;
- `headwidth` (ширина наконечника),  
по умолчанию `headwidth = 2 * shaftwidth`;
- `headlength` (длина наконечника),  
по умолчанию `headlength = 3 * shaftwidth`.

Присваивание `shaftwidth = 0` устанавливает значение по умолчанию (одна десятая от длины стрелки). Предельно допустимая величина `headlength` составляет половину длины стрелки; при превышении этого размера вся стрелка масштабируется, так чтобы устранить это превышение. Аналогичное масштабирование происходит в случае, когда древко становится тоньше, чем 1/50 от длины.

Такое поведение приводит к уменьшению ширины очень коротких стрелок и увеличению ширины очень длинных стрелок (при отображении правильной общей длины). Если требуется, чтобы размеры поперечных сечений древка и наконечника не менялись при изменении длины, следует задать требуемое значение для параметра `shaftwidth`. В этом случае единственная разрешенная автоматическая корректировка заключается в

настройке длины наконечника так, чтобы она никогда не превышала половины общей длины стрелки.

В качестве примера использования стрелок приведем фрагменты кода построения фигур на рис. 6. Стрелка, изображающая ось  $Ox$  задается соотношением

```
x1 = arrow(pos=vector(0,0,0), axis=vector(10,0,0),
           color=color.black,
           shaftwidth=0.02, headwidth=0.12)
```

Стрелка, изображающая вектор  $up$ , привязана к положению и ориентации параллелепипеда `mybox`:

```
up_arrow = arrow(pos = mybox.pos, axis=2*mybox.up,
                 shaftwidth=0.15, headlength=0.8,
                 color=color.red)
```

**Присоединение стрелки.** Функция `attach_arrow()` (присоединить стрелку) позволяет создать объект «стрелка», привязанный к тому объекту, чье движение требуется проиллюстрировать. Например, таким образом можно продемонстрировать в некотором масштабе вектор скорости объекта или изобразить действующую на объект силу. Основная особенность присоединенной к объекту стрелки состоит в том, что она динамически меняется в соответствии с одной из характеристик объекта, к которому она привязана. Именно объект и его характеристика (атрибут) должны быть указаны в конструкторе создания привязанной стрелки, например, так:

```
arr = attach_arrow(ball, "axis", scale=100)
```

В данном примере стрелка будет привязана к объекту `ball`, причем точка привязки стрелки (центр основания древка) будет совпадать с точкой привязки этого объекта (центром сферы). Направление стрелки будет совпадать с направлением оси (`axis`) объекта, а длина стрелки получится умножением параметра `scale` на длину вектора `ball.axis`.

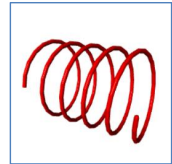
По умолчанию значение параметра `scale` равно единице 1, цвет создаваемой стрелки совпадает с цветом того объекта, к которому она привязана, а значение параметра `shaftwidth` определяется формулой `0.5*ball.size.y`.

Параметр, иллюстрируемый стрелкой, может быть любым векторным атрибутом объекта. Например, если в процессе моделирования положение объекта `ball` постоянно меняется на основе информации о векторе скорости, который добавлен в список атрибутов (`ball.velocity`) то для постоянного отображения скорости в той точке, в которой расположен объект `ball`, достаточно написать:

```
vel = attach_arrow(ball, "velocity")
```

С помощью функций `vel.stop()` и `vel.start()` можно приостанавливать и снова запускать отображение присоединенной стрелки.

**Спираль.** Для создания спирали – объекта, часто используемого в задачах визуализации механических систем для изображения пружинок – используется конструктор `helix`. Базовые атрибуты спирали – `pos`, `axis`, `length`, `color` и `up` – и их дефолтные значения совпадают со случаем цилиндра. Как и у цилиндра, точка привязки, `pos`, находится на краю спирали.



Специфическими для данного объекта являются следующие атрибуты:

`radius`, представляет собой наружный радиус<sup>9</sup> витка спирали, т.е. внешний радиус кольца – проекции спирали на плоскость, перпендикулярную ее оси; значение по умолчанию равно единице.

`thickness`, совпадает с толщиной (точнее, диаметром) поперечного сечения кривой, которая изогнута в спираль; у пружин такая характеристика называется диаметром проволоки; значение по умолчанию равно одной двадцатой части радиуса спирали, `radius/20`.

`coils`, количество витков спирали; по умолчанию равно пяти.

**Пирамида и конус.** Эти объекты создаются конструкторами `pyramid()` и `cone()` соответственно. Основные параметры фигур похожи; точка привязки `pos` лежит в центре основания: для пирамиды это

---

<sup>9</sup> Как и в случае цилиндра, вместо задания параметра длины спирали, `length`, можно устанавливать ее размер в формате `size=vector(length, height, width)`, т.е. допустимы спирали с эллиптическим поперечным сечением

прямоугольник, для конуса – круг (в общем случае – эллипс). Ось фигуры, вектор `axis`, изображенный темной стрелкой на рис. 7, соединяет середину основания фигуры с ее вершиной

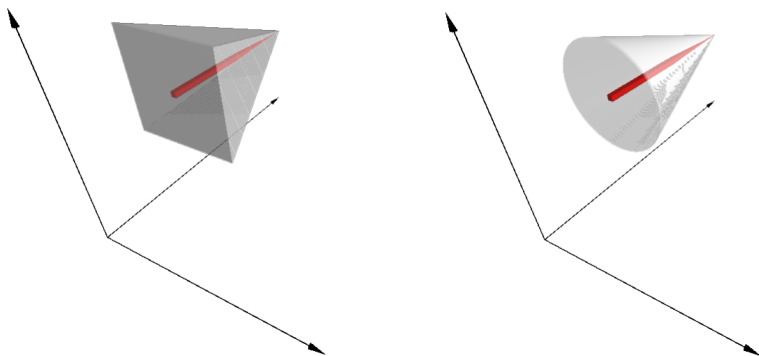


Рис. 7. Пирамида и конус. Темная стрелка соответствует вектору `axis`.

За размеры кругового основания у конуса отвечает параметр `radius`, у пирамиды – `height` (в направлении оси  $Oy$ ) и `width`. По умолчанию все эти размеры равны единице; вектор `axis` направлен вдоль оси  $Ox$ . Для обеих фигур размеры можно задать с помощью вектора `size(length, height, width)`, независимо определив высоту и размеры основания.

**Кольцо.** У этого объекта, имеющего по умолчанию форму кругового кольца и создаваемого функцией `ring()`, есть две специфические характеристики – радиус (`radius`) и толщина (`thickness`). Радиусом кольца является радиус срединной окружности, т. е. окружности, проходящей через центры его поперечных сечений. По-другому этот радиус можно определить как среднее арифметическое внешнего и внутреннего радиусов кольца. Толщина кольца представляет собой разность этих радиусов. По умолчанию, значение толщины равно одной десятой части радиуса кольца.



Ось кольца, атрибут `axis`, определяет его ориентацию в пространстве, длина этого вектора не имеет значения. Точка привязки, атрибут `pos`, расположена в геометрическом центре кольца.



Кольцу можно придать овальную форму, если вместо радиуса и толщины задать вектор `size=vector(length,height,width)`, у которого параметры `height` и `width` не равны друг другу.

**Пространственная ломаная.** Объект представляет собой набор отрезков, соединяющих заданные точки. Если точки расположены достаточно близко, то визуально ломаная может выглядеть как гладкая кривая. Это объясняет имя данного объекта – `curve` (кривая).



Основным входным параметром при создании ломаной является список точек. В простейшем случае точка задается своим радиус-вектором, то есть вектором координат. Если в программе заданы, например, положения точек `a`, `b` и `c`, то ломаную, выходящую из точки `a`, идущую в точку `b`, а затем в точку `c`, можно создать командой

```
line1 = curve(a, b, c)
```

или, эквивалентно

```
line1 = curve([a, b, c])
```

Иллюстрация к данному подразделу создана с помощью следующего фрагмента кода:

```
t = [5+0.1*i for i in range(1, 201)]
arc = [vector(sin(x)/x, cos(x)/x, x/100) for x in t]
sp = curve(arc, radius=0.01, color=color.green)
```

Из приведенного примера видно, в частности, что кривая имеет атрибуты цвета (по умолчанию – белый) и радиус (по умолчанию – 0, что соответствует нескольким пикселям на экране).

Заметим, что каждая точка ломаной может иметь свой цвет и свой радиус, так что, вообще говоря, точка – это не просто вектор положения: для задания отдельной точки можно использовать словарь вида

```
p1 = {'pos':v1, 'color':color.red,
      'radius':0.1, visible=True}
```

В то же время, как отдельные точки ломаной, так и вся она целиком, не имеют атрибутов прозрачности и наложенной текстуры.

Для добавления точек к ломаной служат два метода: `append` и `unshift`. Первый добавляет точку или список точек в конец уже построенной ломаной, второй – в ее начало. Метод `splice(start, howmany, points)` сначала удаляет `howmany` точек из кривой, начиная с позиции `start`, а потом добавляет в это место новый список точек `points`.

Метод `pop(n)` возвращает информацию о точке кривой с номером `n` (в формате словаря) и удаляет эту точку. Метод `clear` стирает все точки кривой.

Метод `modify` позволяет изменить параметры отдельной точки, например, так:

```
c.modify(n, # номер изменяемой точки
        pos=p, color=c, radius=r, visible=v)
c.modify(n, x=3, y=5) # изменяются две координаты
c.modify(n, vector(x,y,z)) # изменяется положение
```

Поскольку каждая точка может иметь свой цвет, VPython использует интерполяцию (смешивание) цветов при переходе от одной точки к другой. Для резкой границы цветов необходимо добавить в ломаную две точки – с одинаковыми координатами, но разными цветами, как показано во фрагменте кода 5.

Фрагмент кода 5.

```
1 c = curve(radius=0.1)
2 #добавляем (белую по умолчанию) точку
3 c.append(vector(-1,0.5,-0.5))
4 #добавляем еще одну белую точку
5 c.append(vector(0,0,-0.5))
6 #добавляем черную точку в это же место
7 #c.append(pos=vector(0,0,-0.5), color=color.black)
8 #добавляем черную точку
9 c.append(pos=vector(1,0.5,-0.5), color=color.black)
```

На рисунке 8 показаны результаты двух вариантов выполнения этого кода. Изображение а) получено в результате непосредственного его выполнения, а случай б) соответствует выполнению кода с

раскомментированной седьмой строчкой. Добавление дополнительной точки меняет режим смешивания цветов на режим с резкой границей.

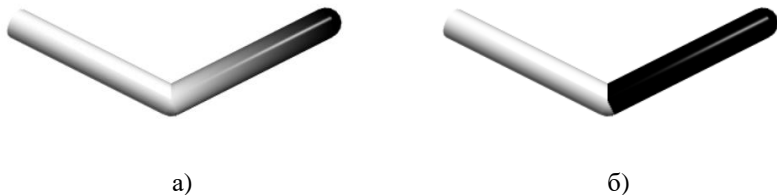
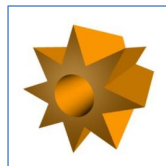


Рис. 8. Режимы смены цветов при построении объекта *curve*:  
а) интерполяция (смешивание), б) – резкая граница

**Экструзия.** Данный объект получается перемещением или «выдавливанием» двумерной формы вдоль некоторой заданной траектории. Термин *экструзия* пришел в компьютерную графику из мира реальных технологий, где он обозначает процесс продавливания формуемого вещества (например, вязкого расплава материала или густой смеси) через отверстие в профилирующем инструменте (экструзионной головке). В результате получается изделие с поперечным сечением нужной формы. Схема создания 3D-объекта аналогична: задается форма сечения и правило перемещения этого сечения (в простейшем случае – расстояние).



Для конструирования экструзии в VPython необходимы три шага:

- 1) Создание 2D-формы. В простейшем случае это может быть набор координат точек, например вершин треугольника:  
`shape2d = [[2,0], [0,4], [-2,0], [2,0]]`  
Форма должна быть замкнутой, поэтому последняя точка в примере совпадает с первой.  
Для более сложных форм VPython содержит библиотеку `shapes`, содержащую такие фигуры как многоугольник, круг, эллипс, звезда, шестеренка и т. п. Более подробно с этими фигурами и способами их описания можно ознакомиться в документации по пакету [1.5].
- 2) Задание двумерной или трехмерной траектории, вдоль которой будет перемещаться созданная на предыдущем этапе форма. Это

может быть ломаная, заданная списком точек, например, простой отрезок

```
trajectory = [vec(0,0,0), vec(0,0,1)]
```

или более сложная криволинейная траектория, построенная на основе кривых из библиотеки `shapes`.

3) Вызов функции создания экструзии, например, так

```
ex = extrusion(path=trajectory, shape=shape2d,  
              color=color.red)
```

2D-форма может содержать отверстия. Для этого в качестве параметра `shape` при создании экструзии можно использовать список кривых. Тогда та кривая в этом списке, которая охватывает остальные, будет считаться внешней границей формы, остальные – внутренними границами (границами вырезов). Так, например, звездочка, приведенная в качестве иллюстрации к данному подразделу, получена в результате выполнения следующего фрагмента кода

```
star = shapes.star(n=8, radius=3)  
circ = shapes.circle(radius=1)  
s = [vec(0,0,0), vec(10,0,0)]  
ex = extrusion(path=s, shape=[star, circ],  
              color=color.orange)
```

**Вывод текстовой информации.** Результат выполнения операции `print` зависит от используемой системы. В случае `GlowScript` вывод осуществляется в специальное текстовое окно под графическим изображением в браузере. При использовании `VPython 7` вывод осуществляется в традиционную среду `shell`.

Существуют два способа для добавления текста в окно с графическим изображением. Первый – использование объекта `text` – носит скорее демонстрационный характер, иллюстрируя возможности графического движка. Создаваемый экземпляр представляет собой трехмерную надпись, которая обладает всеми основными свойствами 3D объектов – положением, цветом, размером (точнее, параметрами `height`, `length` и `depth`) и т. д. На рис. 9 этот объект использован для создания текста «Планета и ее спутник». При перемещении камеры такой текст также может поворачиваться, приближаться/удаляться и т. п. К основным недостаткам объекта `text` относятся достаточно высокие вычислительные

затраты на его отображение и невозможность изменить текст надписи после создания объекта.

Второй способ предназначен для существенно более быстрого добавления текстовых комментариев или вывода той или иной числовой информации. Он основан на использовании объекта `label` (метка). Строго говоря, метка не является трехмерным объектом – она всегда обращена лицевой стороной к пользователю, а ряд ее характеристик задается в пикселах, а не в координатах трехмерной сцены.

На рис. 10 приведены основные параметры метки, отличающие ее от остальных объектов:

`pos`, вектор координат точки привязки метки; по умолчанию это т. н. «мировые» координаты, т. е. трехмерные координаты отображаемой сцены;



Рис. 9. Добавление текстовой информации в графическое окно VPython

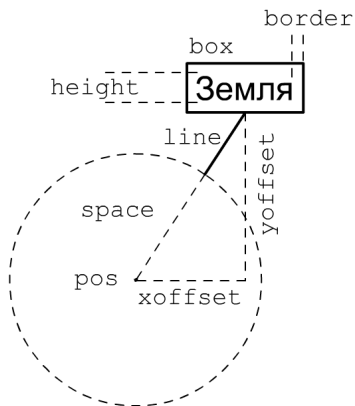


Рис. 10. Основные параметры объекта label

`pixel_pos`, логический атрибут, установка которого в **True** меняет смысл предыдущего параметра: компоненты вектора `pos` в этом случае соответствуют пикселям экрана, при этом точка с трехмерными координатами (0,0,0) располагается в левом нижнем углу графического окна;

`align`, параметр выравнивания текста метки (может принимать значения 'left', 'right', или 'center') относительно точки привязки;

`xoffset`, `yoffset` – смещение метки относительно точки привязки в пикселях;

`text`, текстовое содержимое метки. Может содержать HTML тэги для наклонного (<i> или <em>) и полужирного (<b> или <strong>) начертания символов, а также тэги верхнего и нижнего индексов (<sup> и <sub>). Кроме того, поддерживается тэг <br> или <br/> для разбивки строки. При задании параметру `text` числового значения, оно автоматически преобразуется к строковому типу;

`font`, имя шрифта, используемого для отображения текста метки. Стандартными являются 'sans' (по умолчанию), 'serif' и 'monospace';

`height`, высота шрифта в пикселях, по умолчанию – 15;

`color`, цвет текста, по умолчанию совпадает с цветом переднего плана холста, `scene.foreground`;

`background`, цвет заливки прямоугольника метки, по умолчанию совпадает с цветом фона холста, `scene.background`;

`opacity`, прозрачность заливки прямоугольника метки, по умолчанию – 0.66;

`border`, расстояние в пикселах от текста до ограничивающего его прямоугольника; по умолчанию – 5 пикселей;

`box`, логический параметр (по умолчанию **True**) – отображать или нет прямоугольную рамку вокруг текста;

`line`, логический параметр (по умолчанию **True**) – отображать или нет прямую линию, соединяющую метку с ее точкой привязки;

`linecolor`, цвет соединительной линии и прямоугольной рамки;

`linewidth`, толщина соединительной линии и рамки, по умолчанию – 1 пиксел;

`space`, радиус в пикселах мысленного круга вокруг точки привязки, внутри которого соединительная линия не отображается.

## ОБЩИЕ ХАРАКТЕРИСТИКИ ОБЪЕКТОВ

**Работа с цветом.** Для отображения цветов VPython использует цветовую модель RGB. Цвет задается вектором, компоненты которого, изменяющиеся от 0 до 1 включительно, представляют собой интенсивность красного, зеленого и синего оттенка, соответственно. При этом белому цвету соответствуют максимальные значения интенсивностей, т.е. `vector(1, 1, 1)`, а черному – минимальные, т.е. `vector(0, 0, 0)`. Класс `color` предлагает набор предопределенных констант для задания основных цветов (Таблица 1)

Таблица 1. Константы описания цветов

Значение цвета	Атрибут <code>color</code>	RGB вектор <sup>10</sup>
черный	<code>color.black</code>	<code>vec(0, 0, 0)</code>
белый	<code>color.white</code>	<code>vec(1, 1, 1)</code>
ярко-красный	<code>color.red1</code>	<code>vec(1, 0, 0)</code>
ярко-зеленый	<code>color.green</code>	<code>vec(0, 1, 0)</code>

<sup>10</sup> В пакете `vpthon` объект `vec` является синонимом объекта `vector`

ярко-синий	<code>color.blue</code>	<code>vec(0,0,1)</code>
желтый	<code>color.yellow</code>	<code>vec(1,1,0)</code>
оранжевый	<code>color.orange</code>	<code>vec(1,0.6,0)</code>
голубой	<code>color.cyan</code>	<code>vec(0,1,1)</code>
малиновый	<code>color.magenta</code>	<code>vec(1,0,1)</code>
сиреневый	<code>color.purple</code>	<code>vec(0.4,0.2,0.6)</code>

Кроме того, метод `gray()` этого класса позволяет задавать оттенки серого цвета в формате

```
color = color.gray(0.75)
```

что эквивалентно заданию цвета в виде

```
color = vector(0.75, 0.75, 0.75)
```

так что `color.gray(0)` соответствует черному цвету, а `color.gray(1)` – белому.

Класс `color` содержит две функции для конвертирования между цветовыми схемами RGB и HSV: `rgb_to_hsv(rgb_color)` и `hsv_to_rgb(hsv_color)`, аргументами и значениями которых являются векторы соответствующих схем. Функция `rgb_to_grayscale(rgb_color)` этого класса возвращает цветовой вектор того оттенка серого цвета, который максимально соответствуют цвету аргумента.

Разработчики пакета VPython регулярно напоминают, что на различных экранах и при разных условиях освещения 3D-сцены цвета могут заметно отличаться. Это в меньшей степени относится к предопределенным цветам, представленным в Таблице 1.

**Прозрачность.** Большинство объектов VPython имеют атрибут `opacity`, изменяющийся в пределах от 0 до 1. Значение 0 означает полную прозрачность, единице соответствует обычный непрозрачный объект. Как и все остальные атрибуты, прозрачность может меняться динамически, позволяя создавать эффекты вида «растворение объекта» и «проявление объекта»



В настоящее время свойство прозрачности отсутствует у объектов `curve` и `helix`.

**Текстуры.** Добавление текстуры – растрового изображения, накладываемого на поверхность объекта для придания ей цвета, окраски или иллюзии рельефа – осуществляется с помощью задания атрибута `texture` при создании объекта.

Пакет `VRython` содержит коллекцию из 12 изображений, приведенных на рис. 11, которые могут быть использованы следующим образом:

```
my_box = box(texture=textures.stucco)
```



Рис. 11. Коллекция встроенных текстур

Атрибут `texture` может задаваться также в виде словаря, если требуется применить текстуру не ко всей поверхности объекта, а только к некоторым его граням, например, к правой:

```
my_box = box(color=color.green,  
             texture={'file':textures.stucco,  
                    'place':'right'})
```

Другими возможными значениями для ключа 'place' служат 'left' (левая грань), 'sides' (все грани, кроме левой и правой), 'ends' (левая и правая грань вместе), а также 'all' (все грани – значение по умолчанию). Смысл «левой» и «правой» грани определяется тем, что по умолчанию осью объекта GlowScript является вектор (1,0,0), направленный вдоль оси  $x$ , и правой гранью считается та, что располагалась со стороны положительного направления этой оси в момент создания, до каких-либо возможных изменений ориентации объекта. В случае цилиндра 'sides' означает его боковую поверхность.

Значением ключа 'place' также может быть и список из перечисленных выше вариантов.

В случае использования VPython 7 в качестве текстур могут выступать графические файлы, хранящиеся в одной папке с программой или в подкаталоге этой папки. В этом случае значением атрибута texture должна служить строка с именем файла. Чтобы графический файл с текстурой был доступен всем python-программам на данном компьютере, его можно поместить в подкаталог

```
...../ Lib/site-packages/vpython/vpython_data
```

каталога, в который установлен Python. В этом каталоге находятся файлы стандартных текстур. При использовании IDLE текстуры из этой папки подключаются аналогично файлам, находящимся в одном каталоге с программой:

```
b = box(texture='T.jpg')
```

Теоретически возможным является и использование в качестве текстур «внешних» файлов, т. е. изображений из сети Интернет. Для этого в качестве значения атрибута texture нужно передать полный адрес изображения, например

```
site = "https://s3.amazonaws.com"  
folder = "/glowscript/textures/"  
file = "flower_texture.jpg"  
my_box.texture = site + folder + file
```

Существенным ограничением такого использования текстур является требование, чтобы файлы с ними располагались на т. н. "CORS enabled" веб-сайтах. Некоторые подробности о разрешении использования кросс-доменных изображений можно найти, например, в [1.3, 1.11].

**Дополнительные атрибуты и методы.** Ниже приведен ряд характеристик (атрибутов и методов), присущих всем 3D объектам VPython:

- `canvas`, холст, на котором будет создан 3D объект. В начале работы программы VPython создает холст с названием "scene". По умолчанию все создаваемые объекты попадают на этот холст. Атрибут `canvas` позволяет поместить создаваемый объект на другой холст:

```
scene23 = canvas(title="Действие II, Сцена 3")
rod = cylinder(canvas=scene23)
```

Метод `select()` холста делает его «выбранным»: после вызова `scene23.select()` все создаваемые объекты будут размещаться на нем.

Более подробно о свойствах холста, создаваемого пользователем, написано ниже в разделе «Холст: свойства и события».

- `frame`, фрейм, в котором помещается объект, например `ball = sphere(frame=f1)`. Данный атрибут не поддерживается в GlowScript.
- `visible` (видимость). По умолчанию значение этого логического атрибута равно **True**; если установить его в **False**, то объект не будет отображаться. Присваивание вида `ball.visible = True` снова сделает объект видимым.
- `rotate()` (вращение). С помощью этого метода объект  $B$  можно повернуть на угол  $\theta$  вокруг вектора  $\vec{a}$ , отложенного от т. н. точки приложения, имеющей радиус-вектор  $\vec{r}_0$ , следующим образом: `B.rotate(angle=theta, axis=a, origin=r0)`. По умолчанию ось вращения совпадает с осью объекта (`axis=B.axis`), а точка приложения совпадает с положением объекта (`origin=B.pos`).
- `bounding_box()` (ограничивающий параллелепипед). Данный метод, реализованный в GlowScript, но пока (во время написания данного пособия) не доступный VPython 7, позволяет найти координаты восьми вершин области, полностью ограничивающей заданный объект.

Результат метода `bounding_box()` представляет собой список трехмерных координат восьми угловых точек параллелепипеда, ограничивающего объект. При этом параметры `pos/axis/size` объекта могут быть любыми.

- `make_trail` (оставлять след). Данный атрибут позволяет изображать траекторию движения, указав, что за движущимся объектом – кольцом, конусом, параллелепипедом, пирамидой, стрелкой, сферой, цилиндром, эллипсоидом, а также составным объектом – будет оставаться след. Атрибут является логическим:

```
ball = sphere(make_trail=True)
```

Каждый раз при прорисовке сцены новая позиция объекта `ball` будет добавляться к создаваемой кривой, которая и будет представлять собой след движущегося объекта.

По умолчанию данный параметр установлен в **False**, т. е. след не изображается. Обычно его устанавливают в **True** при создании объекта, но оставление следа может быть включено (а при необходимости – выключено) в любой момент работы программы. Параметр след имеет свои дополнительные характеристики:

- `trail_type` По умолчанию данный параметр равен "curve" (сплошная кривая), но может быть изменен на "points" (набор точек) (см. рис. 12)

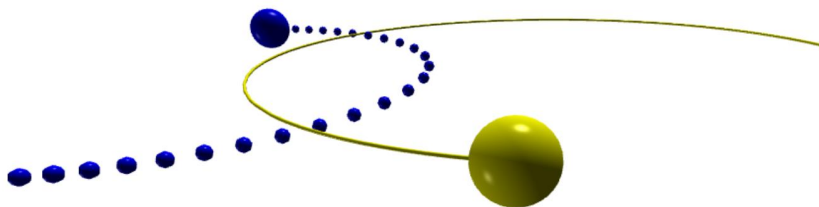


Рис. 12. След за движущимся объектом

- `interval` (интервал добавления точек в отображаемую траекторию). Значение параметра `interval`, равное, например, десяти, как у темного шарика на рис. 12, означает, что точка

добавляется в траекторию при каждом десятом передвижении объекта. Если этот параметр равен нулю, то траектория не отображается. Для следа в виде набора точек значение этого параметра по умолчанию равно единице; для следа в виде сплошной кривой добавление точки происходит при каждой новой перерисовке экрана, как правило, это от 30 до 60 раз в секунду. При необходимости откорректировать эту частоту без изменения параметра `interval`, можно использовать параметр `pps` (`points per second` – количество точек в секунду), значение которого будет приблизительно совпадать с частотой добавления точек в сплошную кривую.

- `retain` (количество оставляемых на экране точек траектории). Задание этого параметра позволяет убирать с экрана старые точки, так чтобы их общее число не превосходило заданного. В ряде случаев это позволяет избавиться от линий, загромождающих рисунок.
- `trail_color` (цвет следа). Данный параметр определяет цвет точек или кривой, отображающей траекторию. По умолчанию цвет следа совпадает с цветом самого объекта. Данный параметр позволяет динамически изменять цвет траектории.
- `trail_radius` (радиус следа) Определяет радиус кривой или радиус точки. Для кривой значение этого параметра равно нулю, что делает кривую достаточно тонкой. Для следа в виде точек их радиус по умолчанию равен 0.2 от значения параметра `radius` объекта, 0.1 от параметра `height` или `size.y` объекта в момент его создания, в зависимости от того, какие из этих параметров имеет объект. Как и остальные характеристики следа, его радиус может быть изменен динамически в процессе анимации движения.
- `clear_trail()` (стереть след). Данный метод стирает все точки текущей траектории, при этом добавление новых не прекращается.

## ХОЛСТ: СВОЙСТВА И СОБЫТИЯ

**Создание холста.** В начале работы VPython автоматически создает холст с названием `scene`. Этот холст не будет отображаться, пока на нем не потребуется разместить какой-то из 3D-объектов. Поэтому ни команда подключения модуля `vpython`, ни команда создания пользовательского холста в среде IDLE не приводят к открытию нового окна браузера. Однако создание любого из отображаемых объектов приведет к размещению его на сцене, а значит, сначала потребует открытия окна или вкладки в браузере и отображения холста.

Пользовательский холст с заданными атрибутами создается с помощью функции `canvas()`, которая возвращает объект с нужными свойствами и делает его *выбранным*. Все созданные после этого отображаемые объекты будут по умолчанию размещаться именно на этом холсте. В качестве примера приведенная ниже функция возвращает холст размера 600 x 200, с расположенным выше холста названием «Пример сцены», закрашенный голубым цветом. Центр холста расположен в точке (5, 0, 0), поэтому созданный на нем шар (имеющий по умолчанию положение в начале координат) будет выглядеть сдвинутым влево.

```
scene2 = canvas(title='Пример сцены',
                width=600, height=200,
                center=vector(5,0,0),
                background=color.cyan)
ball = sphere()
```

Еще раз отметим, что без вызова функции создания объекта `ball` холст не будет отображаться.

По умолчанию все вновь создаваемые объекты размещаются на самом новом из созданных холстов. Для отмены этого есть два способа. Во-первых, выбранным (т.е. предназначенным для размещения по умолчанию новых объектов) можно сделать любой из уже созданных холстов с помощью его метода `select()`. Чтобы узнать, какой холст является выбранным в данный момент, используется функция `canvas.get_selected()`, возвращающая ссылку на тот холст, в котором создаются объекты. Во-вторых, при создании объекта можно явно указывать тот холст, на котором его нужно разместить, например, `box(canvas=scene1, .....`).

## Основные характеристики холста.

`width, height` – ширина и высота холста в пикселах. По умолчанию холст имеет размер 640x400. Размеры обычно создаются при создании холста, но могут быть изменены и в ходе работы с ним.

`align` – параметр выравнивания. Может принимать значения "left" (холст придвигается к левой границе окна), "right" (холст сдвигается к правой границе), или "none" (по умолчанию). Если в программе используется только один холст, то использование значений `left` и `none` приводит к одинаковому расположению холста – около левой границы окна, но в первом случае подрисовочная подпись (`caption`) будет отображаться справа от холста. Выравнивание по левой границе часто используется для отображения графика справа от графического холста, при этом атрибут выравнивания у графика устанавливается в значение `right`. Если в этом случае требуется подпись под рисунком, то схема чуть другая: сначала нужно создать график с выравниванием по правому краю и активировать его каким-то набором точек, а только потом создать холст без указания выравнивания. Еще один распространенный вариант размещения холстов и графиков: задание у всех этих объектов выравнивания по левой границе. В этом случае точно не произойдет перестройки расположения объектов при уменьшении размера окна браузера.

`resizable` – логический параметр, значением которого по умолчанию является **True**, определяющий, может ли пользователь изменять размер холста.

`background` – цвет холста; по умолчанию – черный.

`foreground` – цвет переднего плана, т.е. цвет, используемый при создании на холсте новых объектов; по умолчанию – белый.

`ambient` – цвет ненаправленного («естественного») освещения. По умолчанию имеет значение `color.gray(0.2)`.

`lights` – список источников света (объектов класса `light`), имеющих на данном холсте. Некоторые подробности об этом списке приведены ниже в разделе «Освещение сцены».

`objects` – список всех видимых объектов холста (светильники и невидимые объекты в него не включаются). В качестве примера приведем код изменения цвета всех сфер холста `scene2` на красный:

```
for obj in scene2.objects:
    if isinstance(obj, sphere):
        obj.color = color.red
```

`visible` – логический параметр, установка которого в **False** приводит к тому, что все объекты холста перестают быть отображаемыми до тех пор, пока это параметр снова не установлен в **True**.

Метод холста `capture(filename)` приводит к загрузке через браузер скриншота этого холста в формате `png`. Расширение файла в строке `filename` можно не указывать.

**Представление содержимого холста на экране.** Приведенная на рис. 13 диаграмма из документации по пакету `VPython` [1.5], демонстрирует соотношение основных параметров, определяющих то, как холст и его содержимое будут отображаться на экране.

`center` – центр сцены, точка в пространстве, на которую направлен объектив камеры, даже если пользователь меняет положение камеры. При изменении координат этого центра камера перемещается так, чтобы ее направление осталось прежним, но направлена она была уже на новый центр. Чтобы, при необходимости, избежать такого перемещения камеры, необходимо корректировать атрибут `forward` (см. ниже). По умолчанию центр сцены находится в начале координат, т.е. представляет собой вектор  $(0, 0, 0)$ .

`forward` – вектор, отвечающий за направление камеры, т.е. вектор направленный из текущего положения камеры, `scene.camera.pos`, в центр сцены, `scene.center`. Изменение вектора `forward`, либо программное, либо пользовательское, например, при вращении камеры мышью, приводит к изменению положения камеры, чтобы она продолжала быть направленной в центр. По умолчанию значением этого атрибута является вектор  $(0, 0, -1)$ ; длина вектора не существенна.

`fov` – угол обзора (**field of view**) камеры в радианах. Этот параметр определяется как максимум из горизонтального и вертикального углов обзора. Таким образом, его можно отождествлять с угловым размером



более длинной стороны графического окна холста, видимой наблюдателем. Значением этого параметра по умолчанию является  $\pi/3$  радиан ( $60^\circ$ ).

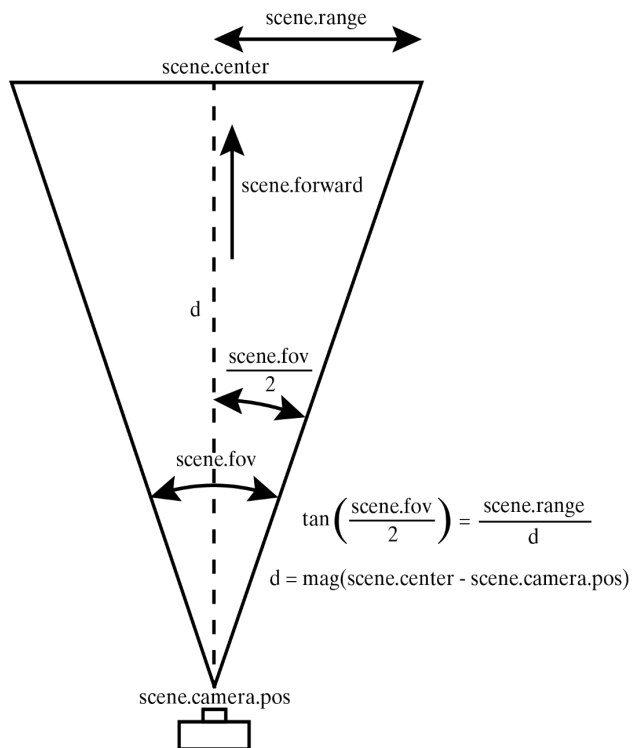
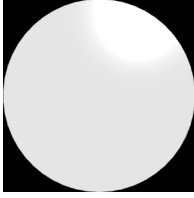


Рис. 13. Характеристики отображения сцены

`range` – протяженность наблюдаемой зоны влево и вправо от центра сцены. Таким образом, точка с абсциссой `scene.center.x+scene.range` будет находиться на правой границе окна. Куб с ребром, в два раза превышающим `range`, и даже сферический объект с радиусом `range` не поместятся полностью в окне просмотра, потому что размер их ближних к наблюдателю сечений плоскостью  $Oxy$  в 3D проекции окажется больше, чем  $2range \times 2range$ . В качестве примера на рис. 14 приведен код построения цилиндра, основание которого получается вписанным в квадратное окно сцены заданной протяженности.



```
scene2 = canvas(width=400, height=400,  
                range=10)  
cyl = cylinder(radius=scene2.range,  
              axis=vec(0,0,-1))
```

Рис. 14. Основание цилиндра, соответствующее протяженности наблюдаемой зоны

$up$  – вектор, соответствующий понятию «верха» трехмерного мира. Данный вектор перпендикулярен плоскости чертежа на рис. 13 и всегда проецируется на вертикальную линию графического окна. При «горизонтальном» вращении мышкой камера вращается вокруг этого вектора. По умолчанию он совпадает с осью  $Oy$ , т. е. вектором  $(0, 1, 0)$ .

Векторы  $up$  и  $forward$  не являются полностью независимыми. По умолчанию, камера направлена вдоль оси  $z$ , в сторону ее убывания, т. е. по вектору  $(0,0,-1)$ . В этом случае вектор  $up$  может быть любым вектором, перпендикулярным оси  $z$ , но не может стать сонаправленным с ней, потому что смена  $up$  осуществляется вращением камеры вокруг ее оси, т. е. вокруг оси  $z$ . Если требуется сделать ось  $z$  направленной вверх, сначала нужно изменить направление, т. е. вектор  $forward$ , камеры, например, ориентируя ее вдоль вектора  $(1,0,0)$ .

$autoscale$  – автоматическое масштабирование (по умолчанию **True**); если оно включено, то параметр  $range$  при необходимости увеличивается так, чтобы все объекты сцены были видны в графическом окне («камера отъезжает» от центра сцены). Практические рекомендации от разработчиков состоят в том, чтобы оставить этот параметр включенным до наполнения сцены объектами, а затем выключить, чтобы положение камеры не менялось автоматически из-за движения объектов.

Следующие атрибуты сцены могут быть использованы для «уменьшения» интерактивного влияния пользователя на построенное в окне браузера изображение:

$userzoom=False$  отключает пользовательское масштабирование,  
 $userspin=False$  отключает вращение сцены пользователем,  
 $userpan=False$  отключает панорамирование (перемещение камеры параллельно плоскости экрана)

**Освещение сцены.** По умолчанию, на любом вновь создаваемом холсте возникают два (удаленных) источника света:

```
[distant_light(direction=vec(0.22,0.44,0.88),
                color=color.gray(0.8)),
distant_light(direction=vec(-0.88,-0.22,-0.44),
                color=color.gray(0.3))]
```

Источники света, созданные по умолчанию, могут быть удалены путем присваивания их списку пустого значения: `scene.lights = []`. Так же просто изменить цвет любого из них, например, сделать первый красным: `scene.lights[0].color = color.red`. Любой из источников света можно включать и отключать, устанавливая параметр `visible` в **True** или **False** соответственно.

Кроме этих двух удаленных источников, на сцене присутствует естественное (ненаправленное) освещение, задаваемое параметром `scene.ambient`, по умолчанию имеющим значение `color.gray(0.2)`. Дополнительный удаленный источник света, например, голубого, расположенный в направлении  $(x, y, z)$  от центра сцены, создается командой

```
distant_light(direction=vector(x,y,z),
               color=color.cyan)
```

Цвета удаленных источников и величину естественного освещения надо выбирать с определенной осторожностью, поскольку если интенсивность освещенности некоторой части сцены превысит единицу, то результат отображения этой части не предсказуем.

Кроме удаленных источников света возможно создание источника местного освещения, или точечного источника:

```
lamp = local_light(pos=vector(x,y,z),
                   color=color.yellow)
```

Если в эту же позицию `pos` поместить сферу или цилиндр со свойством излучения света `emissive=True`, то получим некоторое подобие светящейся лампы.

**Интерактивные элементы.** Кроме холста (одного или нескольких) и графиков на веб-странице, создаваемой средствами VPython,

можно расположить стандартные интерактивные элементы, позволяющие управлять поведением и характеристиками объектов на сцене, запускать и останавливать анимацию и т. п. К этим элементам относятся кнопки, переключатели, ползунки, выпадающие меню, флажки (чекбоксы), поля ввода. Использование некоторых из этих элементов будет продемонстрировано на примере создания настраиваемой анимации согласно схеме, приведенной на рис. 15.

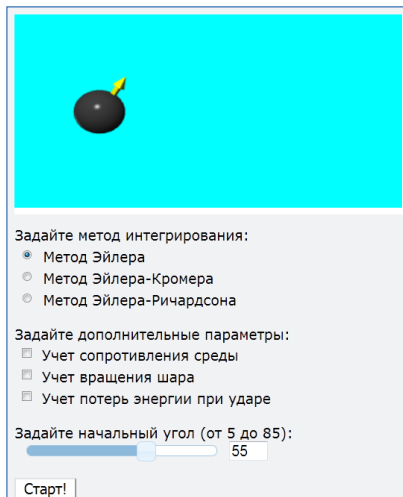


Рис. 15. Интерактивные элементы управления анимацией

Данной схеме соответствует код, приведенный ниже. Рассмотрим более подробно основные его части.

Фрагмент кода б.

```

1 from vpython import *
2 scene = canvas(width=400,height=200,
3               background=color.cyan)
4
5 ball = sphere(pos=vec(-5,0,-1),
6              color=color.gray(0.25))
7 arr = arrow(pos=ball.pos,
8            axis=vec(sqrt(2), sqrt(2),0),
9            color=color.yellow)
10
11 scene.append_to_caption('\n')
```

```

12 wtext(text='Задайте метод интегрирования:')
13 scene.append_to_caption('\n')
14
15 def rb_handler(rb):
16     for btn in radios:
17         if btn.id!=rb.id:
18             btn.checked = False
19
20 radios=[]
21 for i in range(3):
22     radios.append(radio(bind=rb_handler, id=i))
23     scene.append_to_caption('\n')
24 radios[0].text='Метод Эйлера'
25 radios[1].text='Метод Эйлера-Кромера'
26 radios[2].text='Метод Эйлера-Ричардсона'
27 radios[0].checked = True
28 scene.append_to_caption('\n')
29
30 wtext(text='Задайте дополнительные параметры:')
31 scene.append_to_caption('\n')
32
33 def cb_handler(cb):
34     pass
35
36 fr = checkbox(bind = cb_handler,
37               text='Учет сопротивления среды')
38 scene.append_to_caption('\n')
39 rot = checkbox(bind = cb_handler,
40               text='Учет вращения шара')
41 scene.append_to_caption('\n')
42 en = checkbox(bind = cb_handler,
43               text='Учет потерь энергии')
44 scene.append_to_caption('\n\n')
45
46 wtext(text='Задайте начальный угол (от 5 до 85):')
47 scene.append_to_caption('\n')
48
49 def sl_handler(sl):
50     # изменение ориентации стрелки
51     alpha = radians(sl.value)
52     arr.axis=2*vec(cos(alpha), sin(alpha), 0)
53     # вывод информации о текущем значении угла
54     inp.text = sl.value

```

```

55
56 ang = slider(bind=sl_handler,
57               min=5, max=85,
58               step=1, value=45, length=243)
59
60 def inp_handler(w):
61     n = int(w.number)
62     if n<5:
63         n = 5
64     elif n>85:
65         n = 85
66     ang.value = n
67     sl_handler(ang)
68
69 inp = winput(bind=inp_handler, type='numeric',
70             text='45', width=40)
71 scene.append_to_caption('\n\n')
72
73 def start(b):
74     pass
75
76 go = button(bind=start, text='Старт!' )

```

Строки с 1 по 10 создают изображение, представленное на холсте – шар со стрелкой. Начальная ориентация стрелки – 45 градусов относительно горизонтали – соответствует значению ползунка, описанного ниже. Интерактивность данного кода, разумеется, минимальна – перемещение ползунка, либо задание значение величины угла приводит к изменению ориентации стрелки. Важно отметить, что созданный для рисования 3D-изображения холст называется `scene` – интерактивные элементы в данном примере располагаются именно в зоне «подписи» (`caption`) к этой сцене.

Кроме интерактивных элементов в данном фрагменте кода используется объект `wtext`, основное назначение которого – вывод на экран (вне зоны области рисования) текстовой информации. Этот объект используется в строках 12, 30 и 46.

Обратите внимание – и объект `wtext`, и интерактивные элементы располагаются вне зоны рисования, в области подписи к ней, в соответствии с правилами `html`-разметки, то есть рядом друг с другом. Для

того, чтобы расположить следующий элемент не рядом, а под предыдущим, используется добавление символа перевода строки в области подписи – команда `scene.append_to_caption('\n')` в строках 11, 13, 23, 28, 31, 38, 41, 47, 71. Если требуется дополнительный вертикальный пробел между элементами, символ `\n` можно повторить несколько раз.

Кроме зоны подписей к сцене на `html`-странице создается и так называемая область заголовка (`title`). Для размещения элемента в зоне заголовка (над 3D-сценой) используется параметр `pos` в формате `pos=scene.title_anchor`. «Перевод строк» в этой области осуществляется командами вида `scene.append_to_title('\n')`.

Радиокнопки, или переключатели, создаются функцией `radio`. На примере этой функции остановимся на общем для всех интерактивных элементов параметре `bind`, задающем функцию – обработчик события взаимодействия пользователя с интерактивным элементом. Отсутствие этого параметра при создании любого интерактивного элемента в текущей версии `VRPython 7` приводит к сообщению об ошибке

```
AttributeError: bind missing
```

В строках 20–23 обсуждаемого кода создается список переключателей, каждый из которых имеет свой идентификатор, `id`. Этот параметр не является стандартным, он использован в качестве примера создания объекта в Питоне с пользовательским свойством. В строках 24–26 задается текст, расположенный справа от переключателя. Первый из группы переключателей считается заданным (`checked`) по умолчанию (строка 27).

Существенным недостатком переключателей `VRPython` является невозможность их объединения в группу так, чтобы выбор одного автоматически отключал другой. Приведенная в примере кода (строки 15–18) функция-обработчик нажатия на переключатель `rb_handler` – единая для всех трех переключателей – предназначена исключительно для решения этой проблемы: она отключает (делает неактивными) все переключатели, кроме того, на котором она была вызвана. Проверка осуществляется на основе параметра `id` переключателя.

Строки 36–37, 39–40, 42–43 создают последовательно три чекбоксы или флажковых переключателя. Они могут устанавливаться или сниматься

независимо; функция `cb_handler` в данном случае используется чисто номинально, она ничего не делает, но наличие хоть какого-то пользовательского обработчика нажатия необходимо. Как и в случае радиокнопок, параметр `text` отвечает за текст, расположенный справа от флажка.

В строках 56–58 создается слайдер; в данном примере он отвечает за ориентацию стрелки. Кратко перечислим использованные в примере параметры:

`min` – минимальное значение слайдера (по умолчанию 0),  
`max` – максимальное значение слайдера (по умолчанию 1),  
`step` – минимально возможная величина перемещения слайдера (по умолчанию  $0.001 * (max - min)$ ),

`value` – текущая величина (между `min` и `max`) слайдера. Эта величина меняется интерактивно пользователем, а также может быть установлена программой,

`length` – размер слайдера (по умолчанию 400 пикселей). Число 243 в данной программе получилось на основании таких расчетов: восемьдесят один возможный вариант значения (от 5 до 85 включительно) умножен на три пиксела.

К остальным важным параметрам данного объекта следует отнести:

`vertical` (по умолчанию **False**): установка этого параметра в **True** создает вертикальный слайдер;

`width` (по умолчанию 10 пикселей) – высота (для горизонтального) и ширина (для вертикального) слайдера;

`left`, `right` (по умолчанию 12 пикселей) – поле слева и справа от слайдера соответственно;

`top`, `bottom` (по умолчанию 0 для горизонтального, для вертикального требуется не менее 8) – поле над и под слайдером соответственно.

Значения полей по умолчанию являются минимально возможными, чтобы предотвратить пересечение с соседним текстом.

Обработчик события перемещения ползунка слайдера, функция `sl_handler`, решает две задачи: изменение ориентации стрелки (строки 51–52) в соответствии с углом наклона, заданным слайдером, и вывод



информации о значении угла в градусах в очередной интерактивный объект – поле ввода (строка 54).

Поле ввода – объект `winput` – создается в строках 69–70 рассматриваемого кода. Параметры `text` и `width` имеют достаточно естественный характер, более интересен параметр `type`. Значение `'string'` позволяет вводить в это поле любое значение, а значение `'numeric'` – числа и числовые выражения, например, `2+3`. Обработка содержимого поля ввода происходит, когда фокус находится на нем, после нажатия пользователем клавиши `Enter`. В случае некорректного значения возникает окно с сообщением об ошибке, и функция-обработчик события не вызывается. Если числовое значение корректно, то оно помещается в параметр `number` данного объекта.

В рассмотренном примере обработчик `inp_handler` проверяет, что введенное в поле ввода значение (параметр `number`) находится в диапазоне от 5 до 85, при необходимости осуществляя корректировку (строки 62–65), и присваивает это значение параметру `value` слайдера (строка 66). Следует отметить, что программное изменение параметра `value` изменяет положение ползунка слайдера, но не приводит к вызову обработчика события перемещения ползунка, поэтому для поворота стрелки на холсте в новое положение этот обработчик приходится вызывать вручную (строка 67).

Последним интерактивным элементом является кнопка (`button`). В данном примере обработчик события нажатия на нее является формальным – функция `start` не выполняет никаких действий. В реальных программах эта функция может, например, начать выполнение анимации, добавить или удалить объект и т. п.

Общим для всех интерактивных элементов является параметр `disabled`, установка которого в **True** приводит к тому, что данный элемент становится серым и неактивным.

## ЛИТЕРАТУРА К ГЛАВЕ 1

- 1.1. Гетманова Е. Е. Visual Python – язык для моделирования физических явлений [Электронный ресурс] // Компьютерные инструменты в образовании. – 2005, № 3. – С. 7–21. – URL: <http://ipo.spb.ru/journal/article/619/> (дата обращения 01.02.2021)
- 1.2. Гетманова Е. Е. Использование Visual Python для моделирования физических процессов [Электронный ресурс] // Компьютерные инструменты в образовании. – 2005, № 4. – С. 43–47. – URL: <http://ipo.spb.ru/journal/article/684/> (дата обращения 01.02.2021)
- 1.3. Разрешение использования изображений из разных источников и canvas [Электронный ресурс] – URL: [https://developer.mozilla.org/ru/docs/Web/HTML/CORS\\_enabled\\_image](https://developer.mozilla.org/ru/docs/Web/HTML/CORS_enabled_image) (дата обращения 01.02.2021)
- 1.4. Flot: Attractive JavaScript plotting for jQuery [Электронный ресурс] – URL: <https://www.flotcharts.org> (дата обращения 01.02.2021).
- 1.5. GlowScript VPython and VPython 7. Documentation [Электронный ресурс] – URL: <https://www.glowscript.org/docs/VPythonDocs/index.html> (дата обращения 01.02.2021)
- 1.6. Morgan W.A., English L.Q. VPython for Introductory Mechanics: Complete Version [Электронный ресурс] – URL: <https://scholar.dickinson.edu/vpythonphysics/1> (дата обращения 01.02.2021)
- 1.7. Philhour B. Physics through GlowScript – an Introductory Course [Электронный ресурс] – URL: <https://bphilhour.trinket.io/physics-through-glowscript-an-introductory-course> (дата обращения 01.02.2021)
- 1.8. Plotly Python Open Source Graphing Library [Электронный ресурс] – URL: <https://plotly.com/python/> (дата обращения 01.02.2021)
- 1.9. Schroeder D. V. Physics Simulations in Python: [Электронный ресурс]. – URL: <https://physics.weber.edu/schroeder/scicomp/PythonManual.pdf> (дата обращения 01.02.2021)
- 1.10. Olenick R., Case D., Peping E., Spearman W. VPython Simulations in a Computational Physics Course [Электронный ресурс] – URL: <https://doi.org/10.13140/RG.2.1.3661.7202> (дата обращения 01.02.2021)
- 1.11. WebGL – Кросс-доменные изображения [Электронный ресурс] – URL: <https://webglfundamentals.org/webgl/lessons/ru/webgl-cors-permission.html> (дата обращения 01.02.2021)

## ГЛАВА 2. ПРОЕКТНОЕ ЗАДАНИЕ «ВИЗУАЛИЗАЦИЯ МОЛЕКУЛЫ»

### ФОРМУЛИРОВКА ЗАДАНИЯ

Требуется разработать программу, визуализирующую структуру молекулы вещества, указанного в варианте проектного задания согласно Таблице 2. Схема визуализации должна в той или иной мере соответствовать изображению, приведенному на рис. 16, одинаковые атомы должны изображаться шариками одного цвета, атомы разных элементов – шариками разных цветов. Связи между атомами должны быть изображены достаточно тонкими цилиндрами, при этом одинарной связи должен соответствовать один цилиндр, двойной – соответственно, два. Цвет цилиндров, изображающих связь, должен соответствовать цветам соединяемых ею атомов, если атомы разные – цилиндр должен быть двухцветным. Тип окрашивания такого цилиндра – резкий переход цвета или смешивание цветов – указан в Таблице 2. Над окном с изображением молекулы должно быть представлено название вещества и его химическая формула.

В качестве интерфейсного элемента должна быть реализована возможность включения/отключения меток, обозначающих название атома. Расположение метки относительно изображения атома должно соответствовать варианту из Таблицы 3.

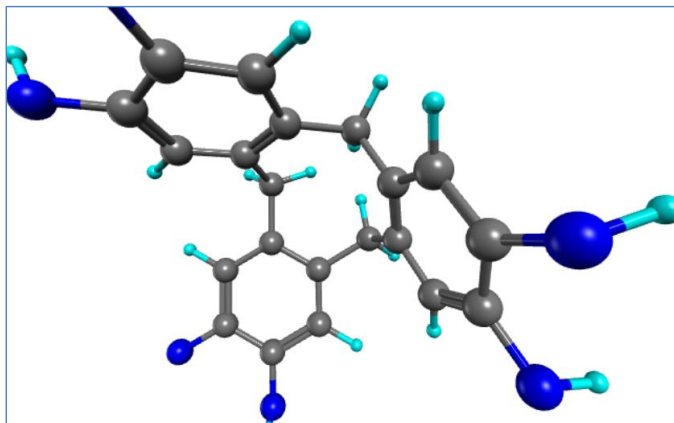


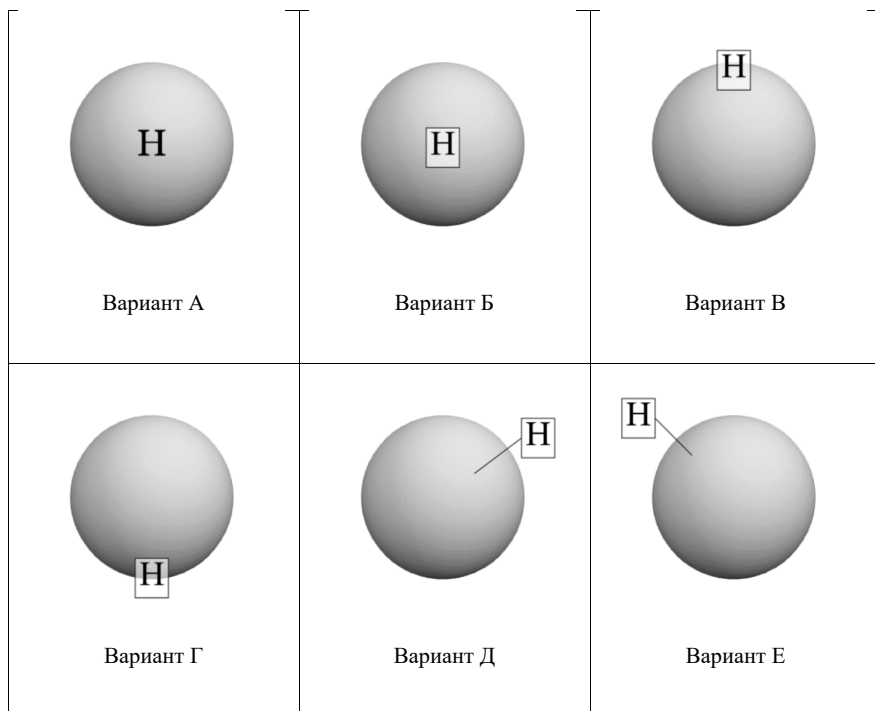
Рис. 16. Примерный ожидаемый результат визуализации

Таблица 2. Варианты проектного задания

Номер варианта	Название вещества, химическая формула	Вариант расположения меток	Тип окрашивания
1	Cholesterol, $C_{27}H_{46}O$	А	Резкий переход
2	Alpha-eucaine, $C_{19}H_{27}NO_4$	Б	Смешивание цветов
3	2-methylpropyl 4-phenylbutanoate, $C_{14}H_{20}O_2$	В	Резкий переход
4	2,3-Diphenylquinoxaline, $C_{20}H_{14}N_2$	Г	Смешивание цветов
5	Bromopride, $C_{14}H_{22}BrN_3O_2$	Д	Резкий переход
6	6-Benzoyl-2-naphthol, $C_{17}H_{12}O_2$	Е	Смешивание цветов
7	Aequinetin, $C_{21}H_{20}O_9$	А	Резкий переход
8	Trp-AFC, $C_{21}H_{16}F_3N_3O_3$	Б	Смешивание цветов
9	Sempervilam, $C_{19}H_{16}N_2$	В	Резкий переход
10	Decyl hydrogen phthalate, $C_{18}H_{26}O_4$	Г	Смешивание цветов
11	Hexaphenol, $C_{21}H_{18}O_6$	Д	Резкий переход
12	Cyclenar, $C_{15}H_{27}N_5$	Е	Смешивание цветов
13	Lamotrigine, $C_9H_7Cl_2N_5$	А	Резкий переход
14	Andrograpanin acetate, $C_{22}H_{32}O_4$	Б	Смешивание цветов
15	ROSIN, $C_{15}H_{20}O_6$	В	Резкий переход
16	Cannabifuran, $C_{21}H_{26}O_2$	Г	Смешивание цветов
17	Sorgomol, $C_{19}H_{22}O_6$	Д	Резкий переход
18	Hemistepsin, $C_{19}H_{22}O_6$	Е	Смешивание цветов

19	Asperinin A, $C_{32}H_{26}O_{10}$	А	Резкий переход
20	7281P, $C_{40}H_{44}O_6$	Б	Смешивание цветов
21	AMINOPYRINE, $C_{13}H_{17}N_3O$	В	Резкий переход
22	Melatonin, $C_{13}H_{16}N_2O_2$	Г	Смешивание цветов
23	Cortisone, $C_{21}H_{28}O_5$	Д	Резкий переход
24	Hexahelicene, $C_{26}H_{16}$	Е	Смешивание цветов

Таблица 3. Варианты расположения меток



**Химическая информатика.** Знакомство с основными возможностями пакета VPython начнется с выполнения проектного задания, связанного с построением статической трехмерной модели молекулы некоторого заданного вещества. Задачи такого класса (правда, гораздо более серьезные) относятся к науке, известной под названием химическая информатика (или хемоинформатика<sup>11</sup>, или даже хеминформатика, от английского варианта cheminformatics).

Хемоинформатика – это относительно новая комплексная междисциплинарная область науки, связанная с использованием информационных технологий для сбора, хранения, анализа и обработки химических данных. Сами эти данные обычно включают информацию о формулах, структурах, свойствах, спектрах молекул химических веществ и соединений, а также возможной роли и областях применения этих соединений (как в биологии живых организмов, так и в промышленности).

Строго говоря, классические методы хранения и поиска химических данных существовали задолго до появления компьютеров. К их числу относятся, например, всевозможные печатные указатели (авторские, предметные, формульные и т. д.). Однако взрывной рост количества как синтезированных или обнаруженных химических соединений, так и публикаций об этих соединениях сделал весьма неэффективным ручной труд в этой сфере. В качестве примера можно привести данные о ежедневно обновляемой базе данных CAS (Chemical Abstracts Service) [2.5] – подразделения Американского химического общества, где хранится информация о 163 миллионах органических и неорганических веществ, включая соли, сплавы, координационные соединения, минералы, смеси и полимеры, опубликованных в литературе с начала 1800-х годов. В этой же базе находятся данные о 68 миллионах последовательностей белков и нуклеиновых кислот и о 127 миллионах химических реакций. Информация о структуре соединений дополняется сведениями об их экспериментально установленных свойствах (четыре с половиной миллиона параметров для трех миллионов веществ) и предсказанных

---

<sup>11</sup> <https://ru.wikipedia.org/wiki/Хемоинформатика>

характеристик (более семи с половиной миллиардов параметров для более чем ста миллионов веществ).

Хемоинформатика оформилась как самостоятельная наука на рубеже 20–21 вв. в результате развития таких направлений исследований, как хранение и обработка химической информации, математическая и вычислительная химия, анализ связи структуры соединений с их свойствами и биологической активностью, молекулярное моделирование, анализ экспериментальных данных, конструирование соединений с заданными свойствами и оптимизация условий реакций [2.4]. Хемоинформатика изначально возникла как средство, помогающее процессу открытия и разработки лекарств, однако в настоящее время она играет все более важную роль во многих областях биологии, химии и биохимии. Хемоинформатика тесно связана также и с биоинформатикой.

В последние годы в сфере интересов хемоинформатики попали методы машинного обучения (machine learning, ML) [2.11]. Здесь возникает много интересных задач, связанных с преобразованием трехмерных структур молекул в исходные данные для ML-моделей, решением проблем нехватки данных, выделением тех свойств молекул, которые наиболее важны для предсказаний свойств и результатов реакций.

Что касается исходных данных для моделирования, то основные химические информационные базы делятся на коммерческие (платные) и открытые. К коммерческим относится упомянутая выше CAS. База данных Mol-Instincts [2.12] тоже является коммерческой, однако часть информации о соединениях, например, файлы с данными о пространственной структуре молекул, может быть получена свободно, как будет продемонстрировано позднее.

К базам данных с открытым доступом относятся:

PubChem [2.13] – самая большая в мире коллекция свободно доступной химической информации. Обслуживается Национальным центром биотехнологической информации США (NCBI), подразделением Национальной медицинской библиотеки США, которая в свою очередь является подразделением Национального Института Здоровья США (NIH). База содержит около 100 миллионов соединений и более 236 миллионов субстанций. Здесь необходимо отметить, что в этой системе под *соединением* понимается объект с уникальной стандартизированной

химической структурой, так что одному химическому соединению соответствует хотя бы одна *субстанция*. Обратное уже не верно: субстанции может не соответствовать соединения, если его структура не стандартизирована или вообще не известна; примером такой субстанции является зеленый чай. База PubChem хорошо документирована, содержит большое количество руководств и инструкций для пользователя. Открытый характер этой базы означает не только возможность свободного получения информации, но и возможность исследователям размещать в системе свои результаты.

ChemSpider [2.8] – база данных, аналогичная PubChem, но с меньшим количеством химических соединений. В настоящее время она предоставляет возможности быстрого поиска и доступа к данным о 86 миллионах структур, полученных из 275 источников.

ChEMBL [2.6] – тщательно отобранная и структурированная база данных биоактивных молекул с лекарственными свойствами. Цель создания этой базы – объединение химических, биологических и геномных данных для помощи создателям новых эффективных лекарств. В настоящее время в версии ChEMBL 27 базы содержится около двух миллионов химических соединений и более 16 миллионов описаний реакций. Разработчики базы активно используют интеллектуальный анализ данных из научных статей и патентных документов для получения новых данных о молекулах.

**Хранение и визуализация структурных данных.** Ясно, что из всего разнообразия областей хемоинформатики нас будет интересовать лишь малая часть: как хранится информация о составе молекул химических соединений, какие форматы файлов существуют, как их можно получить и как обрабатывать, какие способы визуализации структуры молекул считаются общепринятыми и как их реализовать. Строго говоря, само выполнение проектного задания потребует работы с файлом только одного, вполне определенного типа, поиска его в заданной базе данных и визуализации структуры молекулы только с помощью одной конкретной схемы. Более широкая информация, приведенная в данном разделе, может оказаться полезной для решения схожих задач, но уже выходящих за рамки сформулированного проектного задания.



Прежде всего, выберем схему визуализации. Если говорить о трехмерных моделях, то в большинстве систем отображения структуры молекул на экране или в документах используются четыре варианта, приведенные на рис. 17. Эти четыре изображения молекулы кофеина,  $C_8H_{10}N_4O_2$  построены в системе PubChem [2.13].

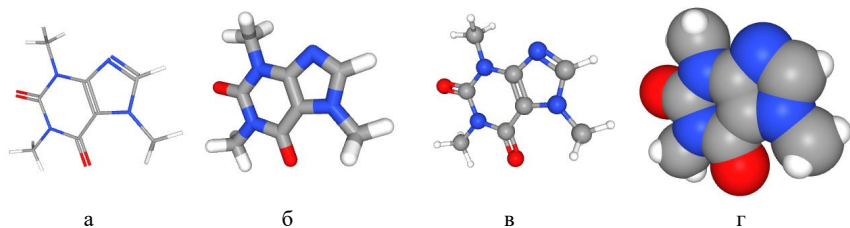


Рис. 17. Модели химических структур (на примере молекулы кофеина): а – каркасная, б – стержневая, в – шаростержневая, г – пространственное заполнение

Сравнение проектного задания с этим рисунком показывает, что построить необходимо шаростержневую модель молекулы, в которой атомы обозначаются сферами (шариками), а связи между ними – трубочками или стерженьками цилиндрической формы. При этом одинарной связи соответствует один стержень, двойной – два. Шарик, обозначающий атомы различных химических элементов, различаются по цвету и размеру; цвет трубочек-связей соответствует цветам атомов, которые они соединяют.

Ниже, следуя работе [2.10], будет представлена информация о некоторых распространенных форматах хранения информации о структуре молекулы вещества. Этот раздел не претендует на полноту информации, поскольку таких форматов достаточно много, и они продолжают появляться, например, в новых системах моделирования методами молекулярной динамики. Его цель – дать представление об имеющемся разнообразии и предоставить возможность решить задачу визуализации структуры молекулы на основе заданного файла в одном из существующих форматов. Все форматы, представленные здесь, являются текстовыми, т. е. могут быть интерпретированы человеком на основе изучения файлов, открытых в любом простейшем редакторе. Бинарные форматы для таких задач, разумеется, тоже существуют и в ряде случаев (когда необходимо обрабатывать большое количество многоатомных молекул) являются более эффективными, но их рассмотрение выходит за рамки данного пособия.

**Формат XYZ.** Данный формат является одним из самых простых, именно поэтому он достаточно распространен, несмотря на то что информации в XYZ-файлах содержится заметно меньше, чем в большинстве остальных форматов. Продемонстрируем хранение данных в этом формате на примере молекулы воды, H<sub>2</sub>O:

```
1 3
2 Water - XYZ format.
3 O 0.00 0.00 0.0
4 H 0.58 0.75 0.0
5 0.58 -0.75 0.0
```

*Строка 1* содержит информацию о количестве атомов в молекуле

*Строка 2* представляет собой заголовок или комментарий. При отсутствии заголовка эта строка остается пустой.

*Строки 3–5* содержат данные об атомах: сначала символ элемента или его атомный номер, а затем три вещественных числа, представляющих координаты атома в трехмерном пространстве (*x*, *y* и *z*). Количество строк с данными должно совпадать с числом атомов, указанным в первой строке.

Формат XYZ является т. н. свободным (free) форматом в отличие от фиксированного (fixed) формата. В случае фиксированного формата информационные единицы (символы, числа, слова) должны располагаться в строго определенных колонках; вещественные числа должны быть записаны с фиксированным количеством десятичных цифр после запятой. В случае свободного формата ограничений на запись параметров нет; единственное требование – возможность четко отделить один параметр от другого. В качестве разделителей обычно используются пробелы.

**Формат MOL.** Данный формат является фиксированным; он был введен в употребление компанией MDL Information Systems. Вот какой вид будет иметь файл с данными о молекуле воды в этом формате:

```
1 Water - MOL format.
2
3
4 3 2 0 0 0 0 0 0 0 0 0999 v2000
5 0.0000 0.0000 0.0000 O 0 0 0 0 0 0 0
6 0.5800 0.7500 0.0000 H 0 0 0 0 0 0 0
7 0.5800 -0.7500 0.0000 H 0 0 0 0 0 0 0
8 1 2 1 0 0 0
9 1 3 1 0 0 0
10 M END
```

*Строки 1-3* образуют блок заголовка (он всегда состоит из трех строк), остальные – представляют собой собственно данные об атомах и связях между ними.

В строках заголовка может располагаться название молекулы или соединения, информация о версии программного обеспечения или пакета, в котором был создан данный файл, комментарии различного рода. Любая из этих строчек может быть пустой.

*Строка 4* (строка количественных параметров, или линия подсчета – counts line) содержит несколько целых чисел, определяющих основные параметры таблицы данных о химическом соединении. Всего здесь есть место для двенадцати полей фиксированной ширины: 11 полей по три символа и одно поле из шести символов. Максимально допустимое значение каждого из 11 полей – 999; числа в пределах каждого поля выравнены по правому краю. Наиболее важными являются первые два поля: в них содержится информация об общем количестве атомов и числе связей между этими атомами в описываемом химическом соединении. В рассматриваемом примере эти числа равны 3 и 2, соответственно, что означает наличие в файле 3 строчек с информацией о каждом атоме и двух строчек с информацией о межатомных связях.

В остальных полях этой строки может содержаться, например, информация о хиральности молекулы, о наличии других дополнительных свойств и т. д. Наличие нулей в этих полях обязательно лишь формально, многие системы допускают наличие пробелов вместо нулей. Строчка V2000 в последней позиции приведенного примера отсылает к версии формата MOL файла; версия V2000 является де-факто стандартом в настоящее время. Модификация V3000 используется, например, если значения в 999 уже недостаточно.

Некоторые программные системы допускают отсутствие в MOL файле атомов водорода, связанных с атомами кислорода и углерода. Его наличие и связи в этом случае восстанавливается автоматически на основании информации о валентности элементов<sup>12</sup>.

*Строки 5–7* в рассматриваемом примере образуют блок атомов. Каждая из них имеет следующий формат: 3 позиции для координат атома в

---

<sup>12</sup> Файлы, использующиеся при выполнении проектного задания, будут содержать полную информацию.

ангстремах в формате 10.4f (всего 10 символов, из которых 4 находятся в дробной части числа; выравнивание по правому краю), затем (в колонках с 32 по 34) символ элемента, выровненный по левому краю. Оставшиеся позиции (целые числа, по три символа на число) могут содержать информацию о заряде, стереохимических особенностях атома, его нестандартной валентности и некоторых других свойствах. Число таких позиций не может превышать 12.

*Строки 8–9* представляют собой блок связей, являющийся фактически текстовым представлением молекулярного графа химического соединения. Каждая строка в этом блоке состоит из шести или семи целых чисел, на каждое из которых отведено по три символа. Первые два числа обозначают номера атомов, между которыми существует связь. Под номером атома здесь понимается порядковый номер строки этого атома в блоке атомов, счет начинается с единицы. Третье число указывает на тип связи: одинарная (1), двойная (2), тройная (3), ароматическая (4). Четвертый параметр связан со стереохимическим характером связи (0 означает, что у связи нет стерео-специфичности).

Необязательный (за одним исключением) раздел файла – это раздел свойств. Строки этого раздела начинаются с символа M. В этом разделе могут быть уточнены заряды отдельных атомов (команда M CHG), наличие изотопов (M ISO) и др. К обязательным относится только строчка M END, завершающая файл (строка 10).

**Формат SDF.** Данный формат, как и предыдущий, разработан специалистами компании MDL. Аббревиатура "SDF" означает structure-data file, то есть файл с информацией о структуре. Файлы SDF представляют, по сути, обертку над форматом MOL. Основных отличий здесь два. Во-первых, это возможность объединения в одном файле информации о нескольких химических соединениях (разделитель между данными об отдельных соединениях, фактически совпадающими с содержанием их MOL файлов, представляет собой строку из четырех знаков доллара). Во-вторых, появление в файле еще одного раздела с так называемыми метаданными. Различные базы данных химических веществ и соединений, разнообразное программное обеспечение, работающее с файлами структуры, включают в этот раздел большое количество дополнительной фактической информации. В этом разделе может,

например, находится химическая формула данного соединения в различных нотациях, температурные и другие физические характеристики и т.п. Каждое поле данных должно начинаться с символа >, за которым в угловых скобках приводится имя этого поля. В следующей строке, или нескольких строках, длиной до 200 символов, располагается соответствующее значение. Одно поле отделяется от другого пустой строкой. Завершается файл строкой \$\$\$\$.

В качестве примера ниже приведем (сокращенный!) вариант sdf-файла молекулы воды, полученный из библиотеки PubChem [2.13]

```

1 962
2 -OEChem-07082010102D
3
4 3 2 0 0 0 0 0 0999 V2000
5 2.5369 -0.1550 0.0000 O 0 0 0 0 0 0 0 0
6 3.0739 0.1550 0.0000 H 0 0 0 0 0 0 0 0
7 2.0000 0.1550 0.0000 H 0 0 0 0 0 0 0 0
8 1 2 1 0 0 0 0
9 1 3 1 0 0 0 0
10 M END
11 > <PUBCHEM_COMPOUND_CID>
12 962
13
14 > <PUBCHEM_IUPAC_SYSTEMATIC_NAME>
15 oxidane
16
17 > <PUBCHEM_IUPAC_TRADITIONAL_NAME>
18 water
19
20 > <PUBCHEM_IUPAC_INCHI>
21 InChI=1S/H2O/h1H2
22
23 > <PUBCHEM_EXACT_MASS>
24 18.010565
25
26 > <PUBCHEM_MOLECULAR_FORMULA>
27 H2O
28
29 > <PUBCHEM_MOLECULAR_WEIGHT>
30 18.015
31
32 $$$$

```

Из приведенного примера видно, что с точки зрения задачи визуализации трехмерной структуры молекулы, форматы MOL и SDF эквивалентны.

**Формат PDB.** Аббревиатура PDB расшифровывается как Protein Data Bank, то есть Банк данных о белках. И означает она, прежде всего, именно базу данных с информацией о трехмерной структуре больших биологических молекул, таких как белки и нуклеиновые кислоты. Эти данные, как правило, получены методами рентгеновской кристаллографии, ЯМР-спектроскопии или криоэлектронной микроскопии.

Банк данных PDB был создан в 1971 году. В 1998 года ответственность за его функционирование взяло на себя RCSB – Исследовательское партнерство в области структурной биоинформатики. В 2003 году была создана wwPDB (worldwide PDB) – международная организация, управляющая архивом данных и гарантирующая свободный доступ к его содержимому всему исследовательскому сообществу.

Биологические молекулы обычно очень сложны, они насчитывают тысячи атомов. Кроме белковой цепи у них часто имеется дополнительная небелковая группа – лиганд. В роли лиганда могут выступать, например, молекулы, выполняющие в белке структурную функцию – липиды, углеводы, нуклеиновые кислоты, минеральные элементы, какие-либо другие органические соединения: гем в гемоглобине, углеводы в гликопротеинах, ДНК и РНК в нуклеопротеинах и т. д.

В качестве примера на рис. 18 приведено схематическое изображение белка *миоглобин*, структура которого состоит из белковой цепи, содержащей около 150 аминокислот, и лиганда. Как внутри живого организма, так и при кристаллографическом исследовании структуры, белок находится в растворе воды и ионов. Подавляющее большинство этих молекул не будет определяться экспериментально, не показаны они и на рисунке. Однако какая-то их часть все равно попадает в результаты обработки экспериментальных данных. Задача «очистки» информации о структуре белка, полученной на основе эксперимента, от молекул растворителя часто представляет собой самостоятельную исследовательскую проблему.

Что касается формата файла PDB, первая версия которого появилась в 1976 году, то он разрабатывался как человекочитаемый (т. е. текстовый) формат для хранения в базах и обмена между исследователями данными о координатах атомов в белковых молекулах. PDB представляет собой фиксированный формат, данные в котором располагаются в строчках

длиной в 80 символов, что соответствует размеру перфокарты – основной единицы хранения информации того времени. За время своего существования данный формат неоднократно усовершенствовался; его последняя версия, утвержденная в 2012 году, имеет номер 3.3.

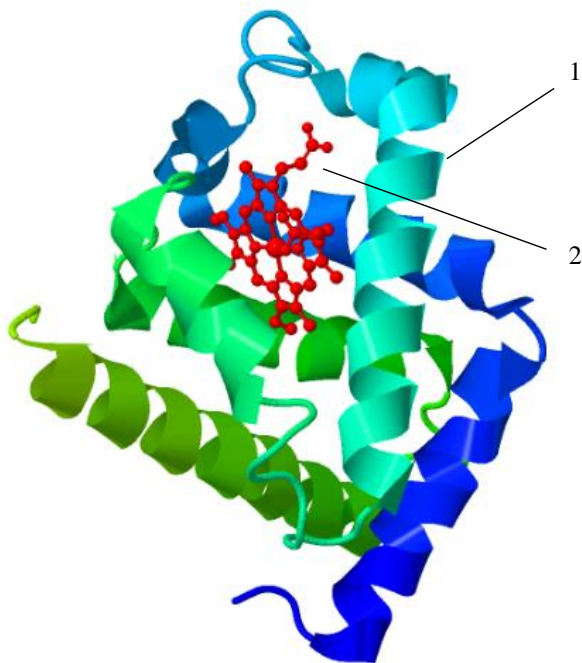


Рис. 18. Структурная схема белка миоглобин: 1 – белковая цепь, 2 – лиганд (гем). Изображение получено с помощью системы визуализации сайта <https://www.rcsb.org>

Сложной структуре соединений, описываемых форматом PDB, соответствует достаточно сложная схема иерархического структурирования и классификации данных, состоящая из *цепочек*, *остатков* и *атомов*. Цепочка представляет собой непрерывный участок пептида, белка или нуклеиновой кислоты, и каждой цепочке в системе присваивается уникальное односимвольное имя. В свою очередь цепочка подразделяется на остатки, так что пептидные и белковые цепочки состоят из последовательностей аминокислотных остатков, а цепочки нуклеиновых кислот содержат нуклеотидные остатки. Каждый остаток

идентифицируется трехсимвольным именем, а также кодом, чаще всего целым числом, определяющим его положение в последовательности. Наконец, остатки делятся на атомы, причем каждый атом в остатке имеет уникальное имя, самое большее, из четырех символов.

Реальная классификация еще чуть более сложная, потому что остатки могут быть двух типов. Стандартные остатки включают около двадцати аминокислот, встречающихся в белках, либо пять-шесть нуклеотидов, встречающихся в рибо- и дезоксирибонуклеиновых кислотах. Все остальные остатки являются нестандартными или гетероатомными. Например, в случае миоглобина в цепи белка будет присутствовать лиганд – гем, который обеспечивает миоглобину красный цвет и способность взаимодействовать с кислородом. Большое количество гетероатомных остатков вообще не объединяется в цепочки, поскольку являются упомянутыми выше молекулами или ионами растворителя.

Каждая строка PDB файла является самостоятельной записью, содержащей данные определенного типа. Этот тип определяется первыми шестью символами в строке. Полная спецификация возможных значений весьма обширна, даже краткий ее обзор выходит далеко за рамки данного учебного пособия. Отметим только, что файл с данными о структуре миоглобина, представленного на рис. 18, содержит 1817 строк. В качестве простого, хотя и не вполне показательного, примера, ниже приведен возможный вариант для хранения в этом формате информации о молекуле воды

1	TITLE	WATER - PDB FORMAT.										
2	COMPND	Water										
3	HETATM	1	O	HOH	1	0.251	-0.360	-0.046	1.00	0.00	O	
4	HETATM	2	1H	HOH	1	0.249	0.684	0.231	1.00	0.00	H	
5	HETATM	3	2H	HOH	1	0.586	-0.954	0.791	1.00	0.00	H	
6	CONNECT	1	2	3								
7	MASTER	0	0	0	0	0	0	0	3	0	1	0
8	END											

*Строка 1* содержит заголовок. Признаком этого является тип записи TITLE. Сам заголовок может быть любым, он начинается с 11-го символа в строке и может продолжаться до 70-й позиции. Если заголовок длинный, то записей типа TITLE может быть несколько.

*Строка 2* содержит информацию о названии химического соединения. Правила ее оформления аналогичны строке заголовка. Еще одним примером чисто текстовой записи может служить строка или строки



AUTHOR, в которых можно перечислить экспериментаторов, определивших структуру вещества или сведения об использованном программном обеспечении.

*Строки 3–5* содержат данные о координатах атома. В типичных файлах они имеют тип АТОМ, в данном же случае это НЕТАТМ, поскольку вода не относится к аминок- или нуклеиновым кислотам и, следовательно, представляет собой гетероатомный остаток. Однако, кроме первых шести символов, эти строки содержат одинаковую по смыслу информацию:

- в позициях 7–11 содержится порядковый номер атома в записанной последовательности молекулы;

- позиции 13–16 предназначены для имени атома. В формате PDB каждый атом из остатка имеет уникальное имя, на которое отводится до четырех символов. Имя атома водорода может начинаться с цифры;

- в позициях 18–20 указывается имя остатка; НОН – это принятое в PDB обозначение воды;

- позиции 23–26 содержат порядковый номер остатка в общей структуре хранимой в файле информации о молекуле. В данном случае в файле находится только один остаток, поэтому его номер – 1;

- в позициях 31–54 находятся координаты x, y и z данного атома в ангстремах; формат этих чисел 8.3f;

- позиции 55–66 определяют два вещественных числа (в формате 6.2f), содержащие некоторую специфическую информацию об эксперименте;

- в позициях 77–78 может храниться символ элемента из химической таблицы, выровненный по правому краю.

*Строка 6* представляет собой запись с информацией о связях между атомами соединения. Целое число в позициях 7–11 соответствует порядковому номеру атома в связи, а целые числа в позициях 12–16 и 17–21 представляют собой порядковые номера тех атомов, с которыми он связан. В данном случае у атома кислорода имеется связь с каждым из атомов водорода. Заметим, что в отличие от формата MOL/SDF данный формат не поддерживает хранение информации о типе ковалентной связи. Разумеется, атом может иметь больше двух связей, поэтому в общем случае в данной записи могут быть заполнены поля 22–26, 27–31 и т. д.

Строка 7 представляет собой так называемую MASTER-запись. Она используется для хранения итоговой информации о наличии и количестве записей различных типов в файле. Всего в этой записи находится 12 счетчиков – целых чисел, на каждый из которых выделено по пять позиций, начиная с одиннадцатой. В случае молекулы воды все эти счетчики равны нулю за исключением счетчика записей типа HETATM, расположенного в позициях 51–55, и счетчика записей типа CONECT, расположенного в позициях 61–65.

Строка 8 обозначает конец файла.

**Формат CML.** Аббревиатура CML (Chemical Markup Language), означающая «химический язык разметки», сразу вызывает ассоциации с XML, расширенным языком разметки, средством для создания, структурирования и обработки данных различной природы. Действительно, CML является языком разметки, созданным для поддержки представления и хранения физической и химической информации, в том числе о структуре молекул, химических реакциях, спектрах, кристаллографических данных и др.

Авторами формата являются Peter Murray-Rust и Henry Rzepa, чья пионерская статья об использовании XML для представления и передачи информации о химических соединениях во всемирной паутине была опубликована в 1999 году. С тех пор набор спецификаций формата CML, известный как Schema [2.7], расширился и уточнялся. Официальная (стабильная) версия, принятая в 2005 году, имеет номер 2.4, последний вариант – Schema 3.

В качестве примера ниже приведен сокращенный вариант CML файл молекулы воды.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <molecule xmlns="http://www.xml-cml.org/schema">
3   <formula concise=" H 2 O 1 "/>
4   <name convention="IUPAC">Oxidane</name>
5   <atomArray>
6     <atom id="a1" elementType="H"
7       x3="0.631087" y3="-0.026505" z3="0.474853"/>
8     <atom id="a2" elementType="O"
9       x3="0.147925" y3="0.029981" z3="-0.342190"/>
10    <atom id="a3" elementType="H"
11      x3="-0.779012" y3="-0.003476" z3="-0.132663"/>
12  </atomArray>
```

```

13 <bondArray>
14   <bond atomRefs2="a1 a2" order="1"/>
15   <bond atomRefs2="a2 a3" order="1"/>
16 </bondArray>
17 <propertyList>
18   <property title="Melting point">
19     <scalar dataType="xsd:double" errorValue="1.0"
20       units="units:celsius">0</scalar>
21   </property>
22   <property title="Boiling point">
23     <scalar dataType="xsd:double" errorValue="1.0"
24       units="units:celsius">100</scalar>
25   </property>
26 </propertyList>
27 </molecule>

```

*Строка 1* в этом файле – традиционная отсылка к XML формату.

*Строки 2* и *27* отмечают, соответственно, начало и конец описания очередной молекулы. Согласно формату XML все данные выражаются в терминах элементов, так что `molecule` это элемент CML, который представляет информацию о молекуле, в данном случае, молекуле воды. Начало и конец элемента отмечается соответствующими открывающим и закрывающим тэгами: имя элемента в угловых скобках `<...>` – начало, имя элемента в закрывающих угловых скобках `</...>` – конец. Элементы внутри этих границ считаются дочерними. Элемент может быть пустым, тогда у него нет закрывающего, и он обозначается тэгом вида `<.../>`.

И открывающий тэг, и тэг пустого элемента могут содержать атрибуты, которые называются свойствами элемента. Так, открывающий тэг элемента `molecule` содержит атрибут `xmlns`, определяющий адрес интернет-страницы, содержащий базовое пространство имен (той самой Schema).

*Строки 3* и *4* содержат информацию о химической формуле и названии соединения согласно номенклатуре системы наименований химических соединений Международного союза теоретической и прикладной химии (IUPAC).

*Строки 6–12* определяют атомы молекулы. Эти атомы сгруппированы внутри элемента `atomArray` и описываются отдельно элементами `atom`. Каждый из элементов `atom` является пустым (т. е. не имеет собственного значения), но он содержит атрибуты, определяющие

основные характеристики атома: его идентификатор, `id`, наименование элемента в химической таблице, `elementType`, и его координаты в трехмерном пространстве, `x3`, `y3` и `z3`.

*Строки 13–16* определяют связи между атомами, используя синтаксис, аналогичный использованному выше при описании отдельных атомов. Атрибутами элемента `bond` (связь) являются `atomRefs2` – список идентификаторов двух атомов, между которыми существует связь, и `order`, определяющий порядок связи (одинарная, двойная и т. д.).

*Строки 17–26* содержат элемент `propertyList` (список свойств) и его дочерние элементы – отдельные свойства, например температуру плавления (`Melting point`, *строки 18–21*) и температуру кипения (`Boiling point`, *строки 22–25*).

В заключение отметим, что в отличие от MOL/SDF и PDB, форматы, основанные на XML, являются свободными, потому что хранимые здесь данные занимают не фиксированную позицию в строке, а явно отмечаются синтаксисом тэгов. Это относится и к элементам (например, скалярный параметр, характеризующий температуру кипения воды, равен 100 градусам Цельсия), и к атрибутам (например, абсцисса первого атома водорода в молекуле равна 0.631087).

## ПРИМЕР ВЫПОЛНЕНИЯ ЗАДАНИЯ

Рассмотрим задачу визуализации молекулы этилена и на ее примере разберем основные этапы реализации проекта.

**Получение файла со структурой молекулы.** Химическая формула этилена –  $C_2H_4$ ; шаростержневая модель его молекулы, программу построения которой нужно разработать, приведена на страничке этого вещества в Википедии и ниже на рис. 19.

Для построения модели, прежде всего, необходимы данные о координатах ее атомов, т. е. файл в одном из представленных выше форматов. Для поиска такого файла воспользуемся базой химических соединений Mol-Instincts [2.12]. На момент написания данного пособия файлы всех химических соединений, представленных в таблице 2 на стр. 59, были свободно доступны в этой базе.

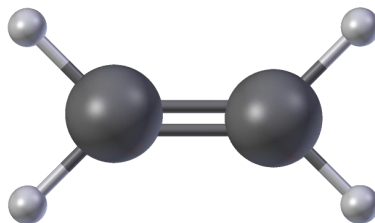


Рис. 19. Модель молекулы этилена. Источник:  
<https://ru.wikipedia.org/wiki/Этилен>

Для поиска файла структуры можно воспользоваться формой поиска на сайте [www.molinstincts.com](http://www.molinstincts.com), а можно искать с помощью привычной поисковой системы Google, введя в строку поиска следующий текст:

ethylene mol file Mol-Instincts

В этом запросе *ethylene* – название искомого вещества, *mol* – формат файла, который мы надеемся найти. Результаты поиска приведены на рис. 20, видно, что уже первые две ссылки, скорее всего, позволят получить нужные данные.

Рис. 20. Результаты поиска файла структуры молекулы

В зависимости от выбранной ссылки можно попасть на немножко разные страницы базы данных об этилене. В первом случае ссылка на загрузку файла сразу бросается в глаза:



## Structure Data File (SDF/MOL File) Description

The structure data file (SDF/MOL File) of ETHYLENE is available for download. Click the link below to start downloading.



[Download structure data file \(SDF/MOL File\)](#)



Во втором случае эта ссылка находится в середине страницы, после описания химической структуры и списка свойств, в разделе **Additional Information** (Дополнительная информация) и имеет приблизительно такой вид:



### • [Structure Data File \(SDF/MOL File\) of ETHYLENE](#)

The structure data file (SDF/MOL File) of ETHYLENE is available for [download in the SDF page of ETHYLENE](#), which provides the information about the atoms, bonds, connectivity and coordinates of ETHYLENE. The ETHYLENE structure data file can be imported to most of the cheminformatics software systems and applications.



Обратите внимания на аббревиатуру SDF/MOL. Как отмечалось выше, структура этих файлов практически идентична, если в них хранится информация не о нескольких веществах сразу, а только об одном, как в данном случае.

Загруженный файл имеет имя ETHYLENE-3D-structure-ST1002310794.sdf, для удобства дальнейшей работы переименуем его в ethylene.sdf. Открыв загруженный файл в любом текстовом редакторе (вполне можно обойтись стандартным приложением Блокнот), убеждаемся, что это действительно файл, структура которого описана выше

```

1  CT1002310794
2
3
4      6  5  0  0  0          999 v2000
5      -0.0126  1.0758  0.0080 C  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
6      0.0021  -0.0041  0.0020 H  0  0  0  0  0  0  0  0  0  0  0  0  0  0
7      0.9153  1.6285  0.0021 H  0  0  0  0  0  0  0  0  0  0  0  0  0  0
8      -1.1558  1.7153  0.0169 C  0  0  0  0  0  0  0  0  0  0  0  0  0  0
9      -1.1705  2.7952  0.0228 H  0  0  0  0  0  0  0  0  0  0  0  0  0  0
10     -2.0837  1.1627  0.0183 H  0  0  0  0  0  0  0  0  0  0  0  0  0  0
11     1  2  1  0  0  0  0
12     1  3  1  0  0  0  0
13     1  4  2  0  0  0  0
14     4  5  1  0  0  0  0
15     4  6  1  0  0  0  0
16  M  END
17  $$$

```

**Обсуждение структуры программы.** Прежде чем писать код визуализации структуры молекулы этилена на основе полученного файла, опишем кратко основные возможные классы, которые могут быть реализованы для этих целей, их атрибуты и методы. Разумеется, при выполнении проектного задания можно использовать другой набор классов, а можно обойтись вообще без них. Представленный здесь вариант реализации проектного задания является исключительно демонстрационным.

Первым классом, предлагаемым к использованию, станет класс Atom, объекты, или экземпляры, которого будут хранить информацию об обозначении химического элемента, названии и координатах атома в трехмерном пространстве, размере (радиусе) и цвете шарика, визуализирующего данный атом в шаростержневой модели. К методам объектов этого класса, кроме инициализации, следует отнести метод show отображения шарика (т. е. визуализации атома) и методы show\_label, hide\_label, соответственно включающий и выключающий изображение метки с названием химического элемента рядом с шариком.

Класс Bond будет предназначен для визуализации связей между отдельными атомами в молекуле. Инициализироваться экземпляр этого класса будет ссылками на два атома, между которыми существует данная связь и типом связи (одинарная или двойная). К атрибутам визуализации следует отнести радиус трубочки, визуализирующей связь. Метод show

будет создавать и изображать на экране эту трубочку (или несколько трубочек). Какой объект VPython использовать для отрисовки трубочек – `cylinder` или `curve` – предстоит решить разработчику программы, возможно с учетом требования к визуализации связи (резкий переход цветов или их смешивание).

Основным классом станет `Molecule`. Основными атрибутами этого класса будут два списка – список атомов, т. е. объектов класса `Atom`, и список связей – объектов класса `Bond`. Инициализация молекулы, т. е. создание ее основных атрибутов, будет осуществляться на основе чтения данных из файла. Методами экземпляров этого класса будут `show` – изображение шаростержневой модели молекулы, а также `show_labels`, `hide_labels` для отображения/скрытия меток атомов молекулы.

**Класс `Atom`.** Очевидно, что информацию о названии химического элемента и координатах атома, т. е. центра шарика, мы получим, анализируя содержимое загруженного файла. Пока открытым остается вопрос о радиусе этого шарика и его цвете. Конечно, цвет атома – понятие очень условное; анализ шаростержневой модели на рис. 17(в) показывает, что требование к цветам шариков только одно: атомы одинаковых химических элементов при визуализации должны иметь одинаковый цвет, цвета атомов различных элементов не должны совпадать. Сказанное позволяет просто определить в программе «таблицу цветов»: словарь, устанавливающий соотношение между названием химического элемента и цветом шарика, изображающим атом этого элемента, примерно так:

```
ATOM_COLORS = {'H':color.white, 'C':color.gray(0.25)}
```

Таким образом, атомы водорода будут изображаться белыми шариками, а атомы углерода – темно-серыми. Заметим, что такой набор цветов соответствует принятым на обоих рисунках: и на рис. 17, и на рис. 19. В различных программных комплексах и системах визуализации структур молекул эти цветовые схемы могут немного отличаться. Подробно с классическими и современными схемами визуализации атомов можно познакомиться в [2.9].

Ситуация с «размерами» атомов чуть сложнее. Анализ рис. 17 и рис. 19. показывает, что атомы принято изображать шариками разного размера. Давайте определим возможную схему задания радиусов этих



шариков. Самый простой вариант состоит в учете следующего факта: «при движении сверху вниз по столбцам периодической таблицы атомный радиус увеличивается, поскольку есть больше энергетических уровней и, следовательно, больше расстояние между протонами и электронами» [2.2]. Схема визуализации может быть следующей: радиус шариков атомов, пропорционален номеру ряда элемента в периодической таблице. Тогда радиус шарика для атома водорода можно условно принять за единицу, для углерода, азота и кислорода – за двойку, серы и фосфора – за тройку и т. д. Более тонкий вариант связан с приведенной в [2.2] таблицей значений измеренных опытным путём ковалентных радиусов для большого количества химических элементов. Тогда, словарь «размеры атомов» может быть задан следующим оператором:

```
АТОМ_SIZES = {'H':25, 'C':70}
```

Числа 1 в первой схеме, и 25 во второй являются некоторыми условными размерами атома водорода, используемыми исключительно с целью визуализации. Во втором случае можно учесть, что данные о координатах атомов в MOL формате задаются в ангстремах, и использовать именно ангстремы, т. е.  $10^{-10}$  м, для задания размеров. Учитывая, что величины в таблице [2.2] заданы в пикометрах, т. е. в  $10^{-12}$  м, словарь размеров можно переписать в виде:

```
АТОМ_SIZES = {'H':0.25, 'C':0.7}
```

Другим способом обеспечить наглядность модели и визуально правильное соотношение между размерами атомов и расстояниями между ними, когда шарики не перекрываются, как в модели с пространственным заполнением, и в то же время достаточно заметны, является введение масштабного множителя для размеров атомов.

Из схемы, приведенной на рис. 21, видно, что визуализация будет гарантированно удовлетворять требуемым условиям, если наибольший из всех радиусов шариков ( $r_{max}$ ) будет несколько меньше половины наименьшего из расстояний между всеми атомами ( $dist_{min}$ ), например  $r_{max} = dist_{min}/3$  или  $r_{max} = dist_{min}/4$ . Таким образом, перед отображением шариков на экране необходимо сначала вычислить масштабный множитель, например по формуле

```
scale = dist_min/r_max*0.33
```

а затем умножить радиусы всех атомов молекулы на этот коэффициент.

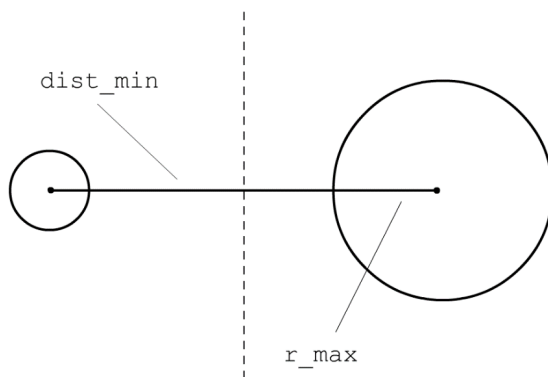


Рис. 21. Визуально удовлетворительное расположение моделей атомов

Операция вычисления масштабного множителя относится уже не к объекту класса `Atom`. Она должна выполняться для молекулы в целом, в качестве завершающего этапа чтения данных из файла.

Прежде чем переходить к разработке следующего класса, подведем краткий итог сделанного. Во фрагменте кода 7 содержится определение описанных выше констант и класса `Atom` с требуемыми функциями. Метка с названием химического элемента в данном случае располагается в точке привязки – центре сферы. Остальные ее характеристики имеют значения, установленные по умолчанию. В каждом индивидуальном варианте эти значения по умолчанию (например, размер шрифта, или наличие окантовки у метки), возможно, потребуются скорректировать, чтобы изображение названия элемента соответствовало схеме, приведенной в Табл. 3.

Фрагмент кода 7. Класс для работы с отдельным атомом

```
1 from vpython import *
2
3 #по данным сайта ru.wikipedia.org/wiki/Радиус_атома
4 ATOM_SIZES = {'H':0.25, 'C':0.70}
5 #условные цвета атомов:
6 ATOM_COLORS = {'H':color.white, 'C':color.gray(0.25)}
7
```

```

8  class Atom:
9
10     def __init__(self, name, pos):
11         '''
12         Экземпляры класса инициализируются
13         названием химического элемента (name, str)
14         и вектором координат в трехмерном пространстве
15         (pos, vector)
16         '''
17         self.name = name
18         self.pos = pos
19         # радиус атома определяется по названию
20         # из таблицы размеров
21         # (на этапе визуализации возможна корректировка)
22         self.radius = ATOM_SIZES[name]
23         # цвет атома определяется по названию
24         # из таблицы цветов
25         self.color = ATOM_COLORS[name]
26
27     def show(self):
28         '''
29         Показать атом - значит создать шарик
30         нужного размера и цвета и расположить
31         рядом с ним невидимую пока метку
32         с названием химического элемента
33         '''
34         self.ball = sphere(pos=self.pos,
35                             radius=self.radius,
36                             color=self.color)
37         # в соответствии с заданием рядом с шариком
38         # разместим метку
39         self.lbl = label(pos=self.pos,
40                          text=self.name, visible=False)
41     def show_label(self):
42         self.lbl.visible = True
43
44     def hide_label(self):
45         self.lbl.visible = False

```

**Класс Bond.** Основным параметром каждой из визуализируемых связей является ее положение в пространстве. Будем считать, что связь соединяет центры атомов, т. е. для инициализации связи потребуется

информация о координатах этих центров. Для простоты в качестве аргументов конструктора связи будем использовать два экземпляра класса `Atom`. Третьим аргументом станет тип связи: 1 – для одинарной и 2 – для двойной. Других типов связи в файлах проектного задания встречаться не будет.

Важной характеристикой визуализации связи является радиус трубочки, которая изображает связь. На рис. 22 приведены несколько вариантов соотношения между радиусом трубочки и радиусом шарика, изображающего атом. Видно, что диапазон отношения радиуса трубочки к радиусу шарика от 0.1 до 0.25 является вполне приемлемым. В приведенном фрагменте кода использован коэффициент 0.25. Поскольку радиус всех трубочек-связей должен быть одинаковым, этот коэффициент должен быть применен к наименьшему из всех возможных радиусов атомов. Выше мы ввели словарь `ATOM_SIZES` с информацией о размерах атомов, поэтому радиус трубочки вычисляется как

$$\min(\text{ATOM\_SIZES.values}()) / 4$$

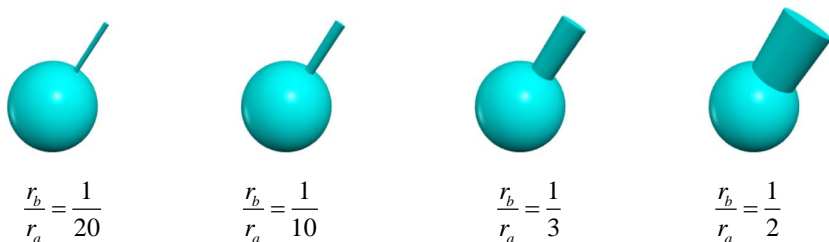


Рис. 22. Варианты соотношения радиуса трубочки, изображающей связь ( $r_b$ ), и радиуса сферы, изображающей атом ( $r_a$ ):

Как и в случае атомов, данный размер является условным и должен быть позднее (на этапе создания молекулы) масштабирован с тем же коэффициентом, с которым будут масштабированы шарики, изображающие атомы.

Представленный ниже фрагмент кода 8 реализует только один из возможных вариантов визуализации: одинарная связь, резкий переход цветов. Для изображения связи используем две трубочки – каждая из них имеет длину, равную половине расстояния между атомами, а цвет совпадает с цветом того атома, к которому она «привязана»: точка привязки первой трубочки, параметр `pos`, совпадает с центром первого

атома, а ее ось равна половине вектора, идущего из центра первого атома в центр второго. Для второй трубочки ситуация аналогична, только привязана она ко второму атому.

Если требуется мягкий переход цветов, данный вариант реализации не подходит. В этом случае вместо объекта `cylinder` необходимо использовать объект `curve`. Первую точку этого объекта следует поместить в центр первого атома, вторую – в центр второго. Если при этом цвета точек будут совпадать с цветами атомов, то требуемый мягкий переход от одного цвета к другому будет реализован автоматически.

Фрагмент кода 8. Класс для работы со связями

```
1 class Bond:
2
3     def __init__(self, atom1, atom2, bond_type):
4         '''
5         Экземпляры класса "связь" инициализируются
6         теми атомами, между которыми существует
7         эта связь (atom1 и atom2, типа Atom),
8         типом связи (bond_type, integer).
9         Кроме того, задается "базовый"
10        радиус цилиндра, визуализирующего связь.
11        '''
12        self.atom1 = atom1
13        self.atom2 = atom2
14        self.bond_type = bond_type
15        self.radius = min(ATOM_SIZES.values())/4
16
17    def show(self):
18        '''
19        Показать связь - значит создать
20        набор цилиндров нужного размера и цвета.
21        В зависимости от типа связи количество
22        таких цилиндров будет разным.
23        '''
24        a1 = self.atom1
25        a2 = self.atom2
26        if self.bond_type==1:
27            tubel = cylinder(pos=a1.pos,
28                            axis=(a2.pos-a1.pos)/2,
29                            radius=self.radius,
```

```

30         color=a1.color)
31     tube2 = cylinder(pos=a2.pos,
32                     axis=(a1.pos-a2.pos)/2,
33                     radius=self.radius,
34                     color=a2.color)
35     elif self.bond_type==2:
36         pass
37         # для самостоятельной разработки
38
39     def length(self):
40         return mag(self.atom1.pos-self.atom2.pos)

```

Разработку и реализацию схемы построения двойной связи (двух трубочек) мы оставляем обучающимся. В качестве небольшой подсказки приведем возможную схему «раздвоения». Вместо трубочки, реализующей одинарную связь, необходимо нарисовать две. Их центры следует расположить в плоскости, перпендикулярной оси исходной трубочки, сдвинув от центра к краю на расстояние  $s$ , которое можно выбрать из следующих соображений (см. рис. 23):

1.  $s > r_b$ , где  $r_b$  – радиус трубочки. Выполнение этого соотношения будет гарантировать, что новые трубочки не будут пересекаться

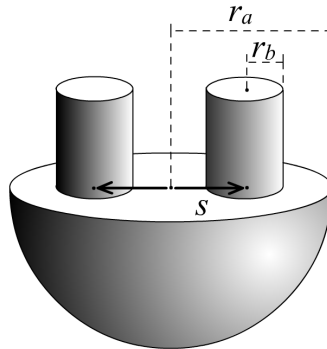


Рис. 23. Возможная схема визуализации двойной связи

2.  $s + r_b < r_a^{min}$ , где  $r_a^{min}$  – наименьший из всех возможных радиусов атомов. Это требование обеспечит расположение нижнего основания трубочки внутри шарика атома.

Учитывая, что в нашем случае  $r_a^{min} = 4r_b$ , значение величины  $s$  следует выбирать из интервала  $(r_b, 3r_b)$ .

Единичный вектор, перпендикулярный оси трубочки, построить достаточно легко из следующих соображений. Пусть вектор  $\vec{d}$  соединяет центры атомов  $a_1$  и  $a_2$ , связь между которыми требуется изобразить, т. е.  $d = a2.pos - a1.pos$ . Предположим, что вектор  $\vec{d}$  имеет компоненты  $(d_x, d_y, d_z)$ . Тогда очевидно, что вектор  $\vec{e}$  с компонентами  $(-d_y, d_x, 0)$ , будет ортогонален вектору  $\vec{d}$ , так как  $\vec{d} \cdot \vec{e} = 0$ . Превращение вектора  $\vec{e}$  в единичный реализуется с использованием функции `norm` пакета `VPython`:

```
e = norm(vec(-d.y, d.x, 0))
```

Метод `length`, вычисляющий длину связи, т. е. расстояние между атомами, которые эта связь соединяет, введен исключительно с одной целью – для нахождения минимального расстояния между атомами в ходе вычисления коэффициента масштабирования радиусов.

**Класс `Molecule`.** Объект этого класса, код создания которого приведен во фрагменте 9, представляет собой два списка – список атомов, составляющих молекулу, и список связей между этими атомами. Поэтому процесс отображения молекулы (метод `show()`, строки 56–60) сводится просто к последовательному отображению составляющих ее атомов и связей. Аналогично, отображение или скрытие меток оставляющих ее атомов (методы `show_labels()` и `hide_labels()`, строки 62–68) сводится к вызову соответствующих методов каждого атома.

Основным же методом класса является его конструктор, считывающий данные о структуре молекулы из файла и создающий упомянутые выше списки атомов и связей. Остановимся на реализации этого конструктора чуть подробнее.

Фрагмент кода 9. Класс для создания и визуализации молекулы

```
1 class Molecule:
2
3     def __init__(self, molfile_name):
4         '''
5         Экземпляр класса создается
```

```

6     на основе информации из файла
7     (molfile_name, str)
8     '''
9     with open(molfile_name) as f:
10        for i in range(3): # пропускаем
11            f.readline() # первые три строки
12            # из четвертой строки
13            # получаем информацию
14            # о числе атомов и о количестве связей
15            s = f.readline()
16            self.n_atoms = int(s[:3])
17            self.n_bonds = int(s[3:6])
18            # создаем список атомов
19            self.atoms=[]
20            for i in range(self.n_atoms):
21                # обрабатываем строки с информацией
22                # о названии химического элемента
23                # и координатах атомов
24                s = f.readline()
25                el_name = s[31:34].rstrip()
26                x = float(s[:10])
27                y = float(s[10:19])
28                z = float(s[20:29])
29                self.atoms.append(
30                    Atom(el_name, vec(x, y, z)))
31
32            # создаем список связей
33            self.bonds=[]
34            for i in range(self.n_bonds):
35                # обрабатываем строки
36                # с информацией о связях
37                s = f.readline()
38                n1 = int(s[:3]) - 1
39                n2 = int(s[3:6]) - 1
40                bond_type = int(s[6:9])
41                self.bonds.append(Bond(self.atoms[n1],
42                                       self.atoms[n2],
43                                       bond_type))
44            # работа с файлом завершена
45
46            # этап корректировки радиусов
47            r_max = max(ATOM_SIZES.values())
48            dist_min = min(b.length() for b in self.bonds)

```



```

49     scale = dist_min/r_max*0.33
50     for a in self.atoms:
51         a.radius *= scale
52     bond_radius = min(ATOM_SIZES.values())*scale/4
53     for b in self.bonds:
54         b.radius = bond_radius
55
56     def show(self):
57         for a in self.atoms:
58             a.show()
59         for b in self.bonds:
60             b.show()
61
62     def show_labels(self):
63         for a in self.atoms:
64             a.show_label()
65
66     def hide_labels(self):
67         for a in self.atoms:
68             a.hide_label()

```

Работа с файлом, имя которого передается конструктору в параметре `molfile_name`, осуществляется в строках 9–43 приведенного кода. В строке 9 файл открывается на чтение и связывается с переменной `f`. В соответствии со структурой MOL/SDF файла (см. стр. 65) важная для визуализации информация начинается с четвертой строчки, поэтому первые три просто пропускаем. Из четвертой строки получаем информацию о количестве атомов и связей; введенные переменные `n_atoms` и `n_bonds` становятся атрибутами класса. В приведенном фрагменте показано их вычисление на основе информации о фиксированном типе файла и тех позициях в строке, которые они занимают (строки 16–17 кода).

Знание количества атомов и связей позволяет организовать два цикла чтения информации из файла. В первом цикле (строки 20–30) происходит чтение и анализ строк с информацией о названии очередного атома (`el_name`) и его координатах (`x`, `y`, `z`). На этой основе создается очередной экземпляр класса `Atom` и добавляется в список `atoms` молекулы (строки 29–30). Аналогично этому, во втором цикле (строки 34–43) читаются строки с информацией о связях, из которых в соответствии с

фиксированной структурой файла извлекается интересующая нас информация о типе связи (`bond_type`) и номерах атомов (`n1`, `n2`), которые соединяет эта связь (поскольку списки в Питоне индексируются с нуля, то из прочитанного номера вычитается единица). На каждом шаге цикла список `bonds` объекта класса `Molecule` пополняется очередным экземпляром класса `Bond`.

В строках 47–54 осуществляется корректировка радиусов шариков, представляющих атомы и трубочек, представляющих связи. В соответствии с описанным выше подходом радиусы всех шариков умножаются на масштабный коэффициент, пропорциональный отношению минимального расстояния между атомами к радиусу максимального из них. Радиус всех трубочек – связей – принимается одинаковым и равным одной четвертой радиуса наименьшего из шариков.

## ЛИТЕРАТУРА К ГЛАВЕ 2

- 2.1. Пронкин П. Г., Сорокина О. Н., Ботнарь Ю. В. Эволюция средств и методов информатики в практической и фундаментальной химии // Прикладная информатика. – 2009, № 1(19). – С. 69–74.
- 2.2. Радиус атома [Электронный ресурс] – URL: [https://ru.wikipedia.org/wiki/Радиус\\_атома](https://ru.wikipedia.org/wiki/Радиус_атома) (дата обращения 01.02.2021)
- 2.3. Соловьев М. Е., Соловьев М. М. Компьютерная химия – М.: СОЛОН-Пресс, 2005. – 536 с.
- 2.4. Хемоинформатика/Большая российская энциклопедия [Электронный ресурс] – URL: <https://bigenc.ru/chemistry/text/4664214> (дата обращения 01.02.2021)
- 2.5. CAS. The World's Largest Collection of Chemistry Insights [Электронный ресурс] – URL: <https://www.cas.org/about/cas-content> (дата обращения 01.02.2021)
- 2.6. ChEMBL Database [Электронный ресурс] – URL: <https://www.ebi.ac.uk/chembl> (дата обращения 01.02.2021)
- 2.7. Chemical Markup Language [Электронный ресурс] – URL: <http://www.xml-cml.org> (дата обращения 01.02.2021)
- 2.8. ChemSpider: Search and share chemistry [Электронный ресурс] – URL: <http://www.chemspider.com> (дата обращения 01.02.2021)

- 2.9. CPK\_coloring [Электронный ресурс] – URL: [https://en.wikipedia.org/wiki/CPK\\_coloring](https://en.wikipedia.org/wiki/CPK_coloring) (дата обращения 01.02.2021)
- 2.10. Field M.J. A practical introduction to the simulation of molecular systems. Second Ed. – New York: Cambridge University Press, 2007. – 339 p.
- 2.11. Kubara K. Introduction to Cheminformatics: Common Challenges, Libraries, and Datasets for Cheminformatics [Электронный ресурс] – URL: <https://towardsdatascience.com/introduction-to-cheminformatics-7241de2fe5a8> (дата обращения 01.02.2021)
- 2.12. Mol-Instincts: First Chemical Database Based on Quantum Mechanics [Электронный ресурс] – URL: <https://www.molinstincts.com> (дата обращения 01.02.2021)
- 2.13. PubChem [Электронный ресурс] – URL: <https://pubchem.ncbi.nlm.nih.gov> (дата обращения 01.02.2021)

## ГЛАВА 3. МОДЕЛИРОВАНИЕ ДВИЖЕНИЯ МЕХАНИЧЕСКИХ СИСТЕМ

В данной части учебного пособия приведены основные алгоритмы, которые потребуются в дальнейшем при выполнении проектных заданий, связанных с визуализацией движения механических систем на основе численного решения дифференциальных уравнений. Представленные примеры показывают некоторые особенности применения, возможности и ограничения описанных алгоритмов. Кроме того, они также служат в какой-то степени образцами выполнения проектных заданий, знакомя с возможностями научной графики пакета VPython и демонстрируя его использование для создания трехмерных иллюстративных анимаций.

### ЧИСЛЕННЫЕ МЕТОДЫ РЕШЕНИЯ ЗАДАЧИ КОШИ

**Постановка задачи.** Обыкновенные дифференциальные уравнения и их системы встречаются в различных задачах математического моделирования. Они используются при описании колебательных процессов, роста популяций, радиоактивного распада, кинетики химических реакций, динамики взаимодействующих материальных тел и т. д. Задачи динамики – один из важнейших классов физической теории; с точки зрения математики они сводятся к задачам Коши, иначе называемым *начальными* задачами (initial value problems, IVP): по известным в данный момент значениям функции (и, в зависимости от порядка уравнения, нескольких ее производных) требуется построить решение на некотором интервале времени, начиная с данного момента. В большинстве случаев возникающие задачи не имеют аналитического решения и для их исследования активно используются численные методы.

Данный раздел ни в коей мере не претендует на какую-либо степень полноты или строгости изложения материала темы, вынесенной в заголовок. Численным методам интегрирования обыкновенных дифференциальных уравнений посвящены обширные разделы в учебниках, монографии, интернет-сайты, онлайн-курсы. Существует большое количество программных комплексов, пакетов и библиотек, предназначенных для реализации таких задач. В научных журналах

регулярно выходят новые статьи, содержащие описание эффективных в той или иной степени модификаций существующих методов, а также оригинальные численные схемы, ориентированные на конкретные классы уравнений или специфическое компьютерное оборудование (например, высокопроизводительные многопроцессорные системы). Даже краткий обзор существующей литературы по этой теме мог бы занять целый том: ведь по запросу «Численные методы решения задач Коши» Google выдает более 260 тысяч ссылок.

Цель данного раздела состоит в том, чтобы в сжатом виде перечислить и описать лишь те алгоритмы, которые рекомендованы для использования при реализации проектов, описанных в настоящем пособии. Кроме того, поскольку основным средством реализации алгоритмов предполагается язык Питон, то отдельно описана функция пакета `scipy`, позволяющая эффективно численно решать задачи Коши на этом языке.

Заметим еще, что большинство учебников по численным методам или совсем не содержит примеров программной реализации описанных схем, или использует в качестве примеров такие языки высокого уровня, которые традиционно считаются ориентированными на эффективные вычисления (C/C++, FORTRAN). Языку Питон в этом смысле повезло чуть меньше, хотя в последние годы появился ряд учебников, например [3.2, 3.6, 3.8, 3.9, 3.10, 3.11], в которых для иллюстрации численных методов используются программы на Питоне. Особо хочется упомянуть русскоязычный «Вычислительный практикум» [3.2], а также тот факт, что pdf-файл работы [3.11] доступен на сайте издательства<sup>12</sup> в режиме Open Access, а электронный вариант учебника [3.9] с 2020 года официально находится в открытом доступе даже в интерактивном формате `ipython notebook`<sup>13</sup>.

Что касается собственно постановки задачи, то она выглядит следующим образом. Дано дифференциальное уравнение вида

$$y^{(n)}(t) = F\left(t, y(t), y'(t), \dots, y^{(n-1)}(t)\right). \quad (1)$$

Требуется найти его частное решение на отрезке  $[t_0, T]$ , удовлетворяющее условиям

---

<sup>12</sup> <https://www.springer.com/gp/book/9783030168766>

<sup>13</sup> <https://pythonnumericalmethods.berkeley.edu/>

$$y(t_0) = y_0, y'(t_0) = y_1, \dots, y^{(n-1)}(t_0) = y_{n-1}. \quad (2)$$

В (1), (2) обозначение  $y^{(k)}(t)$  использовано для  $k$ -й производной функции  $y(t)$ :

$$y^{(k)}(t) \equiv \frac{d^k y}{dt^k}.$$

Вопросы существования такого решения выходят за рамки данного обсуждения: далее везде предполагается, что решение существует и имеет нужную степень гладкости.

Другим часто используемым вариантом формулировки задачи (1), (2) является ее запись в терминах системы уравнений первого порядка. Вводя обозначения

$$y(t) = y_0(t), y'(t) = y_1(t), \dots, y^{(n-1)}(t) = y_{n-1}(t),$$

вместо (1) и (2) получаем соответственно

$$\begin{cases} y_0'(t) = y_1(t), \\ y_1'(t) = y_2(t), \\ \dots \\ y_{n-2}'(t) = y_{n-1}(t), \\ y_{n-1}'(t) = F(t, y_0(t), y_1(t), \dots, y_{n-1}(t)) \end{cases} \quad (3)$$

и

$$y_0(t_0) = y_0, y_1(t_0) = y_1, \dots, y_{n-1}(t_0) = y_{n-1}. \quad (4)$$

Ну и, наконец, обе формы задачи Коши часто записываются в векторном формате. При этом опять возможны варианты. Так, функцию  $y(t)$  в (1) можно считать вектор-функцией. Строго говоря, при этом (1) становится не уравнением, а системой уравнений, но постановка задачи нахождения ее решения по заданным начальным данным сохраняется. Например, для описания движения материальной частицы массы  $m$  под действием силы  $\vec{F}$  необходимо решить дифференциальное уравнение

$$m \frac{d^2 \vec{r}}{dt^2} = \vec{F},$$

где  $\vec{r}(t) = (x(t), y(t), z(t))$  – радиус-вектор частицы, а в начальный момент известно значение этого вектора  $\vec{r}(t_0) = \vec{r}_0$  и его производной по времени (скорости частицы)  $\vec{r}'(t_0) = \vec{v}_0$ .

**Метод Эйлера и его модификации.** Необходимо отметить, что несмотря на достаточно долгую историю вопроса, проблемы с правильным «именованием» методов, используемых для численного интегрирования задачи Коши, до сих пор существуют. Некоторые методы в разных источниках, особенно относящихся к разным областям знания, называются именами разных исследователей. В качестве примера можно привести статью в англоязычной Википедии<sup>14</sup>, в которой метод Эйлера-Кромера называется также симплектическим методом Эйлера, полу-явным методом Эйлера и методом Ньютона-Штормера-Верле. Абсолютные единомышленники авторы учебников, монографий и интернет-статей (как в России, так и за рубежом) проявляют, пожалуй, только по отношению к самому методу Эйлера, с описания которого начинаются практически все разделы курсов, связанные с численным решением дифференциальных уравнений.

Поскольку в настоящем пособии нас будут интересовать задачи Коши, связанные с динамикой материальных точек, при описании методов ограничимся случаем одномерного движения материальной точки, положение которой определяется ее ординатой  $y$ . Уравнение второго закона Ньютона

$$m \frac{d^2y}{dt^2} = F$$

перепишем в виде системы двух дифференциальных уравнений первого порядка:

$$\frac{dv}{dt} = a(t, y(t), v(t)), \quad \frac{dy}{dt} = v(t), \quad (5)$$

где  $a = F/m$ . В начальный момент времени  $t_0$  считаются известными значения координаты точки и ее скорости

$$y(t_0) = y_0, \quad v(t_0) = v_0. \quad (6)$$

Численно решить задачу Коши (5), (6) на отрезке  $[t_0, T]$  – значит найти приближенные значения функций  $y_k = y(t_k)$  и  $v_k = v(t_k)$  в некоторые заданные моменты времени  $t_k \in [t_0, T]$ ,  $k = 1, 2, \dots, n$ . В качестве таких моментов обычно выбираются значения  $t_0 + \Delta t, t_0 + 2\Delta t, \dots, t_0 + n\Delta t = T$ , т. е.  $t_k = t_0 + k\Delta t$ , где  $\Delta t$  – шаг интегрирования, который задается достаточно малым, чтобы обеспечить устойчивость проведения расчетов. В

<sup>14</sup> [https://en.wikipedia.org/wiki/Semi-implicit\\_Euler\\_method](https://en.wikipedia.org/wiki/Semi-implicit_Euler_method)

случае консервативных систем еще одним требованием к величине  $\Delta t$  может быть сохранение полной энергии системы.

Таким образом, основная задача численного интегрирования состоит в определении величин  $y_{k+1}$  и  $v_{k+1}$  по известным значениям  $y_k$ ,  $v_k$  и  $\Delta t$ . Суть многих вычислительных алгоритмов может быть прояснена путем разложения величин  $v_{k+1} = v(t_k + \Delta t)$  и  $y_{k+1} = y(t_k + \Delta t)$  в ряд Тейлора:

$$\begin{aligned}v_{k+1} &= v_k + a_k \Delta t + O(\Delta t^2), \\y_{k+1} &= y_k + v_k \Delta t + \frac{1}{2} a_k \Delta t^2 + O(\Delta t^3).\end{aligned}\tag{7}$$

Метод Эйлера численного интегрирования состоит в отбрасывании членов второго и более высоких порядков в соотношениях (7):

$$\begin{aligned}v_{k+1} &= v_k + a_k \Delta t, \\y_{k+1} &= y_k + v_k \Delta t.\end{aligned}\tag{8}$$

Порядок отброшенных членов определяет локальную ошибку усечения метода, т. е. ошибку на одном шаге интегрирования; таким образом, локальная ошибка метода Эйлера имеет порядок  $\Delta t^2$ . Глобальная же ошибка этого метода, т. е. ошибка, накапливающаяся при интегрировании по всему временному отрезку, будет пропорциональна  $\Delta t$ . Грубо это оценку можно получить, просто просуммировав все локальные ошибки на каждом шаге интегрирования, количество которых пропорционально  $(\Delta t)^{-1}$ .

Порядок величины  $\Delta t$  в оценке глобальной ошибки определяет порядок метода: метод Эйлера поэтому является методом первого порядка. К его характеристикам следует также отнести то, что он является *явным* и *самостартующим*, то есть, зная  $y(t_0)$  и  $v(t_0)$  по формулам (8) без дополнительных предварительных расчетов можно последовательно вычислить  $y(t_1)$ ,  $y(t_2)$ , и т. д.

В чистом виде метод Эйлера очень редко используется на практике. Это связано не только с невысоким порядком точности, но и с проблемами вычислительной неустойчивости, а также нефизичностью его результатов в ряде задач математического моделирования, например с отсутствием сохранения энергии колебательной системы.

Как раз для решения последней упомянутой проблемы служит модификация метода (8), которую в данном пособии мы будем называть



**методом Эйлера-Кромера** по имени автора работы [3.5], где эта модификация была достаточно подробно изучена. Там для классического варианта метода Эйлера предложено альтернативное название FPA (first point approximation, аппроксимация по первой точке), поскольку для вычисления координаты точки  $y_{k+1}$  используется значение скорости  $v_k$  в начальной точке интервала интегрирования. В качестве альтернативного варианта А. Кромер предложил использовать для вычисления  $y_{k+1}$  значение скорости в конечной точке интервала:

$$\begin{aligned} v_{k+1} &= v_k + a_k \Delta t, \\ y_{k+1} &= y_k + v_{k+1} \Delta t. \end{aligned} \tag{9}$$

Как и метод Эйлера, данный метод имеет первый порядок точности, однако, он попадает в класс так называемых симплектических интеграторов или геометрических методов [3.1, раздел 4.6], которые учитывают свойства дифференциальных уравнений, точнее, их решений. Например, для гамильтоновых систем (а уравнения движения тела, осциллирующего на пружинке, или системы тел, перемещающихся под действием гравитационных сил, относятся к этому классу) численные решения с соответствующей порядку метода точностью сохраняют энергию системы.

В своей работе [3.5] Алан Кромер приводит следующую легенду появления этой модификации. По его словам, она была «открыта» случайно Абби Аспел, старшеклассницей Newton North School, которая занималась численным решением задачи Кеплера. Школьница совершенно правильно запрограммировала классический метод Эйлера. Одномерный вариант ее программы (на ФОРТРАНе) имел примерно такой вид:

```

100 F = (function of X)
110 X = X + V*D
120 V = V + F*D
130 PRINT X, V
140 GOTO 100

```

Эта программа нормально работала для половины орбиты (примерно 14 итераций), а потом численное решение начало расходиться, так что построить вторую половину орбиты уже не удалось. Эбби решила, что она сделала ошибку, и на всякий случай попробовала поменять местами строки 110 и 120. Программа стала выглядеть так:

```

100 F = (function of X)
110 V = V + F*D
120 X = X + V*D
130 PRINT X, V
140 GOTO 100

```

На первый взгляд кажется невероятным, но новый вариант программы построил замкнутую орбиту за 28 итераций и продолжил успешно работать дальше. На самом деле, такое изменение порядка вычисления координаты и скорости в программе как раз и соответствует замене  $v_k$  на  $v_{k+1}$  в уравнениях (8) и переходу к вычислительной схеме (9). Эта легенда привела к тому, что в ряде публикаций данный метод называют методом Эйлера-Аспел-Кромера или даже методом Эйлера-Аспел.

По мнению А. Кромера, незначительное отличие в методах (8) и (9), сводящееся на практике лишь к изменению порядка двух строчек кода, привело к тому, что многие исследователи реально используют именно «метод последней точки», думая, что запрограммировали классический метод Эйлера. А поскольку эта модификация во многих случаях дает очень хорошие результаты, у них даже нет повода подозревать, что они где-то допустили ошибку (или сделали открытие 😊).

Еще одной модификацией метода Эйлера является так называемый **метод аппроксимации по срединной точке**<sup>15</sup>. Для вычисления координаты тела он использует не начальное или конечное значение скорости на отрезке, а их среднее значение:

$$\begin{aligned}
 v_{k+1} &= v_k + a_k \Delta t \\
 y_{k+1} &= y_k + \frac{1}{2}(v_k + v_{k+1})\Delta t
 \end{aligned}
 \tag{10}$$

Если подставить во второе равенство в (10) выражение для  $v_{k+1}$  из первого, то получим следующую схему:

$$\begin{aligned}
 v_{k+1} &= v_k + a_k \Delta t, \\
 y_{k+1} &= y_k + v_k \Delta t + \frac{1}{2} a_k \Delta t^2.
 \end{aligned}
 \tag{11}$$

---

<sup>15</sup> Название и этого, и многих следующих за ним методов уже не являются общепринятыми. Тем не менее при выполнении проектных заданий следует выбирать методы в соответствии с названиями, приведенными в этом пособии.

Из (11) видно, что метод аппроксимации по срединной точке обеспечивает второй порядок точности для перемещения и первый порядок для скорости. Однако существенных практических преимуществ по сравнению с базовым методом Эйлера он не имеет [3.5].

**Методы второго порядка.** Идея вычисления скорости в середине интервала используется еще в одном достаточно распространенном методе, который будем здесь называть **методом Эйлера-Ричардсона**. Второе часто встречающееся название этого метода – метод срединной точки (Midpoint method). Метод Эйлера-Ричардсона обычно используют, когда силы в уравнении движения зависят от скорости, в остальных случаях его эффективность аналогична предыдущим описанным методам. Алгоритм метода состоит из двух шагов. Сначала методом Эйлера вычисляются положение  $y_{\text{mid}}$  и скорость  $v_{\text{mid}}$  в срединной точке временного отрезка  $t_{\text{mid}} = t_k + \Delta t/2$ . Затем вычисляется сила  $F(y_{\text{mid}}, v_{\text{mid}}, t_{\text{mid}})$ , а по ней находится ускорение  $a_{\text{mid}}$  в момент времени  $t_{\text{mid}}$ . Новое положение  $y_{k+1}$  и скорость  $v_{k+1}$  в момент времени  $t_{k+1}$  находятся на основе  $v_{\text{mid}}$  и  $a_{\text{mid}}$ . Таким образом, схему метода Эйлера-Ричардсона можно записать в виде следующих шагов:

$$\begin{aligned}
 a_k &= \frac{1}{m} F(y_k, v_k, t_k), \\
 v_{\text{mid}} &= v_k + \frac{1}{2} a_k \Delta t, \\
 y_{\text{mid}} &= y_k + \frac{1}{2} v_k \Delta t, \\
 a_{\text{mid}} &= \frac{1}{m} F\left(y_{\text{mid}}, v_{\text{mid}}, t_k + \frac{1}{2} \Delta t\right), \\
 v_{k+1} &= v_k + a_{\text{mid}} \Delta t, \\
 y_{k+1} &= y_k + v_{\text{mid}} \Delta t.
 \end{aligned} \tag{12}$$

На каждом шаге метода Эйлера-Ричардсона выполняется в два раза больше вычислений, чем на каждом шаге предыдущих вариантов метода Эйлера. Тем не менее, многие исследователи отмечают его более высокое быстродействие, поскольку, являясь методом второго порядка, для широкого круга задач он позволяет достигать требуемой точности при существенно больших значениях шага по времени.

Метод Эйлера-Ричардсона относится к одному из вариантов семейства методов Рунге-Кутты второго порядка. Другим часто используемым вариантом из этого семейства является **метод Хойна** (Heun's method), идея которого в чем-то похожа: для повышения точности аппроксимации вместо производной в начале отрезка используется некоторое промежуточное значение, на этот раз – среднее арифметическое двух величин: производной в начале отрезка и ее приближенного значения на конце отрезка интегрирования, полученного методом Эйлера. Алгоритмическая вычислительная схема этого метода применительно к системе (5) принимает вид:

$$\begin{aligned}
 a_k &= \frac{1}{m} F(y_k, v_k, t_k) \\
 y_{k+1}^* &= y_k + v_k \Delta t, \\
 v_{k+1}^* &= v_k + a_k \Delta t, \\
 a_{k+1}^* &= \frac{1}{m} F(y_{k+1}^*, v_{k+1}^*, t_{k+1}) \\
 y_{k+1} &= y_k + \frac{v_k + v_{k+1}^*}{2} \Delta t, \\
 v_{k+1} &= v_k + \frac{a_k + a_{k+1}^*}{2} \Delta t.
 \end{aligned} \tag{13}$$

Среди методов второго порядка очень распространенным, особенно при численном решении уравнений молекулярной динамики, является **метод Верле**<sup>16</sup>, использованный автором в работе [3.12] для расчета движения системы из 864 частиц, взаимодействие которых описывалось потенциалом Леннарда-Джонса. Чтобы получить его расчетные формулы, представим выражение для  $y_{k-1}$  в виде, аналогичном второй формуле в (7):

$$y_{k-1} = y_k - v_k \Delta t + \frac{1}{2} a_k \Delta t^2 + O(\Delta t^3).$$

Складывая данное представление для  $y_{k-1}$  с выражением для  $y_{k+1}$  из (7) и выражая из полученного равенства  $y_{k+1}$ , находим

$$y_{k+1} = 2y_k - y_{k-1} + a_k \Delta t^2 + O(\Delta t^4).$$

---

<sup>16</sup> В учебниках по методам вычислений данный метод известен как явный метод центральных разностей.

Если же вычесть эти выражения, то придем к соотношению для скорости:

$$v_k = \frac{y_{k+1} - y_{k-1}}{\Delta t} + O(\Delta t^3).$$

Таким образом, приходим к *оригинальному* алгоритму Верле:

$$\begin{aligned} y_{k+1} &= 2y_k - y_{k-1} + a_k \Delta t^2, \\ v_k &= \frac{y_{k+1} - y_{k-1}}{\Delta t}. \end{aligned}$$

Именно этот алгоритм был реализован в движке *fix*, использованном при разработке достаточно известной в свое время компьютерной игры «Hitman: Codename 47» [3.7].

Локальная ошибка определения координат методом Верле имеет четвертый порядок, а скоростей – второй порядок. Однако глобальная ошибка в обоих случаях одинакова и имеет второй порядок. Заметим, что расчетная схема этого метода вообще не использует значения скоростей для вычисления координат.

Алгоритм Верле не является самостартующим, поэтому для нахождения первых значений необходимо использовать какие-то дополнительные соображения или вычислительные схемы. Весьма распространенной является вычисление первого приближения по формуле, погрешность которой имеет третий порядок:

$$y_1 = y_0 + v_0 \Delta t + \frac{1}{2} a_0 \Delta t^2.$$

где  $a_0 = F(y_0, v_0, t_0)/m$ .

Еще одна проблема его реализации связана с тем, что для вычисления нового значения скорости требуется вычитать величины, близкие по значению, а такая операция, как известно, может приводить к существенным ошибкам округления.

Если ускорение в уравнениях (5) движения частицы не зависит от скорости, то обычно используют другую форму алгоритма Верле, т. н. **алгоритм Верле в скоростях** (*velocity Verlet*). Его вычислительная схема, математически эквивалентная основному варианту, записывается следующим образом:

$$\begin{aligned}
 y_{k+1} &= y_k + v_k \Delta t + \frac{1}{2} a_k \Delta t^2, \\
 a_{k+1} &= a(t_k + \Delta t, y_{k+1}), \\
 v_{k+1} &= v_k + \frac{1}{2} (a_k + a_{k+1}) \Delta t.
 \end{aligned}
 \tag{14}$$

Преимуществом этого варианта, который получается из предыдущего простыми алгебраическими преобразованиями, является то, что он самостартующий, и формула для вычисления скоростей не содержит разности близких по значению величин.

Еще одним вариантом интегрирования уравнений, правая часть которых не зависит от скорости, является **метод с перешагиванием** [3.4], или *leapfrog method*. Точное его авторство установить не удалось; известно только, что именно он был использован в Фейнмановских лекциях по физике (том I, гл. 9, § 6) для численного определения положения грузика, подвешенного на пружинке. Идея метода опять-таки связана с заменой производной на одном из концов отрезка (как в методах Эйлера и Эйлера-Кромера) ее значением в середине отрезка. Первый шаг интегрирования уравнений (5) с заданными начальными условиями (6) имеет вид

$$y_1 = y_0 + v_{1/2} \Delta t,$$

где использовано обозначение

$$v_{1/2} = v(t_0 + \Delta t/2).$$

Предположив далее, что величина  $v_{1/2}$  известна, значение скорости в момент времени  $t_0 + 3\Delta t/2$  найдем, также используя значение производной в средней точке отрезка интегрирования:

$$v_{3/2} = v_{1/2} + a(t + \Delta t, y_1) \Delta t.$$

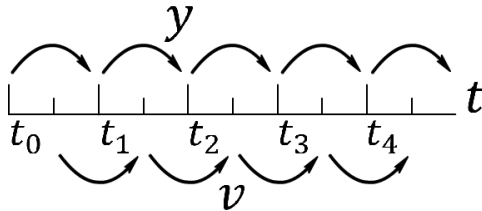


Рис. 24. Схема интегрирования с перешагиванием

Продолжая процесс, приходим к схеме с перешагиванием (см. рис. 24), когда координата и скорость вычисляются в разные моменты времени.

Итерационная схема метода задается соотношениями:

$$\begin{aligned} y_{k+1} &= y_k + v_{k+1/2} \Delta t, \\ v_{k+3/2} &= v_{k+1/2} + a(t_k + \Delta t, y_{k+1}) \Delta t. \end{aligned} \quad (15)$$

Как и метод Верле, метод с перешагиванием имеет второй порядок точности. Он не является самостартующим, для запуска требуется знание величины  $v_{1/2}$ . Простейшим способом ее вычисления является один шаг метода Эйлера (точнее, половина шага):

$$v_{1/2} = v_0 + a(t_0, y_0) \frac{\Delta t}{2}.$$

Погрешность этой формулы имеет второй порядок, но поскольку она применяется лишь один раз, то общий второй порядок вычислительной схемы не ухудшается.

Строго говоря, методы Верле и перешагивания можно применять и в тех случаях, когда правые части зависят от скорости, но если эта зависимость нелинейная, тогда на каждом шаге интегрирования очередное приближение для скорости придется находить, численно решая нелинейное уравнение.

Нужно отметить, что оба последних метода являются симплектическими. В приложениях к физическим задачам это означает, что они являются предпочтительными, если при численном интегрировании важно сохранять некоторые характеристики, например энергию.

Как отмечалось выше, названия методов не являются полностью установившимися. Так, например, в [3.13] leapfrog-методом называется следующая схема:

$$\begin{aligned} y_{k+1/2} &= y_k + v_k \frac{\Delta t}{2}, \\ v_{k+1} &= v_k + a\left(t_k + \frac{\Delta t}{2}, y_{k+1/2}\right) \Delta t, \\ y_{k+1} &= y_{k+1/2} + v_{k+1} \frac{\Delta t}{2}. \end{aligned} \quad (16)$$

Встречается и симметричная к (16) версия:

$$\begin{aligned}v_{k+1/2} &= v_k + a(t_k, y_k) \frac{\Delta t}{2}, \\y_{k+1} &= y_k + v_{k+1/2} \Delta t, \\v_{k+1} &= v_{k+1/2} + a(t_{k+1}, y_{k+1}) \frac{\Delta t}{2}.\end{aligned}\tag{17}$$

Корректировка перемещения по методу Эйлера в англоязычной литературе часто обозначается словом *drift* (дрейф, медленное перемещение), а корректировка скоростей – словом *kick* (удар, толчок). Поэтому соотношения (16) и (17) часто называют вычислительными алгоритмами DKD (drift-kick-drift) и KDK (kick-drift-kick), соответственно. Впрочем, в некоторых статьях аббревиатура `lfkdk` (leapfrog kick-drift-kick) служит для обозначения схемы Верле в скоростях (14). Сравнение различных вариантов leapfrog-метода с описанными выше методами Эйлера-Ричардсона, Хойна и Верле предоставляется читателю.

Обсуждение методов более высоких порядков выходит за рамки данного пособия. Отметим здесь лишь, что одними из наиболее популярных являются методы Рунге-Кутты, позволяющие адаптивно подстраивать шаг интегрирования под особенности ОДУ и обеспечивать требуемую точность даже при достаточно большом значении этого шага. К их недостаткам следует отнести отсутствие сохранения энергии в целом ряде важных приложений. Интересующимся сравнительным анализом современных методов высших порядков можно порекомендовать работу [3.1].

**Решение задачи Коши с использованием библиотеки `scipy`.** Пакет `scipy` содержит несколько функций, предназначенных для решения начальных задач. Самой употребительной из них является универсальная функция `solve_ivp` из подмодуля (sub-package) `integrate`. Данная функция предназначена для решения задачи Коши для уравнения (точнее, системы уравнений) первого порядка следующего вида:

$$\frac{d\vec{y}(t)}{dt} = \vec{f}(t, \vec{y}(t)), \quad \vec{y}(t_0) = \vec{y}_0\tag{18}$$

где  $t$  – независимая переменная («время»),  $\vec{y}(t)$  – подлежащая определению  $n$ -мерная вектор-функция, а  $n$ -мерная вектор-функция  $\vec{f}(t, \vec{y})$  определяет



собственно дифференциальные уравнения, точнее, их правые части;  $\vec{y}_0$  – заданный вектор начальных значений вычисляемой функции.

Для численного нахождения приближенного решения сформулированной задачи на отрезке  $[t_0, T]$  служит следующий вызов<sup>17</sup>

```
solution = solve_ivp(fun, [t0, T], y0, method='RK45',  
                    t_eval=None, dense_output=False,  
                    **options)
```

Кратко перечислим входные параметры функции:

- `fun` – функция с заголовком `fun(t, y)`, возвращающая вектор правых частей системы. Параметр `t` является скалярным, а `y`, как и результат функции `fun`, представляет собой массив (`ndarray`) размерности `n`.
- `y0` – массив начальных значений размерности `n`.
- `method` – строка, задающая численный метод интегрирования. Кроме установленного по умолчанию явного метода Рунге-Кутты порядка точности 4/5, возможен выбор следующих значений: 'RK23', 'DOP853' (явные схемы Рунге-Кутты второго и восьмого порядка, соответственно), 'Radau' ( неявная схема Рунге-Кутты 5-го порядка), 'BDF' ( неявный многошаговый метод переменного порядка), 'LSODA' (выбор метода интегрирования происходит автоматически на основе анализа жесткости дифференциального уравнения).
- `t_eval` – упорядоченный список или массив значений параметра `t`, в которых требуется определить значения неизвестных функций. Все числа из этого списка должны лежать в пределах отрезка  $[t_0, T]$ . Если этот параметр установлен в `None` (значение по умолчанию), то функции вычисляются в точках, выбираемых решателем автоматически.
- `dense_output` определяет, достаточно ли построить решение в точках списка, заданного предыдущим параметром (этот вариант установлен по умолчанию), или в дальнейшем может потребоваться нахождение решения во всех точках отрезка  $[t_0, T]$ . Во втором

---

<sup>17</sup> Некоторые необязательные параметры опущены. Подробнее можно посмотреть в документации по пакету:

[https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.solve\\_ivp.html](https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.solve_ivp.html)

случае результат работы функции – объект `solution` – будет содержать дополнительный метод, позволяющий осуществить такое вычисление на основе сплайн-интерполяции, порядок которой согласуется с порядком метода интегрирования.

- набор дополнительных ключевых параметров `options` связан с передачей настроечных параметров тому или иному методу. В качестве примера упомянем здесь параметры `rtol` и `atol`, которые являются вещественными числами или массивами размерности `n` и задают, соответственно, относительную и абсолютную погрешность вычислительных схем. Решатель пытается гарантировать оценку локальной погрешности величиной  $atol + rtol * abs(y)$ . Приблизительно можно считать, поэтому, что `rtol` управляет количеством верных значащих цифр, если только само значение `y` не становится меньше величины `atol`. Если компоненты вектора `y` имеют разный масштаб, то может быть выгодно устанавливать значения `atol` отдельно для каждой компоненты, используя массив размерности `n`. По умолчанию `rtol=1e-3`, `atol=1e-6`.

Возвращаемое функцией `solve_ivp` значение (в примере выше – `solution`) представляет собой объект достаточно сложной структуры. Перечислим некоторые из его полей:

- `solution.t` – одномерный массив типа `ndarray` длины `n_points`, содержащий список точек из отрезка  $[t_0, T]$ , в которых было вычислено решение. Совпадает с `t_eval`, если последний задан, в противном случае количество точек и шаг (расстояние между ними) определяются автоматически выбранным методом интегрирования;
- `solution.y` – одномерный массив типа `ndarray` длины `n_points`, содержащий значения искомой функции;
- `solution.sol` – объект типа `OdeSolution` (метод, позволяющий вычислить решение в произвольной точке отрезка  $[t_0, T]$  на основе информации о решении в точках массива `solution.t` путем сплайн-интерполяции) или **None** (если параметр `dense_output` был равен **False** при вызове функции);

- `solution.message` – строка, описание причины завершения (успешного или нет) работы, например, 'The solver successfully reached the end of the integration interval.';
- `solution.success` – логическая переменная, имеющая значение **True**, если достигнут конец интервала интегрирования.

В качестве примера рассмотрим задачу Коши, приведенную на странице с описанием метода Рунге-Кутты в Википедии:

$$y'' + 4y = \cos 3t, \quad y(0) = 0.8, y'(0) = 2, \quad t \in [0, 8]. \quad (19)$$

Выполнив в задаче (19) замену

$$y(t) = y_0(t), \quad y'(t) = y_1(t),$$

приведем ее к виду (18), где

$$\vec{y}(t) = (y_0(t), y_1(t)),$$

$$\vec{f}(t, \vec{y}(t)) = (y_1(t), \cos 3t - y_0(t)), \quad (20)$$

$$\vec{y}_0 = (0.8, 2).$$

Текст программы численного решения и визуализации результатов приведен в примере кода 10, построенные графики зависимости функции  $y$  и ее производной по времени представлены на рис. 25

Фрагмент кода 10. Численное решение задачи Коши с помощью `scipy`

```

1  import vpython as vp
2  import numpy as np
3  from scipy.integrate import solve_ivp
4
5  def eq(t, y):
6      y0, y1 = y
7      return [y1, np.cos(3*t)-4*y0]
8
9  t0, T = 0, 8
10 y_0, y1_0 = 0.8, 2
11 solution = solve_ivp(eq, [t0, T], [y_0, y1_0],
12                       dense_output=True)
13
14 t = np.linspace(t0, T, 600)
15 res = solution.sol(t)

```

```

16 |
17 | gd = vp.graph(width=600, height=400, fast=False)
18 | y0 = vp.gcurve(width=2, color=vp.color.blue,
19 |               label="y(t)")
20 | y1 = vp.gcurve(width=2, color=vp.color.orange,
21 |               label="y'(t)")
22 | y0.data = zip(t, res[0])
23 | y1.data = zip(t, res[1])

```

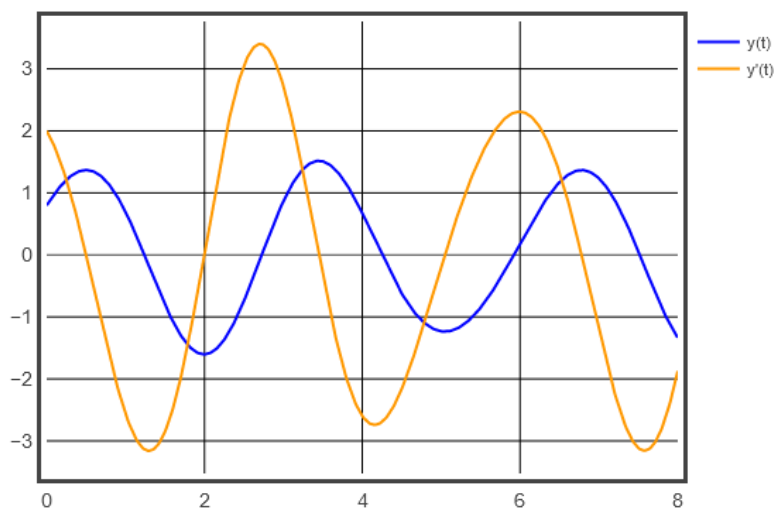


Рис. 25. Численно построенное решение демонстрационной задачи Коши

В строках 5–7 приведенного кода описана функция, возвращающая правые части системы, т.е. компоненты вектор-функции (20), в строках 9 и 10 задаются отрезок интегрирования и начальные значения функции и ее производной, соответственно, в строках 11–12 происходит вызов функции нахождения численного решения. Параметр `dense_output` позволяет воспользоваться методом `sol` объекта `solution` для вычисления решения во всех точках массива `t` (строки 14–15). Альтернативным вариантом могла бы быть передача этого массива в функцию `solve_ivp` в качестве значения для параметра `t_eval`. Строки 17–24 визуализируют результаты с использованием графических возможностей пакета `vpython`: в строке 17 создается графическое окно (область построения графика) с указанными

размерами, строки 18–19 и 20–21 инициализируют объекты для построения кривых, а затем в строках 22–23 эти кривые пополняются данными (координатами точек графика), которые отображаются в окне браузера.

Рассмотрение очень важной и сложной темы **жесткости** систем дифференциальных уравнений безусловно выходит за рамки данного пособия. Но «сказав А», т. е. упомянув о существовании методов, ориентированных на решение жестких систем, приходится хотя бы начать «говорить Б». Ограничимся поэтому неформальным определением, позаимствованным в [3.9]: жесткими принято называть системы ОДУ, численное исследование которых приводит к таким неприятным результатам как вычислительные неустойчивости, чрезмерная чувствительность к начальным данным, необходимость практически неосуществимого уменьшения шага интегрирования для достижения требуемой точности и т. п. даже для тех случаев, когда само решение является гладкой и слабо изменяющейся функцией.

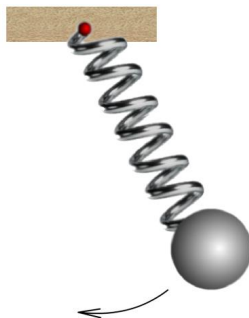


Рис. 26. Система с двумя типами колебаний

При решении прикладных задач часто требуется моделировать физические явления с очень разными временными или пространственными масштабами. Такие приложения обычно приводят к системам ОДУ, решение которых включает несколько членов, величина которых меняется с существенно различной скоростью. Например, как показано на рис. 26, шарик на пружинке может качаться слева направо, а также колебаться вверх и вниз вдоль направляющей пружины. Следовательно, в задаче есть две разные шкалы времени: период маятниковых движений системы и период колебательного движения (сжатия-растяжения) пружины. Если пружина

очень жесткая, то частота колебаний пружины будет во много раз больше частоты качания маятника. Для численного моделирования системы придется выбирать очень маленький шаг по времени, чтобы получить удовлетворительное описание этих колебаний.

Именно в зависимости от свойств изучаемой системы и приходится выбирать метод, используемый для ее решения. Прямые методы, такие как "RK45" или "RK23" подходят для нежестких систем, а методы "Radau" или "BDF" – для жестких. Если свойства системы заранее не известны, то документация по пакету `scipy` рекомендует начать с использования "RK45". Если же для решения требуется чрезмерно большое количество итераций, или численная схема не сходится или завершается аварийно, то рекомендуется пробовать "Radau" или "BDF". Метод "LSODA" тоже может быть удачным универсальным решением, поскольку в нем реализовано автоматическое переключение с явного метода на неявный при обнаружении проблем с интегрированием.

## ПРИМЕРЫ МОДЕЛИРОВАНИЯ МЕХАНИЧЕСКИХ СИСТЕМ

К сожалению, теоретические оценки не всегда позволяют гарантировать точность, скорость и надежность вычислительного алгоритма применительно к конкретной задаче. Именно поэтому так актуальны работы, посвященные реализациям, модификациям и ускорению алгоритмов в том или ином частном случае. В ряде проектных заданий этого пособия тоже будет предложено использовать для вычислительного анализа несколько алгоритмов и сделать соответствующие выводы.

Приведем два примера, частично иллюстрирующие сказанное выше. Кроме анализа и сравнения использованных алгоритмов каждый пример будет сопровождаться кодом трехмерной визуализации рассмотренной в нем механической системы.

**Колебания тележки на нелинейно-упругой пружине.** В качестве первого примера рассмотрим позаимствованную в [3.5] задачу о колебаниях без трения тела на пружине с нелинейной характеристикой, т. е. зависимостью между силой и удлинением. Схема механической системы приведена на рис. 27. Ось координат направим вправо, ее нулевому значению сопоставим начальное состояние, в котором пружина

недеформирована, а тело покоится. В данном случае движение происходит вдоль горизонтальной оси, которую традиционно обозначим через  $x$ , поэтому неизвестной в данном случае будет функция  $x(t)$ . Причиной движения является заданная в начальный момент времени скорость  $v_0$ . Силу упругости, создаваемую пружиной, будем считать нелинейной функцией координаты:

$$F = -kx - bx^3.$$

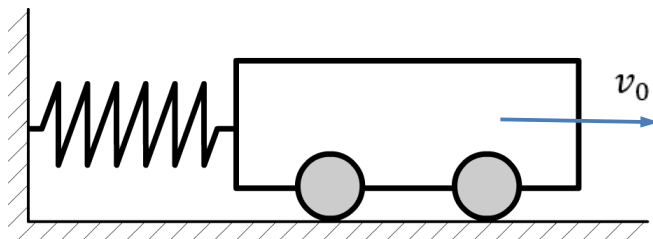


Рис. 27. Механическая колебательная система

Задача Коши, описывающая горизонтальное перемещение тела, примет, таким образом, вид:

$$\begin{aligned} \frac{dx}{dt} &= v, \\ \frac{dv}{dt} &= -\alpha x - \beta x^3, \\ x(0) &= 0, \quad v(0) = v_0 \end{aligned} \tag{21}$$

где использованы обозначения  $\alpha = k/m$ ,  $\beta = b/m$ .

Прежде чем использовать численные схемы, решаемую систему следует привести к безразмерному виду. Зависимость координаты от времени не может иметь, например, вид  $x = \sin t$ , если не указано явно, что означает  $t = 1$ . Одна секунда? Одна минута? Один год? Во избежание подобных недоразумений и осуществляется предварительное обезразмеривание уравнений и систем. За единицу времени часто принимают период колебаний. В данном случае это не очень удобно, потому что этот период достаточно сложно выражается через начальные условия и параметры  $\alpha$  и  $\beta$ . Поступим проще – будем считать, что величины всех параметров системы заданы в системе СИ, т. е. время относим к одной секунде, перемещение – к одному метру, массу тележки – к одному

килограмму. Единицей измерения коэффициента жесткости  $k$  будет ньютон, деленный на метр, а коэффициента  $b$  – ньютон, деленный на метр в кубе. Параметры  $\alpha$  и  $\beta$  имеют, соответственно, размерность  $1/c^2$  и  $1/m^2c^2$ .

Вычислительный эксперимент проведем для следующих значений параметров:  $v_0 = 1\text{ м/с}$ ,  $\alpha = 1\text{ с}^{-2}$ ,  $\beta = 9\text{ м}^{-2}\text{ с}^{-2}$ . Для решения используем методы Эйлера, Эйлера-Кромера, Эйлера-Ричардсона и, поскольку сила не зависит от скорости, Верле. Их программная реализация применительно к системе (21) приведена во фрагментах кода ниже. Общим для всех является раздел подключения библиотеки `vpython`, описания функции вычисления правых частей системы, инициализации графики и задания значений параметрам  $\alpha$ ,  $\beta$  и начальному перемещению и скорости. Этот общий раздел приведен только в тексте примера интегрирования методом Эйлера, который, по сути, является полноценной программой. В остальных случаях приведена реализация только собственно цикла вычислений.

Фрагмент кода 11. Интегрирование системы (21) методом Эйлера

```
1  import vpython as vp
2
3  alpha, beta = 1.0, 9.0
4
5  def a(t, x, v):
6      return -alpha*x-beta*x**3
7
8  gd = vp.graph(width=800, height=400,
9              xmin=0, xmax=20,
10             ymin=-3, ymax=3)
11
12  t = 0.0 #начальное время,
13  x = 0.0 # координата,
14  v = 1.0 # скорость
15  T = 4.8 #конечное время
16  dt = 0.1 #шаг интегрирования Δt
17  Euler = vp.gcurve(width=2, color=vp.color.black,
18                  label=f'метод Эйлера, Δt={dt}')
19  Euler.plot(t,x)
20
21  while t<T:
22      v, x = v+a(t,x,v)*dt, x+v*dt
23      t += dt
24      Euler.plot(t,x)
```



Фрагмент кода 12. Интегрирование методом Эйлера-Кромера

```

1  while t<T:
2      v += a(t,x,v)*dt
3      x += v*dt
4      t += dt

```

Фрагмент кода 13. Интегрирование методом Эйлера-Ричардсона

```

1  while t<T:
2      v_mid = v + a(t,x,v)*dt/2
3      x_mid = x + v*dt/2
4      v += a(t+dt/2, x_mid, v_mid)*dt
5      x += v_mid*dt
6      t += dt

```

Фрагмент кода 14. Интегрирование методом Верле

```

1  a_new = a(t, x, v)
2  while t<T:
3      a_old = a_new
4      x += v*dt + a_old*dt**2/2
5      t += dt
6      a_new = a(t, x, v)
7      v += (a_new+a_old)*dt/2

```

Результаты расчетов для указанных во фрагменте 11 значений параметров задачи с использованием различных методов сведены в единый график, приведенный на рис. 28. По оси абсцисс здесь отложено время, по оси ординат – рассчитанная координата тела. Видно, что при выбранном шаге интегрирования по времени классическая схема Эйлера приводит к существенной ошибке уже после одного периода колебаний и быстро расходится. Метод Эйлера-Кромера демонстрирует прекрасную вычислительную устойчивость, причем точность определения координаты у него даже выше, чем у метода Эйлера-Ричардсона, который дает удовлетворительную точность лишь в течение трех периодов колебаний.

Подобный «успех» метода Эйлера-Кромера, метода первого порядка, демонстрирующего точность, сравнимую с методами второго порядка и даже их превосходящую, позволили А. Кромеру сделать общий вывод о его

предпочтительности. К сожалению, чудес не бывает, и реально класс задач, где этот метод успешен, достаточно ограничен.

Тем не менее, именно этот метод использован в следующем примере кода, создающем трехмерную анимацию движения тележки, представленной на рис. 27.

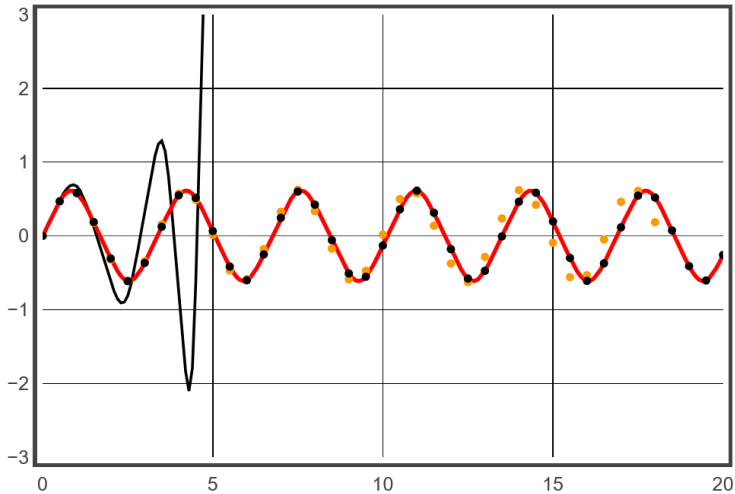


Рис. 28. Координата тележки в зависимости от времени:

- метод Эйлера,  $\Delta t=0.1$
- метод Эйлера-Кромера,  $\Delta t=0.1$
- метод Эйлера-Рунге-Кутты,  $\Delta t=0.1$
- метод Верле,  $\Delta t=0.1$

В анимации использовались следующие объекты, создаваемые средствами `python`: коробка (прямоугольный параллелепипед), четыре цилиндра – колеса, спираль – для изображения пружины, а также два объекта типа «экструзия», которые служат для отображения плоскости, по которой катится тележка, и стенки, к которой прикреплена пружина. Фрагмент кода 15 демонстрирует создание этих объектов.

## Фрагмент кода 15. 3D-изображение тележки

```

1  from vpython import *
2  scene = canvas(width=600, height=300,
3                background = color.cyan,
4                range=1.75)
5  scene.forward = vector(0,-0.5,-1)
6
7  R = 0.2      # радиус колеса
8  D = 0.05    # толщина колеса
9  L = 2       # длина коробки
10 A = 1       # высота и ширина коробки
11 S = 2       # начальная длина пружины
12
13 def whl_x(i):
14     return (2*(i//2)+1)*L/4
15 def whl_z(i):
16     return D - (i%2)*(D+A)
17 wh = []
18 for i in range (4):
19     wh.append(cylinder(pos=vector(whl_x(i), R,
20                               whl_z(i)),
21                      axis=vector(0,0,-D),radius=R,
22                      color = color.black, shininess=0.2))
23 for wheel in wh:
24     wheel.texture={'file':'my_wheel.jpg',
25                  'place':'ends'}
26
27 body = box(pos=vec(L/2, R+A/2, -A/2),
28           length=L, height=A, width=A,
29           color=color.gray(0.75))
30 spring = helix(
31     pos=vec(-S, body.pos.y, body.pos.z),
32     axis=vec(S,0,0),
33     radius=A/4, coils=10,
34     color=color.blue)
35 rt = shapes.rectangle(width=10, height=10)
36 floor = extrusion(path=[vec(0,0,0),
37                        vec(0,-0.1,0)],
38                  shape=rt, color=color.green)

```

```

39 wall = extrusion(path=[vec(-S-0.1,0,0),
40                      vec(-S,0,0)],
41                  shape=rt,
42                  texture=textures.rough)

```

Основная проблема «конструирования» сцены с тележкой состоит в определении размеров и координат составляющих ее объектов. Ниже приведена схема, поясняющая выбор и задание этих размеров в программе. Величины параметров определяются в строках 7–11.

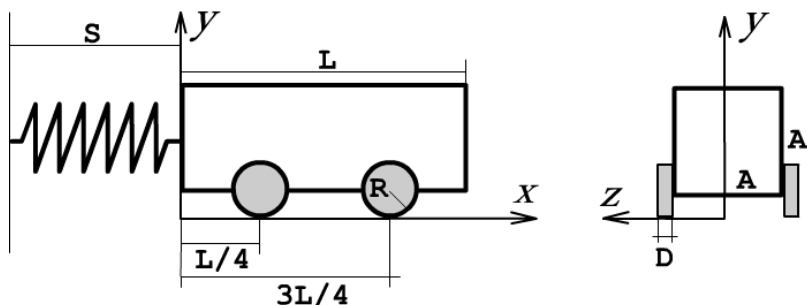


Рис. 29. Схема задания размеров тележки

Функции `whl_x(i)` и `whl_z(i)` используются для задания положения координат  $x$  и  $z$  центров колес тележки для того, чтобы создать список из четырех колес можно было одним циклом (строки 17–22). Для придания дополнительного правдоподобия движению тележки создадим имитацию вращения ее колес. Для визуализации поворота поверхности колес, т. е. торцы цилиндров, должны быть не однородными. Этого можно достичь использованием на них какого-то изображения колеса в качестве текстуры. В строках 22–23 на торцы присоединена текстура из файла, вид которой приведен на рис. 30.

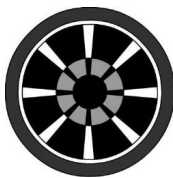


Рис. 30. Текстура колеса тележки

Задание координат корпуса тележки (строки 27–29) и пружины (строки 30–34) выполнено в соответствии со схемой на рис. 29. Что же касается поверхности, по которой тележка должна катиться, и стенки, к которой приделана пружина, то для них использованы экструзии, построенные на одном и том же плоском прямоугольнике (строка 35). Параметр `path` при создании этих объектов является по сути вектором, определяющим толщину, ориентацию и положение центра соответствующей фигуры.

Код анимирования созданной системы приведен во фрагменте 16. Моделирование движения состоит в решении уравнений (21) методом Эйлера-Кромера (строки 14–17) и корректировке в соответствии с найденным приращением координаты  $x$  положения точки привязки тележки (строка 18), колес (строки 19–20) и правого конца пружины (строка 22). Код в строке 21 служит для поворота колес на угол, связанный со смещением тележки; по умолчанию этот поворот происходит вокруг оси цилиндра. Функция `sleep(0.05)` в строке 23 обеспечивает задержку между очередными кадрами, так что частота анимации не превосходит двадцати кадров в секунду. С учетом значения шага интегрирования `dt` движение можно считать происходящим в реальном времени.

Фрагмент кода 16. Движение тележки

```
1 | alpha, beta = 1.0, 9.0
2 | def a(t, x, v):
3 |     return -alpha*x-beta*x**3
4 |
5 | t = 0.0 # начальное время,
6 | x = 0.0 # координата,
7 | v = 1.0 # скорость
8 | T = 200 # конечное время
9 | dt = 0.05 # шаг интегрирования
10 |
11 | def go(b):
12 |     global t, x, v
13 |     while t<T:
14 |         v += a(t,x,v)*dt
15 |         dx = v*dt
16 |         x += dx
17 |         t += dt
18 |         body.pos.x += dx
```

```

19     for wheel in wh:
20         wheel.pos.x += dx
21         wheel.rotate(angle=dx/R)
22     spring.axis.x += dx
23     sleep(0.05)
24
25     start = button(bind=go, text='Старт!')

```

За анимацию отвечает функция `go`, которая вызывается нажатием на созданную в строке 25 кнопку «Старт». Результат совместного выполнения кода во фрагментах 6 и 7 представлен на рис. 31.

У приведенного решения есть существенный недостаток – повторное нажатие на кнопку приведет ко второму вызову функции `go`, при этом характер движения тележки станет слабо предсказуемым. Корректировка программы, предусматривающая, например, превращение кнопки «Старт» в кнопку «Стоп», предоставляется читателю.

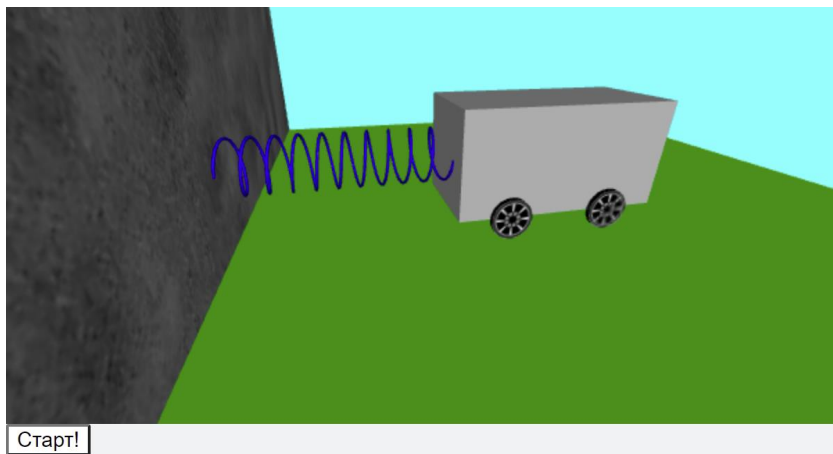


Рис. 31. 3D-визуализация движения тележки

**Полет снаряда.** В качестве второго примера рассмотрим движение снаряда, выпущенного под углом к горизонту (рис. 32). Оговоримся, что снаряд в этом примере совсем не обязательно артиллерийский, а, скорее, спортивный – например, мяч для игры в гольф или теннис. Чтобы решение

задачи описывалось не совсем уж простой параболой, добавим учет сопротивления воздуха. А чтобы аналитическое решение все-таки существовало, будем считать это сопротивление прямо пропорциональным скорости движения снаряда.

Поскольку мы не учитываем боковой ветер, то задача является двумерной: полет снаряда происходит в плоскости, полностью определяемой вектором начальной скорости (эта плоскость содержит как сам вектор скорости, так и его проекцию на горизонтальную плоскость). Будем поэтому считать, что положение снаряда определяется двумерным радиус-вектором  $\vec{r} = (x, y)$ , где  $x$  – горизонтальная координата, а  $y$  – высота снаряда. Вектор скорости снаряда имеет вид  $\vec{v} = (v_x, v_y)$ . На снаряд действует направленная вертикально вниз сила тяжести  $\vec{G} = (0, -mg)$  и сила сопротивления  $-k\vec{v}$ , где  $k$  – коэффициент сопротивления. В начальный момент времени снаряд находится в точке с координатами  $(0,0)$  и начинает двигаться с начальной скоростью  $v_0$ , вектор которой направлен под углом  $\theta$  к горизонту.

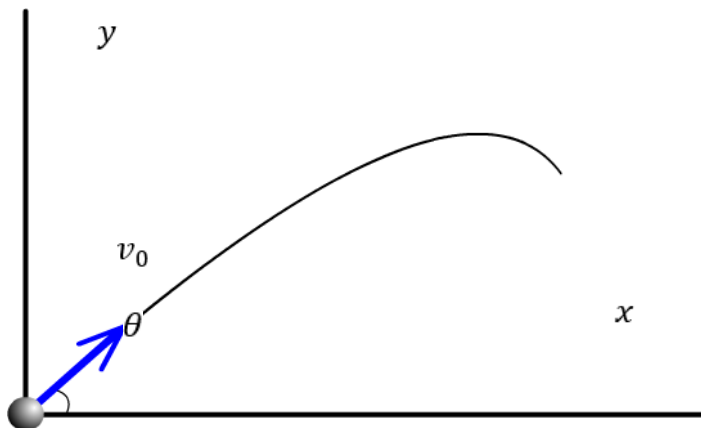


Рис. 32. Движение тела, выпущенного под углом к горизонту

Введем обозначение  $\tau = m/k$  (этот параметр имеет размерность времени). Таким образом, получаем следующую задачу Коши:

$$\frac{d^2x}{dt^2} = -\frac{1}{\tau} \frac{dx}{dt},$$

$$\frac{d^2y}{dt^2} = -g - \frac{1}{\tau} \frac{dy}{dt}, \quad (22)$$

$$x(0) = 0, \quad y(0) = 0, \quad v_x(0) = v_0 \cos \theta, \quad v_y(0) = v_0 \sin \theta.$$

Задача (22) имеет точное аналитическое решение

$$x(t) = v_0 \cos \theta (1 - e^{-t/\tau}),$$

$$y(t) = [(v_0 \sin \theta + g\tau)(1 - e^{-t/\tau}) - gt]\tau, \quad (23)$$

из которого, в частности, следует, что дальность полета снаряда по горизонтали имеет предел: даже при отсутствии поверхности Земли он не может улететь дальше, чем  $\tau v_0 \cos \theta$ .

Как и в предыдущем случае будем считать, что все входящие в систему уравнений величины заданы в системе СИ. При проведении численных расчетов в качестве начальной скорости выберем значение 100 метров в секунду, т. е. 360 километров в час. Эту скорость можно сравнить, например, с максимально известной начальной скоростью (326 км/ч) мячика для игры в гольф. Параметр  $\tau$  положим равным 0.5 с, угол вылета снаряда будем считать равным  $50^\circ$ .

Переходя к программной реализации, заметим, что в пакете `vpython` векторы являются трехмерными, т. е. объект `a` типа `vector` или, что эквивалентно, `vec`, имеет три компоненты: `a.x`, `a.y`, `a.z`. В программе будем пользоваться только двумя из них, считая, что третья компонента всех используемых векторов – радиус-вектора, скорости и ускорения – равна нулю.

Фрагмент кода 17 представляет собой раздел инициализации: здесь подключается пакет `vpython`, задаются параметры модели (строки 3–8), начальные условия для компонент радиус-вектора и скорости (строки 11–15) и вызывается функция пакета `vpython` создания области отображения графиков (строки 17–22). Значение параметра `xmax` приблизительно соответствует величине  $\tau v_0 \cos \theta$ , равной 32.14, а значение `ymax` выбрано



после нескольких экспериментов так, чтобы в получившемся окне отображались все графики.

Фрагмент кода 17. Раздел инициализации

```
1 import vpython as vp
2
3 # параметры модели
4 tau = 0.5
5 g = 9.8
6 v0 = 100
7 theta = 50 # в градусах
8 theta = vp.radians(theta) # в радианах
9
10 # начальные условия
11 t0 = 0.0
12 x0 = 0.0
13 y0 = 0.0
14 vx0 = v0*vp.cos(theta)
15 vy0 = v0*vp.sin(theta)
16
17 gd = vp.graph(width=700, height=350,
18               ymin=0, ymax=35,
19               xmin=0, xmax=35,
20               xtitle='<i>x</i>',
21               ytitle='<i>y</i>',
22               fast=False)
```

Фрагмент кода 18. График точного решения

```
1 def exact(t):
2     e = 1 - vp.exp(-t/tau)
3     return vp.vec(vx0*tau*e,
4                  ((vy0+g*tau)*e-g*t)*tau, 0)
5 r = vp.vec(x0, y0, 0)
6 Exact = vp.gcurve(width=2, color=vp.color.black)
7 Exact.plot(r.x, r.y)
8 t = t0
9 dt = 0.01
10 while t<0.1 or r.y>=0:
11     t += dt
12     r = exact(t)
13     Exact.plot(r.x, r.y)
```

Для построения графика точного решения достаточно реализовать функцию (строки 1–4, фрагмент кода 18), возвращающую вектор значений  $x$  и  $y$  по заданному значению времени; параметры  $g$ ,  $\tau$  и начальные значения скорости  $v_x0$ ,  $v_y0$  в ней считаются заданными глобально. После этого достаточно создать объект отображения кривой на графике (строка 6) и в цикле (строки 10–13) добавить точки к этому объекту. Основным условием завершения цикла является «падение» снаряда на землю – его вертикальная координата перестает быть положительной. Это же условие использовано во всех циклах реализации численных методов во фрагментах кода 20–22.

Если переходить к схемам численного интегрирования, то для них требуется предварительно описать функцию вычисления правых частей, например так, как это приведено во фрагменте 19: по вектору положения снаряда  $r$  и его скорости  $v$  вычисляется вектор ускорения по формулам (22). Строго говоря, аргумент  $r$  функции  $a(t, r, v)$  указан исключительно формально, для общности: в данном конкретном случае сила, а значит и ускорение, зависит только от скорости снаряда, а не от его положения.

Фрагмент кода 19. Функция вычисления правых частей

```
1 | def a(t, r, v):
2 |     return vp.vec(-v.x/tau, -g - v.y/tau, 0)
```

В отличие от точного решения, где величина  $dt$  отвечает лишь за гладкость кривой графика, в случае применения численных схем этот параметр – шаг интегрирования, а значит, его значение во многом является определяющим для устойчивости алгоритма и его быстродействия. Если же говорить об остальных разделах представленных фрагментов 20–22, то общим для них является задание начальных значений параметрам времени, положения и скорости (строки 2–4 во всех фрагментах), создание объекта для отображения графика и его первой точки (строки 5–8).

Что касается цикла **while**, то его тело во фрагментах кода 20, 21 и 22 является полным отражением численных схем (8), (9), (12) соответственно. Единственным отличием является добавление очередной точки на график (последняя строчка кода).

Фрагмент кода 20. Решение методом Эйлера

```
1 dt = 0.3
2 t = t0
3 r = vp.vec(x0,y0, 0)
4 v = vp.vec(vx0, vy0, 0)
5 Euler = vp.gcurve(width=2, color=vp.color.blue,
6                   markers=True,
7                   marker_color=vp.color.orange)
8 Euler.plot(r.x, r.y)
9 while t<0.1 or r.y>=0:
10     v, r = v + a(t, r, v)*dt, r + v*dt
11     t += dt
12     Euler.plot(r.x, r.y)
```

Фрагмент кода 21. Решение методом Эйлера-Кромера

```
1 dt = 0.03
2 t = t0
3 r = vp.vec(x0,y0, 0)
4 v = vp.vec(vx0, vy0, 0)
5 ECr = vp.gcurve(width=3, color=vp.color.green)
6 ECr.plot(r.x, r.y)
7 while t<0.1 or r.y>=0:
8     v += a(t,r,v)*dt
9     r += v*dt
10    t += dt
11    ECr.plot(r.x, r.y)
```

Фрагмент кода 22. Решение методом Эйлера-Ричардсона

```
1 dt = 0.3
2 t = t0
3 r = vp.vec(x0, y0, 0)
4 v = vp.vec(vx0, vy0, 0)
```

```

5 ER = vp.gcurve(width=2, color=vp.color.orange,
6               markers=True,
7               marker_color=vp.color.blue)
8 ER.plot(r.x, r.y)
9 while t<0.1 or r.y>=0:
10     v_mid = v + a(t,r,v)*dt/2
11     r_mid = r + v*dt/2
12     v += a(t+dt/2, r_mid, v_mid)*dt
13     r += v_mid*dt
14     t += dt
15     ER.plot(r.x, r.y)

```

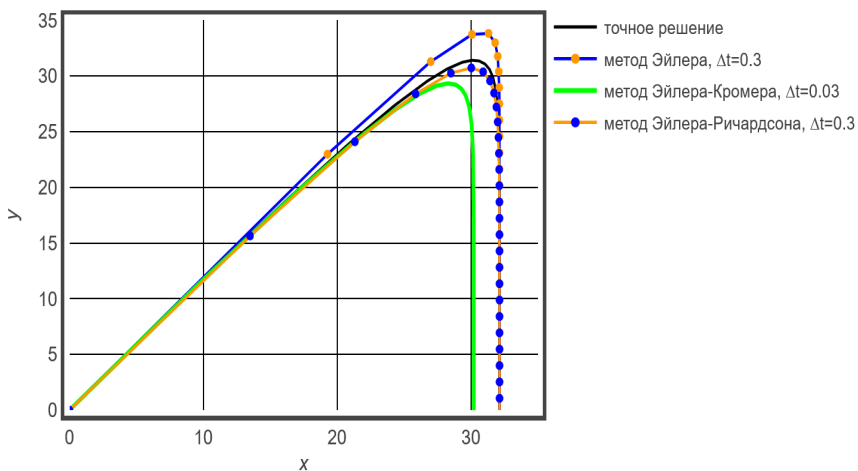


Рис. 33. Положение снаряда в пространстве:  
сравнение численных методов с точным решением

Результаты численных расчетов приведены на рис. 33. В этом случае метод Эйлера-Кромера оказался в «отстающих». Обратите внимание, что для достижения результата, сравнимого по точности с полученным стандартным методом Эйлера, шаг интегрирования метода Эйлера-Кромера пришлось уменьшить в 10 раз! Чтобы график приближенного решения визуально совпал с графиком точного, шаг интегрирования  $\Delta t$  в методе Эйлера-Кромера следовало выбрать равным 0.001, в методе Эйлера – 0.03, а в методе Эйлера-Ричардсона – 0.15. Таким образом, в этой задаче свое

существенное преимущество продемонстрировал метод второго порядка точности.

Создание трехмерной анимации в этом случае существенно проще в силу простоты перемещающегося объекта. Для изображения снаряда естественно использовать сферу. Единственное, что здесь может потребоваться, это найти подходящую текстуру<sup>18</sup> и подобрать размер объекта так, чтобы в полете его было достаточно хорошо видно. В приведенном примере размер мяча существенно увеличен для наглядности (см. рис. 34).

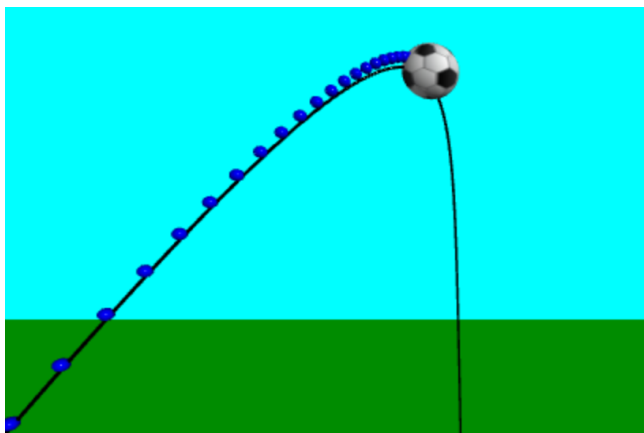


Рис. 34. Фрагмент анимации. Интегрирование методом Эйлера,  $\Delta t = 0.05$

В качестве интерактивных элементов кроме кнопки начала движения в приведенном примере добавлена возможность выбора метода интегрирования и угла наклона к горизонту вектора начальной скорости (рис. 35). Корректировка последнего автоматически приводит к изменению положения траектории, соответствующей точному решению. Для ее изображения, использован объект `curve`; точное решение изображается пространственной кривой черного цвета. Функция создания списка точек, соответствующих точному решению, приведена во фрагменте кода 23. Вызов функции `curve` (строчки 13–14) приводит к отображению траектории.

---

<sup>18</sup> В примере использована текстура футбольного мяча, загруженная с сайта <https://www.vecteezy.com/>

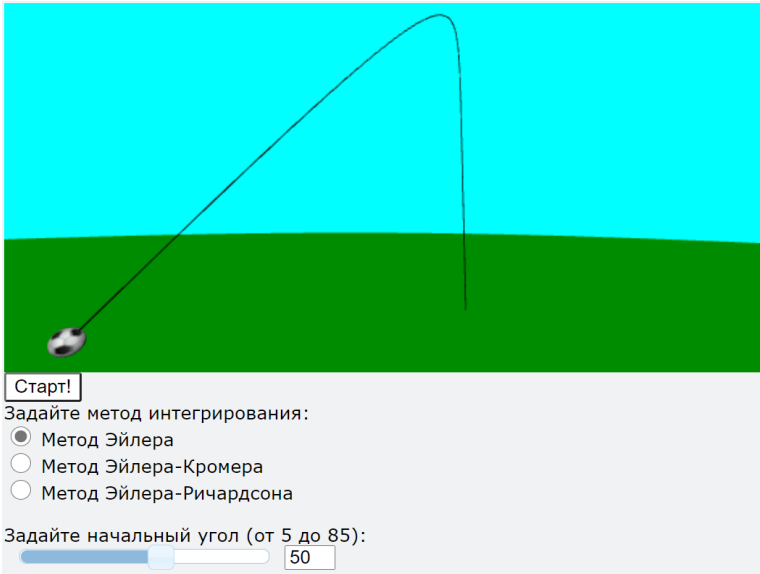


Рис. 35. Интерактивные элементы настройки параметров модели

Фрагмент кода 23. Траектория, соответствующая точному решению

```

1  def make_tr (vx0, vy0, tau):
2      point = vec(x0, y0, 0)
3      tr = [point]
4      t = t0
5      dt = 0.1
6      while t<0.1 or point.y>=0:
7          t += dt
8          e = 1 - exp(-t/tau)
9          point = vec(vx0*tau*e,
10                     ((vy0+g*tau)*e-g*t)*tau,0)
11         tr.append(point)
12     return tr
13 sp = curve(make_tr(vx0, vy0, tau), radius=0.05,
14             color=color.black)

```

Для единообразного использования численных методов один шаг интегрирования каждым из них вынесен в отдельную функцию (фрагмент 24). Входными параметрами этих функций являются значения радиус-вектора и скорости в момент времени  $t$ , шаг интегрирования и функция, вычисляющая правые части системы. В данном случае использовалась

функция из фрагмента кода 10. Результатом являются радиус-вектор и скорость в момент времени  $t + \Delta t$ .

Фрагмент кода 24. Одношаговые интеграторы

```
1 def Euler_step(t, r, v, a, dt):
2     v, r = v + a(t, r, v)*dt, r + v*dt
3     return r, v
4
5 def EulerCr_step(t, r, v, a, dt):
6     v += a(t,r,v)*dt
7     r += v*dt
8     return r, v
9
10 def EulerR_step(t, r, v, a, dt):
11     v_mid = v + a(t,r,v)*dt/2
12     r_mid = r + v*dt/2
13     v += a(t+dt/2, r_mid, v_mid)*dt
14     r += v_mid*dt
15     return r, v
```

Какая из функций из фрагмента 24 будет использована для моделирования полета, зависит от положения переключателя (см. рис. 35). Это потребовало корректировки функции `go`. Новый вариант приведен во фрагменте 25.

Фрагмент кода 25. Анимация полета

```
1 def go(b):
2     t = t0
3     dt = 0.05
4     r = vec(x0, y0, 0)
5     v = vec(vx0, vy0, 0)
6     while t<0.1 or r.y>=0:
7         if radios[0].checked:
8             r, v = Euler_step(t, r, v, a, dt)
9         elif radios[1].checked:
10            r, v = EulerCr_step(t, r, v, a, dt)
11        else:
12            r, v = EulerR_step(t, r, v, a, dt)
13        t += dt
14        ball.pos = r
15        sleep(dt)
```

Добавление интерактивных элементов делает код уже заметно более громоздким. Возможный вариант приведен во фрагменте 26. Подробное описание использованного кода содержится в разделе «Интерактивные элементы» главы 1 (фрагмент кода 6).

Фрагмент кода 26. Интерактивные элементы

```
1 wtext(text='Задайте метод интегрирования:')
2 scene.append_to_caption('\n')
3 def rb_handler(rb):
4     for btn in radios:
5         if btn.id!=rb.id:
6             btn.checked = False
7 radios=[]
8 for i in range(3):
9     radios.append(radio(bind=rb_handler, id=i))
10    scene.append_to_caption('\n')
11 radios[0].text='Метод Эйлера'
12 radios[1].text='Метод Эйлера-Кромера'
13 radios[2].text='Метод Эйлера-Ричардсона'
14 radios[0].checked = True
15 scene.append_to_caption('\n')
16 wtext(text='Задайте начальный угол, от 5 до 85:')
17 scene.append_to_caption('\n')
18
19 def sl_handler(sl):
20     global theta, vx0, vy0
21     theta = radians(sl.value)
22     inp.text = sl.value
23     vx0 = v*cos(theta) # корректировка скорости
24     vy0 = v*sin(theta)
25     sp.clear()         # перерисовка траектории
26     sp.append(make_tr(vx0, vy0, tau))
27
28 def inp_handler(w):
29     n = int(w.number)
30     if n<5:
31         n = 5
32     elif n>85:
33         n = 85
34     ang.value = n
35     sl_handler(ang)
36
```



```

37 | ang = slider(bind=sl_handler,
38 |             min=5, max=85,
39 |             step=1, value=50, length=243)
40 |
41 | inp = wininput(bind=inp_handler, type='numeric',
42 |              text='50', width=40)

```

Здесь нужно выделить строки 23–26, которые служат не только для задания новых значений параметрам начальной скорости в зависимости от угла вылета снаряда, но и для перерисовки траектории полета, соответствующей точному решению: из объекта `sp` типа пространственная кривая, отвечающего за изображение этой траектории, сначала удаляются все данные, а затем он наполняется новым списком, соответствующим текущему состоянию. Перерисовка кривой происходит практически мгновенно.

В заключение следует заметить, что моделирование полета спортивного снаряда, например мяча для гольфа – гораздо более сложная задача, и для ее адекватного исследования уже нельзя ограничиваться моделью материальной точки, а необходимо, как минимум, учитывать вращение. А если говорить о движении мяча по газону, то на эту тему не только публикуются статьи, но и защищаются диссертации [3.3].

### ЛИТЕРАТУРА К ГЛАВЕ 3

- 3.1. Авдюшев В.А. Численное моделирование орбит небесных тел. – Томск: Издательский Дом Томского государственного университета, 2015. – 336 с.
- 3.2. Вабищевич П.Н. Численные методы: Вычислительный практикум. – М.: Едиториал УРСС, 2021. – 320 с.
- 3.3. Мигунова Д. С. О движении мяча по травяному газону [Электронный ресурс]: автореф. дисс. ... канд. физ.-мат. наук: 01.02.01 / МГУ, 2012 – URL: <http://mech.math.msu.su/~snark/files/vak/arzf4.pdf> (дата обращения 01.02.2021).
- 3.4. Поттер Д. Вычислительные методы в физике. – М.: Мир, 1975. – 392 с.
- 3.5. Cromer A. Stable solutions using the Euler approximations // *American Journal of Physics*. – 1981. – Vol. 49. – Pp. 455–459. doi: 10.1119/1.12478
- 3.6. Gezerlis A. *Numerical Methods in Physics with Python*. – Cambridge University Press, 2020. – 586 p.

- 3.7. Jakobsen T. Advanced Character Physics. [Электронный ресурс] – URL: [https://www.researchgate.net/publication/228599597\\_Advanced\\_character\\_physics](https://www.researchgate.net/publication/228599597_Advanced_character_physics) (дата обращения 01.02.2021).
- 3.8. Kiusalaas J. Numerical Methods in Engineering with Python 3. – Cambridge University Press, 2013. – 422 p.
- 3.9. Kong Q., Siau T., Bayen A. Python Programming and Numerical Methods: A Guide for Engineers and Scientists. – Academic Press, 2020. – 480 p.
- 3.10. Landau R.H., Pérez M. J. Computational Problems for Physics with Guided Solutions Using Python. – CRC Press, 2018. – 389 p.
- 3.11. Linge S., Langtangen H. P. Programming for Computations – Python. Second ed. – Springer, 2020. – 224 p.
- 3.12. Verlet L. Computer experiments on classical fluids. I. Thermodynamical properties of Lennard-Jones molecules // Physical Review. – 1967. – Vol. 159, № 1. – Pp. 98–103. doi: 10.1103/PhysRev.159.98
- 3.13. Wang J. Computational modeling and visualization of physical systems with Python. – Hoboken, NJ: John Wiley & Sons, 2015. – 475 p.

## ГЛАВА 4. ПРОЕКТНОЕ ЗАДАНИЕ «ПРУЖИННАЯ МОДЕЛЬ ТВЕРДОГО ТЕЛА»

### ФОРМУЛИРОВКА ЗАДАНИЯ

Проектное задание состоит в разработке пружинной модели твердого тел, ее математической и алгоритмической реализации и визуализации средствами библиотеки `numpy`. В ходе выполнения задания требуется:

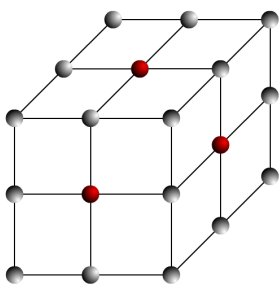
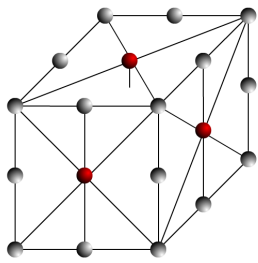
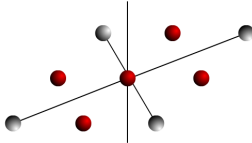
1. Смоделировать падение тела, имеющего нулевую начальную скорость, на горизонтальную твердую поверхность, считая удар абсолютно упругим. Высоту падения выбрать в диапазоне от 5 до 10 характерных размеров созданной модели. Подобрать 2–3 набора параметров (масса частицы, расстояние между частицами, жесткости пружин, время движения), демонстрирующих особенности движения системы.
2. Численное решение дифференциальных уравнений осуществить двумя способами: методом первого порядка и методом второго порядка в соответствии с номером варианта и таблицей 5. Дать сравнительный анализ обоих методов (скорость, зависимость точности/потери устойчивости от выбора шага интегрирования по времени и т. п.). В качестве критерия точности интегрирования использовать величину отклонения полной энергии системы от ее начального значения. Визуализация движения тела должна сопровождаться построением графика зависимости полной энергии системы от времени. Дополнительным критерием при сравнении методов может величина промежутка времени, в течение которого движение остается регулярным.
3. В качестве отчета представить:
  - Файл в формате MS Word или pdf, в котором содержится описание геометрии системы, значения параметров, при которых осуществлялось моделирование, сравнительный анализ методов интегрирования
  - Файл на языке Питон, визуализирующий пружинную модель тела и ее движение. Разработанная программа должна позволять пользователю выбирать метод интегрирования и задавать массу одного из шариков, жесткость пружин, базовое расстояние, шаг и время интегрирования. Значение параметров по умолчанию должно соответствовать одному из описанных в отчете случаев.

В представленных ниже в таблице 4 вариантах необходимо считать, что масса частиц, обозначенных темным (красным) цветом, в два раза больше массы частиц, обозначенных серым цветом. Связи, которые требуется учитывать, изображены сплошными линиями черного цвета, или, для облегчения понимания чертежа, пунктирными линиями. Учитывать разницу в механических характеристиках (жесткости) пружин, изображенных сплошной линией и пунктиром, не требуется, но и не запрещается.

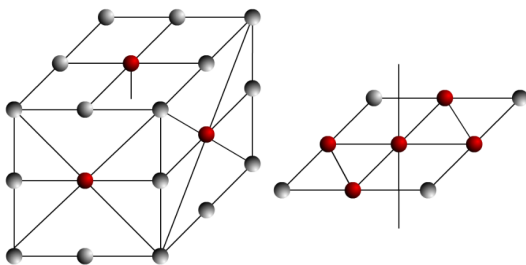
Кроме чертежа в условии каждого варианта содержатся дополнительные комментарии, уточняющие изображение на чертеже, а иногда и дополнительные элементы чертежа (вид сверху, сбоку и т. д.)

В ряде случаев (чтобы не загромождать рисунок) связи вообще не отмечены, но информация об их наличии содержится или на дополнительном рисунке, или в пояснениях к варианту.

Таблица 4. Варианты задания

Вариант 1		
	<p>«Восемь кубиков»</p> <p>Нижний ряд частиц полностью повторяет верхний ряд.</p> <p>В каждом ряду – 9 частиц. Внутренняя частица среднего ряда – серая.</p> <p>Диагональных связей нет.</p>	
Вариант 2		
		<p>Расстояния между частицами аналогичны первому варианту.</p> <p>Нижний ряд частиц полностью повторяет верхний ряд. В каждом ряду – 9 частиц.</p> <p>На правом рисунке изображены частицы среднего ряда и связи между ними</p>

### Вариант 3



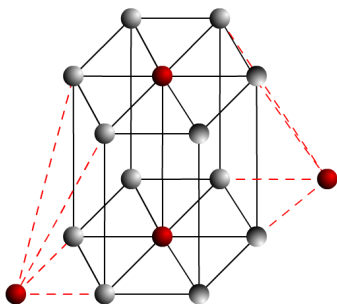
Расстояния между частицами аналогичны первому варианту.

Нижний ряд частиц полностью повторяет верхний ряд.

В каждом ряду – 9 частиц.

На правом рисунке изображены частицы среднего ряда и связи между ними

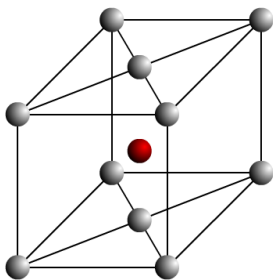
### Вариант 4



Верхняя грань – правильный шестиугольник, в нижней грани кроме шестиугольника имеются еще две частицы.

Расстояние между слоями считать в два раза большим стороны шестиугольника

### Вариант 5

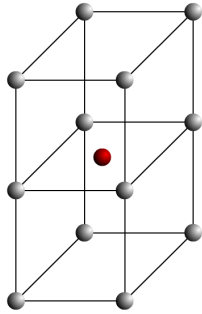


«Кубик с точкой в центре»

Необходимо считать, что красная центральная частица соединена связями со всеми десятью серыми частицами.

Все связи между серыми частицами представлены на рисунке.

### Вариант 6

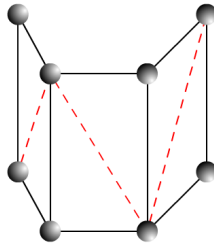
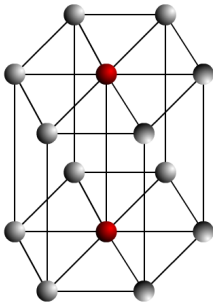


«Кубик на кубике»

Необходимо считать, что красная центральная частица соединена связями со всеми двенадцатью серыми частицами.

Все связи между серыми частицами представлены на рисунке.

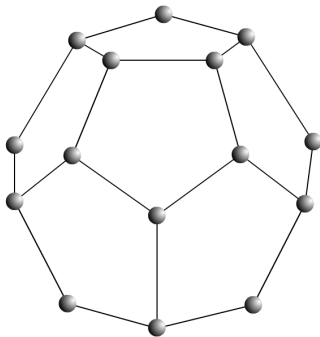
### Вариант 7



В обеих плоскостях лежат правильные шестиугольники. На чертеже справа показаны дополнительные связи между частицами, лежащими на боковых сторонах фигуры.

Расстояние между нижней и верхней гранями равно удвоенной стороне шестиугольника.

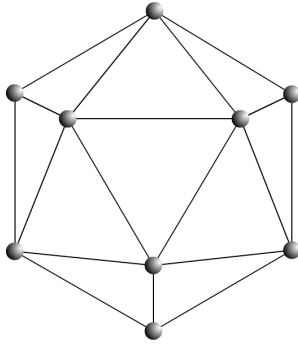
### Вариант 8



Додекаэдр.

Информацию о координатах вершин можно найти в интернете. В отчете в этом случае должен содержаться адрес сайта.

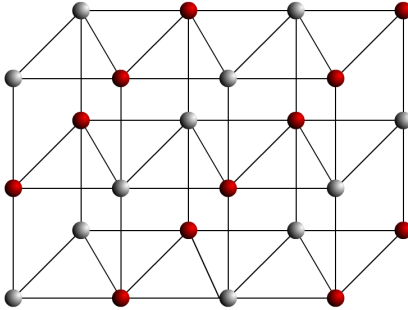
Вариант 9



Икосаэдр.

Информацию о координатах вершин можно найти в интернете. В отчете в этом случае должен содержаться адрес сайта.

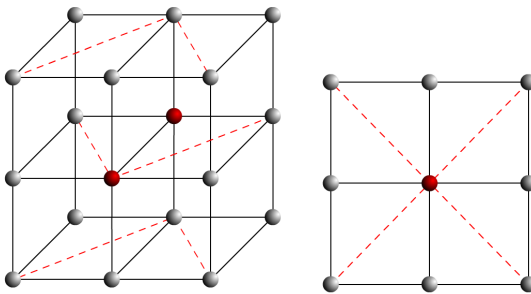
Вариант 10



«Шесть кубиков»

Все связи изображены на рисунке.

Вариант 11

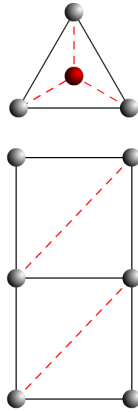
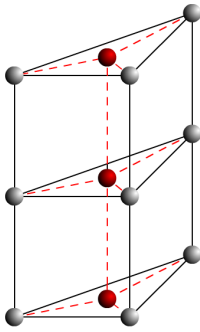


«Четыре кубика»

Отдельно показаны не помещившиеся на рисунке связи между частицами лицевой стороны фигуры. На задней стороне действуют такие же связи.

Другими словами, красная частица взаимодействует со всеми частицами своей грани.

### Вариант 12

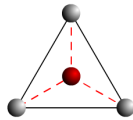
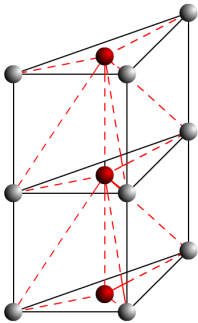


Треугольная призма, два слоя. В основании – правильный треугольник.

На рисунках справа – вид сверху и вид сбоку, на котором отмечены дополнительные связи.

Толщина одного слоя равна длине стороны треугольника.

### Вариант 13



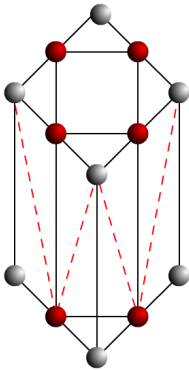
Треугольная призма, два слоя. В основании – правильный треугольник.

Толщина одного слоя равна длине стороны треугольника.

Справа показан вид сверху.

Все связи изображены на основном рисунке

### Вариант 14.



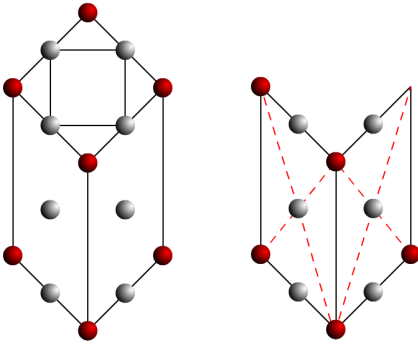
«Квадрат в квадрате»: вершины меньшего лежат на серединах сторон большего.

Нижнее основание полностью копирует верхнее.

На всех боковых гранях, в том числе и на невидимых, действует дополнительная диагональная связь, изображенная красной линией. Высота фигуры равна удвоенной длине стороны большего квадрата.



### Вариант 15



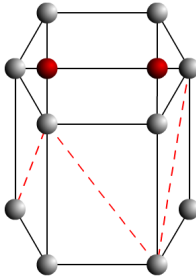
«Квадрат в квадрате»: Вершины меньшего лежат на серединах сторон большего.

Нижнее основание полностью копирует верхнее.

Связи срединных частиц каждой боковой грани отмечены на отдельном рисунке.

Высота фигуры равна удвоенной длине стороны большего квадрата.

### Вариант 16



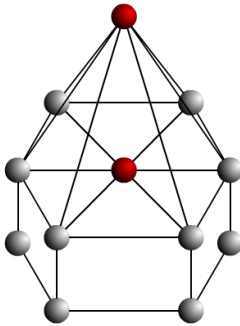
В обеих плоскостях лежат правильные шестиугольники.

Нижнее основание полностью копирует верхнее, в том числе содержит две красные частицы.

На каждой боковой грани действует одна диагональная связь, как показано на рисунке.

Высота фигуры равна длине стороны шестиугольника

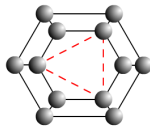
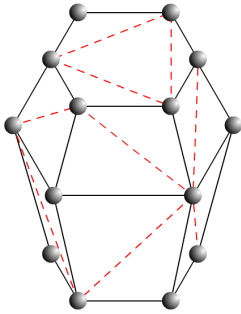
### Вариант 17



Два правильных шестиугольника (нижний – полная копия верхнего). Над верхней гранью расположена дополнительная частица, связи которой образуют правильную пирамиду.

Расстояние между слоями – половина длины стороны шестиугольника, высота пирамиды равна длине этой стороны.

### Вариант 18

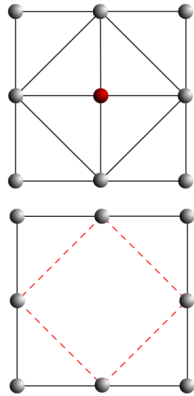
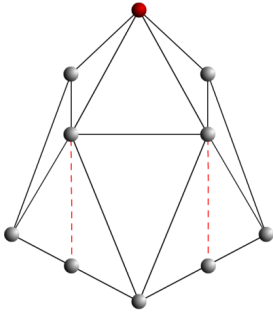


В каждой плоскости – правильный шестиугольник. В срединной плоскости он больше (длина стороны в два раза больше).

Нижний шестиугольник – полная копия верхнего. Расстояние между плоскостями равно длине стороны большего шестиугольника.

На втором рисунке – вид сверху.

### Вариант 19

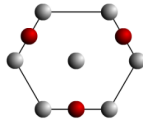
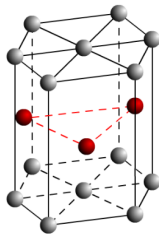


В основании фигуры – квадрат в квадрате (правый нижний рисунок), на среднем уровне – четыре частицы меньшего квадрата. Все венчает дополнительная частица и пирамида ее связей.

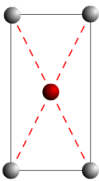
Справа сверху – вид сверху.

Расстояние между плоскостями равно длине стороны большего квадрата, высота пирамиды равна длине стороны меньшего квадрата.

### Вариант 20



Правильная шестиугольная призма. Высота призмы равна удвоенной стороне шестиугольника.



В центре трех боковых граней расположены дополнительные красные частицы, соединенные между собой, а также связанные со всеми частицами своей грани, как показано справа внизу.

Таблица 5. Методы решения уравнений

<b>Номер варианта</b>	<b>Первый метод</b>	<b>Второй метод</b>
1	Метод Эйлера	Метод Эйлера-Ричардсона
2	Метод Эйлера-Кромера	Метод Хойна
3	Метод аппроксимации по срединной точке	Алгоритм Верле в скоростях
4	Метод Эйлера	DKD-вариант метода с перешагиванием
5	Метод Эйлера-Кромера	KDK-вариант метода с перешагиванием
6	Метод аппроксимации по срединной точке	Метод Эйлера-Ричардсона
7	Метод Эйлера	Метод Хойна
8	Метод Эйлера-Кромера	Алгоритм Верле в скоростях
9	Метод аппроксимации по срединной точке	DKD-вариант метода с перешагиванием
10	Метод Эйлера	KDK-вариант метода с перешагиванием
11	Метод Эйлера-Кромера	Метод Эйлера-Ричардсона
12	Метод аппроксимации по срединной точке	Метод Хойна
13	Метод Эйлера	Алгоритм Верле в скоростях
14	Метод Эйлера-Кромера	DKD-вариант метода с перешагиванием
15	Метод аппроксимации по срединной точке	KDK-вариант метода с перешагиванием
16	Метод Эйлера	Метод Эйлера-Ричардсона
17	Метод Эйлера-Кромера	Метод Хойна
18	Метод аппроксимации по срединной точке	Алгоритм Верле в скоростях
19	Метод Эйлера-Кромера	DKD-вариант метода с перешагиванием
20	Метод аппроксимации по срединной точке	KDK-вариант метода с перешагиванием

## БАЗОВЫЕ СВЕДЕНИЯ

На первый взгляд, предложенное задание является чисто демонстрационным, позволяя реализовать приведенные в главе 2 алгоритмы и вычислительные схемы и обеспечить их визуализацию на примере конкретной механической системы. Однако на самом деле, масс-пружинные системы, или, другими словами, модели, основанные изучении движения набора шариков, соединенных пружинами, очень активно используются в различных областях современного математического моделирования и до сих пор ставят перед их исследователями новые задачи.

Прежде всего, важным достоинством этих моделей является возможность продемонстрировать основные принципы математического моделирования: замена сложного явления или процесса набором простых схем, которые затем уточняются, развиваются и в конце концов позволяют достичь нужной степени адекватности. Хорошим примером такого подхода является рассмотрение иерархической цепочки моделей системы шарик-пружина в учебном пособии [4.1]. Эти модели получаются одна из другой при последовательном отказе от предположений, идеализирующих изучаемый объект: изучаются различные варианты действующих на систему внешних сил, меняются точки крепления пружины и свойства закрепления, в рассмотрение вводятся силы трения различной природы и, наконец, учитывается нелинейность свойств пружин. В одних случаях усложнение не вносит ничего нового в поведение системы, в других ее свойства меняются существенным образом. Путь «от простого к сложному» дает возможность поэтапно изучать все более реалистичные модели и сравнивать их свойства.

Масс-пружинные модели дают богатый иллюстративный материал для курсов по дифференциальным уравнениям, ориентированным на инженеров. С разнообразными примерами их использования в этом качестве, начиная от задачи о колебаниях шарика на пружинке и заканчивая простыми моделями автомобильной подвески, можно познакомиться в [4.9].

Использование масс-пружинных моделей деформируемых твердых и газообразных тел имеет давнюю историю. Этот подход позволяет рассчитывать характеристики стержней, пластин и даже многослойных тел, используя цепочки из  $N$  связанных колебательных звеньев. Аналогии между такими цепочками и поведением электрических колебательных

контуров привели к тому, что, с одной стороны, эти системы используются при моделировании волновых явлений в радиоэлектронике, акустике и оптике, а с другой – методики расчета электронных схем применяют в новых задачах, связанных с масс-пружинными моделями [4.3].

В 1989 году Р. Бликхан [4.7] предложил простую модель бега и прыжков, основанную на поведении невесомой пружинки и присоединенной к ней массы. Центр масс человека представлен как единая масса, прикрепленная к пружине, которая контактирует с землей при каждом толчке. Бег, таким образом, состоит из фазы «полета», т. е. движения массы в поле силы тяжести, и фазы соударения с землей, когда пружина упруго деформируется и сначала накапливает энергию, а затем распрямляется и снова запускает массу в полет. Несмотря на кажущуюся простоту эта модель смогла удачно описать качественную зависимость механических параметров, характеризующих бег и прыжки людей, от скорости. Вызывая много споров, она, тем не менее, продолжает активно использоваться в современной биомеханике бега.

Еще одним примером использования масс-пружинных систем в биомеханике является основанный на их применении метод моделирования, в каком-то смысле альтернативой методу конечных элементов. Замена биологической ткани набором масс, соединенных пружинками (рис. 36), получила распространение при моделировании результатов хирургических операций, в тех случаях, когда скорость расчетов является определяющей. Например, в работе [4.4] такие модели использованы для описания свойств кожи и жировой ткани. Предложенный там подход можно использовать для моделирования растягиваемых тканей при помещении под них имплантатов (для улучшения внешнего вида пациента) или эспандеров (в реконструктивных целях, т. е. для выращивания донорских тканей). Целью моделирования является помощь хирургам в правильном выборе имплантата или расчета необходимого количества донорских тканей.

В работе [4.11] этот подход применялся для моделирования механики сердечной деятельности. Сплошная среда заменяется пространственной сеткой тетраэдров, в узлах которой находятся сосредоточенные массы. Свойства материала при этом определяются типом сетки и функцией потенциальной энергии (свойствами пружин). Авторам удалось разработать высокопроизводительный программный комплекс для анализа электромеханического поведения левого желудочка сердца человека.

В системах виртуальной реальности часто требуется моделировать динамику систем тел, часть из которых связаны между собой пружинами или подпружиненными соединениями. Примерами могут служить рессора подвески колесного робота, смягчающая удар и толчки при перемещении по неровной поверхности, хват промышленного робота с пружиной, позволяющий брать хрупкие предметы, автоматически закрывающаяся дверь с пружиной. Системы дифференциальных уравнений, используемые для моделирования движения виртуальных роботов, содержащих такие соединения, являются жесткими. Это существенно усложняет задачу их решения в режиме реального времени – большой шаг интегрирования по времени для визуализации виртуальных сцен, содержащих большое количество объектов, связанных пружинами высокой жесткости, приводит к вычислительным неустойчивостям. Таким образом, очень актуальной является рассмотренная в [4.5] задача разработки быстрых и абсолютно устойчивых методов моделирования динамики систем тел при наличии пружин и подпружиненных соединений между некоторыми из них.

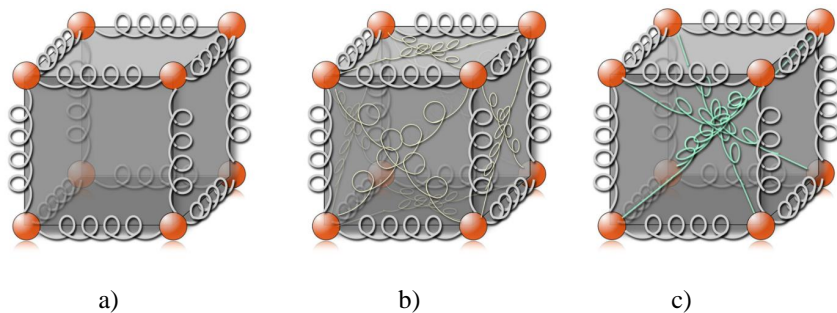


Рис. 36. Примеры масс-пружинных моделей (рисунок из работы [4.10]) с различной топологией пружинной сети: структурные пружины (а), структурные и поверхностные пружины (b), структурные и диагональные пружины (с). Для моделирования геометрии и механики деформируемого объекта такие элементы объединяются в трехмерную сеть.

Интересное приложение масс-пружинные модели нашли в задачах о квазиразвертках поверхностей, т. е. таком отображении поверхности на плоскость, при котором или длины линий, или углы между ними могут искажаться. Например, в швейной промышленности такие задачи возникают в процессе автоматизированного проектирования выкроек одежды и других тканевых изделий. Предложенный в [4.6] алгоритм

решения состоит в замене ткани или поверхности набором соединенных пружинками масс, а процесс развертки состоит в моделировании падения этих масс на плоскость. Свойства пружин, или другими словами, энергия системы, определяется на основе информации о свойствах реальной ткани.

В завершение, следуя работе [4.10], перечислим и слабые места этих моделей, которые ограничивают их использование в физическом моделировании:

- По сравнению с моделями, основанными на теории упругости, такими как методы конечных элементов или конечных разностей, большинство масс-пружинных систем не может гарантировать заданную точность. Во многих случаях уменьшение шага сетки не приводит к сходимости к точному решению исходной задачи.
- Поведение этих систем сильно зависит от топологии и разрешения сетки. При изменении сетки результат моделирования может существенно отличаться от исходного.
- Выбор функций, описывающих свойства пружин, и определение параметров этих функций – весьма трудоемкий процесс, зависящий от области применения модели.
- В ряде случаев трудно подобрать массы так, чтобы свойства материалы были однородны.
- При использовании обычных масс-пружинных систем очень нелегко создать и проконтролировать такие свойства материала, как изотропность, или анизотропия нужного типа.
- Многие биологические ткани в процессе деформирования не меняют свой объем. Обычные масс-пружинные системы не могут обеспечить условие постоянства объема для моделируемого объекта.

Важно отметить, что в отличие от проектного задания данной главы, пружинные модели, используемые в современном математическом моделировании, существенно более сложные. Как правило, это не просто упругая пружина, а пружина с демпфером, изображенная на рис. 37. В этом случае в уравнения движения добавляется слагаемое, пропорциональное скорости. Например, затухающие колебания тела на рис. 37 описываются уравнением

$$m \frac{d^2x}{dt^2} = -kx - c \frac{dx}{dt}.$$

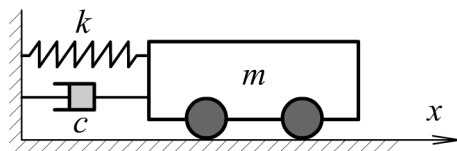


Рис. 37. Пружина с демпфирующим элементом

## ПРИМЕР ВЫПОЛНЕНИЯ ЗАДАНИЯ

В качестве примера выполнения некоторых частей проектного задания разработаем программу, визуализирующую падение тела в форме тетраэдра (рис. 38), все шарики которого имеют одинаковую массу, а пружинки – одинаковую жесткость. Интегрирование будет выполнено тремя методами первого порядка, интервал интегрирования выберем после разработки программы так, чтобы тело успело хотя бы два раза «упасть» на землю и «отскочить» от ее поверхности.

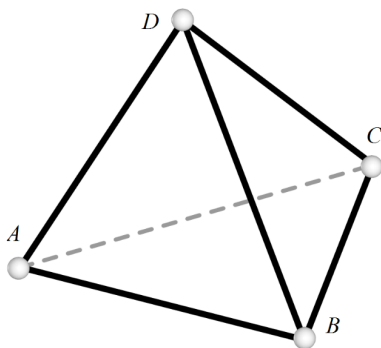


Рис. 38. Моделируемый объект. Линии соответствуют пружинам одинаковой жесткости.

Начнем с решения задачи определения координат вершин тетраэдра. Треугольник  $ABC$  расположим в горизонтальной плоскости  $Oxz$  как показано на рис. 39. Тогда точка  $A$  будет иметь координаты  $(0, 0, 0)$ , точка  $B - (d, 0, 0)$ , точка  $C - (d/2, 0, \sqrt{3}d/6)$ . Точка  $D$ , проектирующаяся в центр треугольника, получит координаты  $(d/2, \sqrt{6}d/3, \sqrt{3}d/6)$ . Эта информация будет использована ниже во фрагменте кода 31 (строки 1–5).



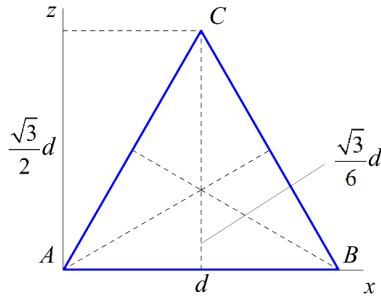


Рис. 39. Расположение основания тетраэдра.

Фрагмент кода 27. Задание параметров

```

1  from vpython import *
2
3  d = 0.1
4  h = 3*d
5  BALL_R = 0.15*d
6  BALL_COLOR = color.red
7  BALL_MASS=0.1
8  HELIX_R = BALL_R/3
9  HELIX_STIFFNESS = 100
10 g = 9.8
11 t, T, DELTA_T = 0.0, 5.0, 0.00001
12
13 scene=canvas(width=800, height=600,
14              background=vector(0.6,0.8,1.0),
15              title='Метод Эйлера')
16 ground=box(pos=vector(0,-0.1,0),
17            size=vector(100,0.1,100),
18            shininess=0, color=color.green)
19 scene.range = 5*h
20 scene.camera.pos = vec(d, 1.5*h, h)
21 scene.forward = vec(0.,0.1, -1)
22
23 gd = graph(width=600, height=200,
24           title='Энергия/центр масс системы',
25           xmin=0, xmax=T_END)

```

Что же касается собственно программы, то, как и в большинстве примеров данного пособия представленные фрагменты кода не будут ориентированы на взаимодействие с пользователем. Дополнение программы интерактивными элементами мы предоставляем читателям. Поэтому во фрагменте кода 27 зададим все параметры моделирования (характерный размер, массу, жесткость пружин, радиус и цвет шариков, визуализирующих массы, шаг и интервал интегрирования) в виде констант (строки 3–12 кода). Кроме того, в этом фрагменте

- определяется область рисования голубого цвета («небо») и задается ее заголовок, соответствующий одному из методов интегрирования (строки 13–15). В интерактивном варианте он должен меняться после выбора метода пользователем
- изображена поверхность земли путем размещения на сцене тонкого параллелепипеда зеленого цвета (строки 16–18)
- зафиксировано положение камеры (строки 19–21)
- создана область отображения графика зависимости энергии системы от времени (строки 23–25).

Определенного комментария заслуживает величина жесткости пружины. Хотя данная задача является модельной, попробуем все-таки хотя бы качественно оценить, как могут быть связаны между собой массы шариков, характерное расстояние между ними и жесткость пружины. Формула для коэффициента жесткости витой цилиндрической пружины, которая намотана из проволоки круглого поперечного сечения, имеет вид [4.2]

$$k = \frac{\mu r^4}{4nR^3}, \quad (24)$$

где  $\mu$  – модуль сдвига материала пружины,  $r$  – радиус проволоки,  $R$  – радиус намотки,  $n$  – число витков пружины.



Рис. 40. Стальные шарики, соединенные пружиной

Опираясь на формулу (24), определим жесткость пружины на рис. 40, соединяющей два стальных шарика, весящих 100 г каждый. С учетом плотности стали порядка 7.7–7.9 г/см<sup>3</sup>, радиус шарика должен быть около полутора сантиметров<sup>19</sup>. Как раз такая величина и была использована для построения изображения на рис. 40. Для значения радиуса проволоки при этом была выбрана величина 0.2 см, а для радиуса намотки – 0.5 см. Длина пружины – 10 см, а общее количество витков – 25. Переводя перечисленные единицы длины в метры и подставляя их и величину модуля сдвига для стали, 80 ГПа, в (24), получаем, что жесткость нарисованной пружины приблизительно равна 100 Н/м.

Таким образом, можно считать, что все размерные входные параметры задачи заданы в системе СИ, а следовательно, время  $t$  измеряется в секундах.

Переходим к описанию классов. В данной программе мы используем два: один для работы с шариками/массами (фрагмент кода 28), другой – для работы с пружинками (фрагмент кода 29).

Фрагмент кода 28. Описание класса для работы с массами

```

1  class Mass:
2
3      def __init__(self, pos, radius=BALL_R,
4                  color=BALL_COLOR,
5                  mass=BALL_MASS):
6
7          self.pos = pos
8          self.mass = mass
9          self.v = vec(0,0,0)
10         self.springs = []
11         self.ball = sphere(pos=pos,
12                             radius=radius,
13                             color=color)
14
15         def add_spring(self, new_s):
16             self.springs.append(new_s)
17
18         def redraw(self):
19             self.ball.pos = self.pos

```

<sup>19</sup> Если быть точным, то радиус должен лежать в диапазоне 1.45–1.46 см.

```

20     def a(self):
21         f = vec(0,0,0)
22         for spr in self.springs:
23             if self == spr.m1:
24                 another = spr.m2
25             else:
26                 another = spr.m1
27                 r = another.pos-self.pos
28                 delta_l = mag(r) - spr.length
29                 f += spr.k*delta_l*hat(r)
30         return f/self.mass + g*vector(0,-1,0)

```

Экземпляр класса `Mass` инициализируется (строки 3–5) своим уникальным положением; остальные же его параметры (собственно масса, цвет и радиус шарика, используемого при визуализации) могут иметь значения, установленные по умолчанию. При инициализации у объекта создается нулевое поле скорости (строка 8), а также пустой в момент инициализации список пружин, которыми это тело соединено с другими (строка 9). В момент создания масса сразу отображается в виде шарика, расположенного в нужном месте (строки 10–12).

К методам класса `Mass` относится пополнение списка пружин новой пружиной (строки 14–15) и перерисовка шарика (строки 17–18). Последний метод позволяет проводить вычисления и корректировать реальные положения объектов, описываемые параметром `pos`, чаще, чем осуществлять достаточно медленный процесс их отрисовки, связанный с параметром `ball.pos`. Этот же подход использован ниже у класса `Spring`, где он даже важнее, поскольку отрисовка пружины еще более медленный процесс.

Ускорение шарика определим не как свойство, а как функцию – метод класса (строки 20–30). При ее описании учтем, что на каждую массу действуют силы упругости присоединенных к ней пружин и сила тяжести (рис. 41). Сила упругости вычисляется в цикле по списку пружин. Чтобы найти длину пружины, сначала определяется второе тело (переменная `another`), как тело, связанное с пружиной, но не совпадающее с данным. Итоговая сила упругости вычисляется как произведение коэффициента жесткости пружины на ее удлинение и на единичный вектор, идущий от заданного тела к найденному второму (строка 29). Итоговое ускорение

вычисляется как сумма сил упругости, деленных на массу тела, и ускорения свободного падения.

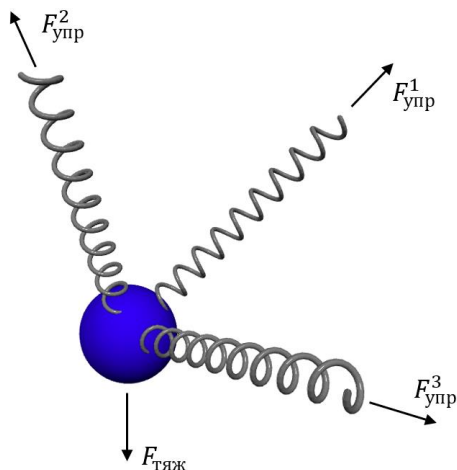


Рис. 41. Силы, действующие на шарик

Объект класса пружина инициализируется двумя массами, которые эта пружина соединяет. Что же касается жесткости, и параметров, визуализирующих пружину, то они могут быть заданы по умолчанию (строки 3–5 фрагмента кода 29). При создании пружинки вычисляется и сохраняется в поле `length` ее начальная длина, т. е. длина в недеформированном состоянии (строка 8). Визуальный элемент – объект `helix` библиотеки `vpython` – создается в строках 10–14. Его начало расположено в центре одного из шариков, а ось представляет собой вектор, соединяющий центры шариков.

Фрагмент кода 29. Описание класса для работы с пружинками

```
1 class Spring:
2
3     def __init__(self, m1, m2, k=HELIX_STIFFNESS,
4                 radius=HELIX_R, coils=25,
5                 thickness=HELIX_R/2.5):
6         self.m1 = m1
7         self.m2 = m2
8         self.length = mag(m1.pos-m2.pos)
9         self.k = k
```

```

10 |         self.spr = helix(pos=m1.pos,
11 |                         axis=m2.pos - m1.pos,
12 |                         radius=radius,
13 |                         thickness=thickness,
14 |                         coils=coils)
15 |
16 |     def cur_length(self):
17 |         return (mag(self.m2.pos - self.m1.pos))
18 |
19 |     def redraw(self):
20 |         self.spr.pos = self.m1.pos
21 |         self.spr.axis = self.m2.pos - self.m1.pos

```

Метод `cur_length` служит для вычисления длины деформированной пружины, что потребуется при вычислении силы, с которой пружина действует на связанные с ней тела. Метод `redraw`, как и в предыдущем случае, позволяет отделить вычислительные процессы от визуализации и осуществлять последнюю по мере необходимости.

Полная энергия системы представляет собой сумму кинетической энергии шариков

$$E_k = \sum \frac{m_i v_i^2}{2},$$

их потенциальной энергии в поле силы тяжести

$$E_p = \sum m_i g y_i$$

и энергии упругой деформации пружин

$$E_{\text{упр}} = \sum \frac{k_i \Delta l_i^2}{2},$$

где  $\Delta l$  – изменение длины пружины. Функция вычисления полной энергии приведена во фрагменте кода 30. Входными параметрами этой функции являются два списка: шариков, т. е. объектов класса `Mass`, и пружинок, т. е. объектов класса `Spring`.

Во фрагменте кода 30 приведена также функция вычисления центра масс (Center of Mass) системы шариков, точнее, координаты  $y$  этого центра по формуле:

$$y_{\text{ц.м.}} = \frac{\sum m_i y_i}{\sum m_i}.$$

Эта величина, также как и энергия, может использоваться для сравнения результатов численных методов.

Фрагмент кода 30. Вычисление энергии системы

```

1 def energy(masses, springs):
2     e1 = sum(m.mass*(m.pos.y*g + mag2(m.v)/2)
3         for m in masses)
4     e2 = sum(spr.k*(spr.cur_length()
5         - spr.length)**2/2
6         for spr in springs)
7     return e1+e2
8
9 def com_y(masses):
10    total_mass = sum(m.mass for m in masses)
11    y = sum(m.mass*m.pos.y for m in masses)
12    return y/total_mass

```

Для визуализации моделируемого тела необходимо создать нужное количество объектов в требуемых положениях. Эту задачу решает фрагмент кода 31. Объекты-шарики обозначены в нем буквами A, B, C, D (строки 1–5), а тело в целом определяется списком `body` этих объектов (строка 7). Для связей между массами используется матрица `links`, элемент `[i][j]` которой равен единице, если между *i*-м и *j*-м шариком есть пружинка, и нулю – в противном случае. Данная матрица создается вручную (строки 10–15), исходя из заданной схемы. Другой способ реализации связей мог бы состоять в добавлении еще одного поля в класс `Mass`, а именно – списка шариков, связанных с данным. Тогда вместо задания матрицы связей можно было бы задавать список «соседей» каждого шарика. Но и в этом случае данная операция будет чисто ручной, потому что, вообще говоря, схемы связей в проектных заданиях не являются полностью алгоритмизируемыми.

Фрагмент кода 31. Создание основных объектов

```

1 A = Mass(pos=vector(0, h, 0))
2 B = Mass(pos=vector(d, h, 0))
3 C = Mass(pos=vector(d/2, h, sqrt(3)/2*d))
4 D = Mass(pos=vector(d/2, h+sqrt(6)/3*d,
5         sqrt(3)/6*d))

```

```

6 | # все массы тела
7 | body = [A,B,C,D]
8 | n_mass = len(body)
9 | # матрица связей
10 | links = [
11 |     [0, 1, 1, 1],
12 |     [1, 0, 1, 1],
13 |     [1, 1, 0, 1],
14 |     [1, 1, 1, 0]
15 | ]
16 | springs = []
17 | for i in range(n_mass):
18 |     for j in range(i+1, n_mass):
19 |         if links[i][j]:
20 |             new_spring = Spring(body[i], body[j])
21 |             springs.append(new_spring)
22 |             body[i].add_spring(new_spring)
23 |             body[j].add_spring(new_spring)

```

После задания матрицы связей в дополнение к списку шариков создается список `springs` – пружин, их соединяющих (строки 16–23). Пружина, созданная в случае ненулевого значения в матрице связей, добавляется и в общий список пружин (строка 21), и в список пружин каждого из двух шариков, которые она соединяет (строки 22–23).

Переходим к анимации движения системы под действием силы тяжести (фрагмент кода 32).

В строках 1–4 создаются два объекта для графиков зависимости энергии и положения центра масс от времени и отображаются первые точки этих графиков, соответствующие начальному состоянию системы. Строка 6 служит для организации паузы в работе программы – ожидания нажатия клавиши мыши или клавиатуры пользователем.

Фрагмент кода 32. Анимация

```

1 | f1 = gcurve(color=color.red)
2 | f2 = gcurve(color=color.blue)
3 | f1.plot([t, energy(body, springs)])
4 | f2.plot([t, com_y(body)])
5 |
6 | scene.waitfor('click keydown')
7 |

```



```

8 display = 0
9 while t<T:
10     t += DELTA_T
11     for m in body:
12         m.v += m.a()*DELTA_T
13     for m in body:
14         m.pos += m.v*DELTA_T
15         if m.pos.y<ground.pos.y+ground.size.y:
16             m.v.y=abs(m.v.y)
17     display += 1
18     if display == 20:
19         display = 0
20         for m in body:
21             m.redraw()
22         for s in springs:
23             s.redraw()
24         f1.plot([t, energy(body, springs)])
25         f2.plot([t, com_y(body)])

```

Переменная `display` используется для того, чтобы достаточно медленные операции – рисование трехмерных объектов и постановка точек на графике – выполнялись не на каждом шаге интегрирования по времени, а, например, на каждом двадцатом шаге. Что касается метода интегрирования, то в данном фрагменте приведен вариант интегрирования методом Эйлера-Кромера, когда сначала вычисляются ускорения всех шариков, затем определяются их скорости, а потом уточняются положения (строки 11–14). В строках 15–16 корректируются скорости шариков, которые достигли уровня земли: вертикальная компонента вектора скорости меняет знак.

Для данной реализации класса `Mass` метод Эйлера-Кромера реализуется наиболее компактно. Для того, чтобы получить схему интегрирования классическим методом Эйлера недостаточно поменять местами набор строк 13–16 со строками 11–12. Дело в том, что при такой замене скорости будут находиться на основе ускорений, вычисленных в момент времени  $t + \Delta t$ , а не в момент времени  $t$ , как требует классический вариант. Для его реализации в класс `Mass` нужно добавить дополнительное поле, например, `acc`, для сохранения значений ускорения данного шарика, инициализировав его, как и скорость  $v$ , нулевым вектором.

Строки 11–16 фрагмента кода 32 нужно переписать в виде:

```
for m in body:
    m.acc = m.a()
for m in body:
    m.pos += m.v*DELTA_T
    if m.pos.y<ground.pos.y+ground.size.y:
        m.v.y=abs(m.v.y)
for m in body:
    m.v += m.acc*DELTA_T
```

Сохранение ускорений понадобится и при реализации метода аппроксимации по срединной точке. Это нужно сделать, чтобы избежать их повторного пересчета. Строчки 11–16 в этом случае следует переписать следующим образом:

```
for m in body:
    m.acc = m.a()
    m.v += m.acc*DELTA_T
for m in body:
    m.pos += m.v*DELTA_T+0.5*m.acc*DELTA_T**2
    if m.pos.y<ground.pos.y+ground.size.y:
        m.v.y=abs(m.v.y)
```

В качестве иллюстрации к приведенным фрагментам кода ниже на рис. 42 приведено изображение системы, полученное путем интегрирования методом Эйлера-Кромера с шагом  $\Delta t = 10^{-5}$  с в момент времени  $t = 1.2$  с.

На рис. 43 приведено сравнение графиков зависимости энергии (верхняя линия) и положения центра масс системы (нижняя линия) для трех методов интегрирования, выполнявшегося с шагом  $\Delta t = 10^{-4}$  с. Видно, что в данной задаче метод Эйлера-Кромера лучше сохраняет энергию системы, а метод аппроксимации по срединной точке более надежно определяет положение точек системы.

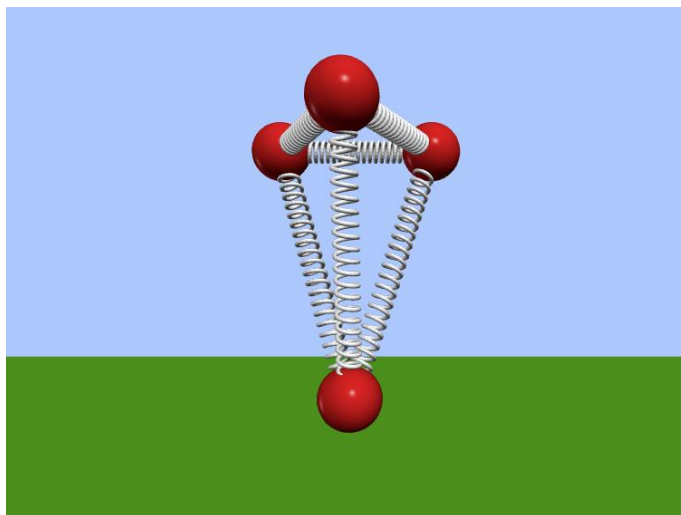


Рис. 42. Форма тела в момент времени  $t = 1.2$  с

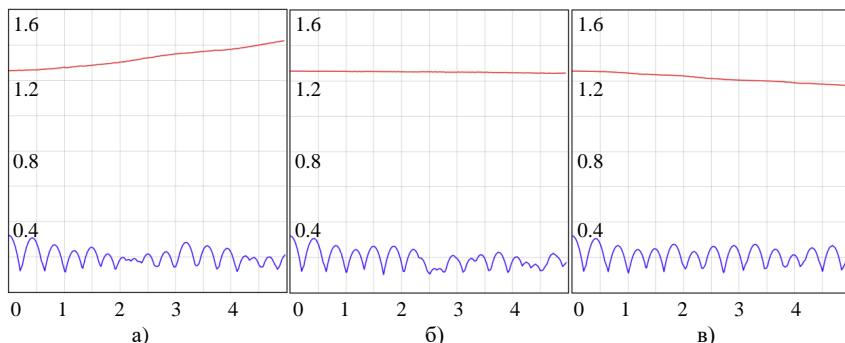


Рис. 43. Зависимость энергии и положения центра масс системы от времени: а) метод Эйлера, б) метод Эйлера-Кромера, в) метод аппроксимации по срединной точке

## ЛИТЕРАТУРА К ГЛАВЕ 4

- 4.1. Асланов В. С., Алексеев А. В. Концепции математического моделирования механических систем и процессов. – Самара: Изд-во Самарского университета, 2017. – 128 с.
- 4.2. Биргер И. А., Шорр Б. Ф., Иосилевич Г. Б. Расчет на прочность деталей машин: Справочник. – М.: Машиностроение, 1993. – 640 с.

- 4.3. Забавникова Т.А. Масс - пружинные модели физики твердого тела в МАТЛАБ R19b и Simulink [Электронный ресурс] – URL: <https://hub.exponenta.ru/post/mass-pruzhinnye-modeli-fiziki-tverdogo-tela-v-matlab-r19b-i-simulink567> (дата обращения 01.02.2021)
- 4.4. Николаев С. Н. Нелинейная система масс-с-пружинками для моделирования больших деформаций мягких тканей // Научно-технический вестник информационных технологий, механики и оптики. – 2013, № 5 (87). – С. 88–93.
- 4.5. Страшнов Е. В., Торгашев М. А., Тимохин П. Ю. Моделирование пружин в системах виртуального окружения с помощью метода мягких ограничений // Информационные технологии и вычислительные системы. – 2017. – Выпуск 3. – С. 70–78.
- 4.6. Сусликов П. И., Фроловский В. Д. Модели, методы, алгоритмы построения квазиразверток поверхностей // Евразийский Союз Ученых = Eurasian Union of Scientists. – 2015. – № 4 (13), ч. 5. – С. 54–56.
- 4.7. Blickhan R. The spring-mass model for running and hopping // J. Biomechanics. – 1989. – Vol. 22, № 11/12. – Pp. 1217–1227. doi: 10.1016/0021-9290(89)90224-8
- 4.8. De la Cruz S. T., Rodríguez M. A., Hernández V. Using Spring-Mass Models to Determine the Dynamic Response of Two-Story Buildings Subjected to Lateral Loads [Электронный ресурс] // Proceedings of the 15th World Conference on Earthquake Engineering, Lisbon, 2012. – Vol. 31. – Pp. 24719–24726. URL: <http://toc.proceedings.com/24574webtoc.pdf> (дата обращения 20.02.2021)
- 4.9. Differential Equation: Modeling: Example: Spring Mass [Электронный ресурс]. – URL: [http://www.sharetechnote.com/html/DE\\_Modeling\\_Example\\_SpringMass.html](http://www.sharetechnote.com/html/DE_Modeling_Example_SpringMass.html) (дата обращения 01.02.2021)
- 4.10. Jarrousse O. Modified Mass-Spring System for Physically Based Deformation Modeling. KIT Scientific Publishing, 2012. 222 p. [Электронный ресурс]. – URL: <https://www.ksp.kit.edu/9783866447424> (дата обращения 01.02.2021)
- 4.11. Pravdin S., Ushenin K., Sozykin A., Solovyova O. Human heart simulation software for parallel computing systems // Procedia Computer Science. – 2015. – Vol. 66. – Pp. 402–41. doi: 10.1016/j.procs.2015.11.046

## ГЛАВА 5. ПРОЕКТНОЕ ЗАДАНИЕ «НЕБЕСНАЯ МЕХАНИКА»

### ФОРМУЛИРОВКА ЗАДАНИЯ

В научно-фантастическом романе китайского писателя Лю Цысиня «Задача трех тел» [5.8] описана жизнь цивилизации, возникшей на планете, находящейся в сфере действия сразу трех солнц. Если, конечно, это можно назвать жизнью: происходящее, скорее, напоминает непрекращающуюся череду катастроф. Жители Трисоляриса в книге представляют собой серьезную угрозу человечеству, потому что обладают супер-технологиями и собираются покинуть свой беспокойный мир, а через 450 лет долететь до Земли и всех на ней покорить. Однако в рамках проектного задания придется решать задачу не борьбы с трисолярианцами, а оказания им помощи – поиска путей обеспечения кратковременной стабильности их солнечной системы. Может быть, тогда они передумают и к нам не полетят?

Итак, задача: на чертеже в таблице 6 представлена схема положения планеты Трисолярис и окружающих ее звезд в некоторый момент времени. Считая массы звёзд равными массе Солнца, а массу планеты равной массе Земли, требуется подобрать расстояние  $a$  и начальные скорости всех четырех объектов так, чтобы астрономическая система существовала без катастроф не менее ста земных лет. Под катастрофами (для планеты) в этой системе будем понимать следующие ситуации:

- приближение планеты к любому из солнц на расстояние, меньшее, чем  $0.1 \text{ а.е.}^{20}$  (слишком жарко!);
- удаление планеты от каждого из солнц на расстояние, большее чем  $30 \text{ а.е.}$  (слишком холодно!);
- сближение солнц на расстояние, меньшее, чем  $0.01 \text{ а.е.}$  (столкновение!) и удаление любого из солнц от остальных объектов системы на расстояние, превышающее  $100 \text{ а.е.}$ ;

---

<sup>20</sup> Астрономическая единица; величина, исторически полагавшаяся равной среднему расстоянию от Земли до Солнца. Формальное значение (с 2012 года):  $149\,597\,870\,700$  метров.

- увеличение полной энергии системы более, чем в полтора раза (строго говоря, это уже не физика, а проблемы с численным методом, но все равно – катастрофа).

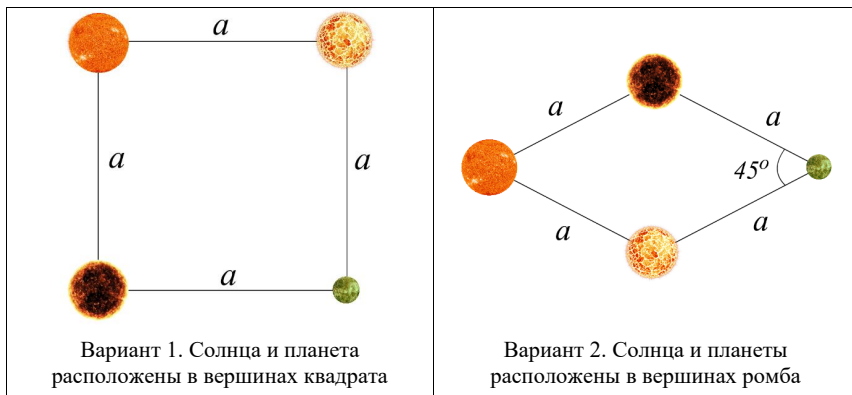
Поскольку масса планеты существенно меньше массы солнц, вычисления можно проводить по упрощенной схеме: на каждом шаге по времени при расчете положения солнц не учитывать влияние планеты, а затем корректировать положение планеты с учетом изменившегося вследствие перемещения солнц гравитационного поля.

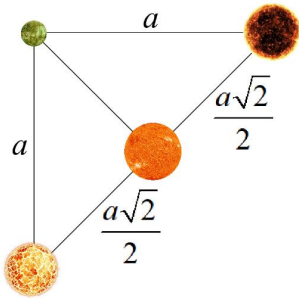
Первичное моделирование динамики системы проводить методом Эйлера–Кромера на промежутке в 25 лет. Шаг интегрирования по времени не должен превышать земных суток. Окончательное моделирование на промежутке в сто лет должно быть проведено методом Эйлера–Кромера и методом Верле.

Требуется описать зависимость результатов расчетов от выбора метода и шага интегрирования. Остается ли система стабильной, если шаг интегрирования станет равным одной неделе? одному месяцу? Как шаг интегрирования выбранных методов влияет на погрешность определения энергии системы?

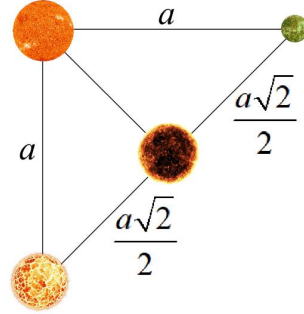
Попробуйте применить для моделирования классический метод Эйлера. Удастся ли подобрать шаг интегрирования по времени, чтобы система существовала без катастроф в течение года, десяти лет, ста лет?

Таблица 6. Варианты проектного задания

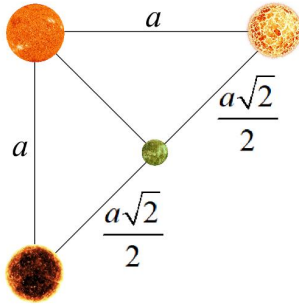




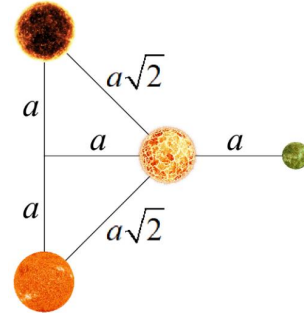
Вариант 3. Солнца и планета расположены в вершинах и на середине гипотенузы треугольника



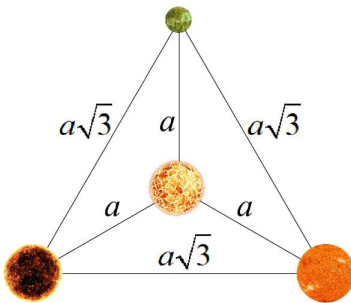
Вариант 4. Солнца и планета расположены в вершинах и на середине гипотенузы треугольника.



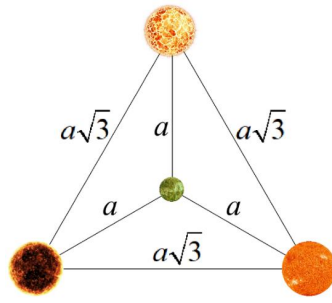
Вариант 5. Солнца и планета расположены в вершинах и в середине гипотенузы равнобедренного треугольника.



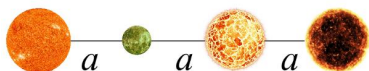
Вариант 6. Солнца расположены в вершинах равнобедренного прямоугольного треугольника, планета – в этой же плоскости.



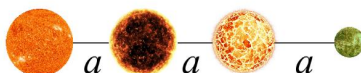
Вариант 7. Солнца и планета расположены в вершинах и центре равностороннего треугольника.



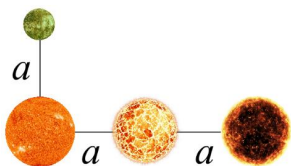
Вариант 8. Солнца и планета расположены в вершинах и центре равностороннего треугольника.



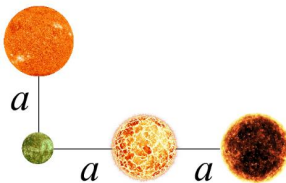
Вариант 9. Сизигия: солнца и планета лежат на одной прямой.



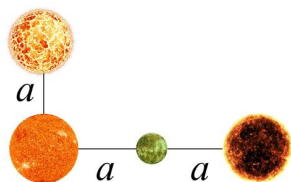
Вариант 10. Сизигия: солнца и планета лежат на одной прямой.



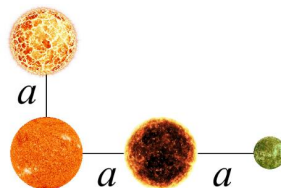
Вариант 11. Солнца и планеты лежат в одной плоскости (буква «Г»).



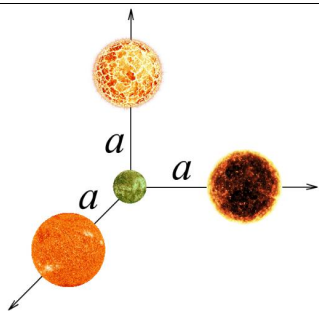
Вариант 12. Солнца и планеты лежат в одной плоскости (буква «Г»).



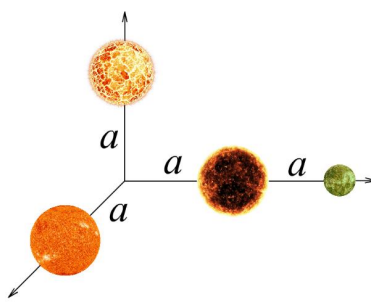
Вариант 13. Солнца и планеты лежат в одной плоскости (буква «Г»).



Вариант 14. Солнца и планеты лежат в одной плоскости (буква «Г»).

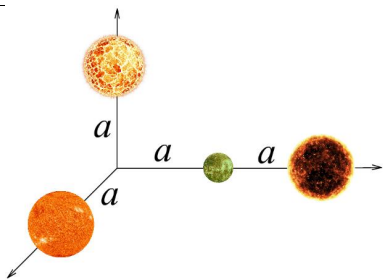


Вариант 15. Пространственное расположение. Планета – в центре, солнца – на координатных осях.

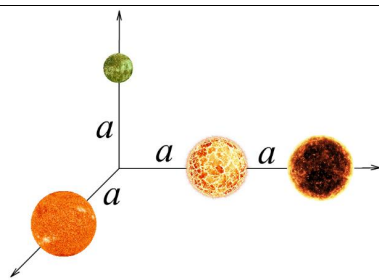


Вариант 16. Пространственное расположение. Солнца и планета лежат на координатных осях.

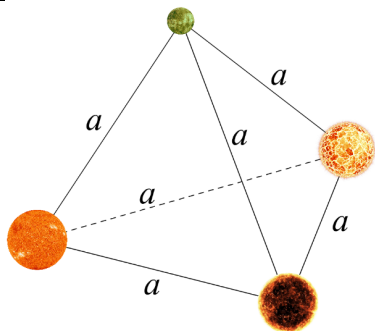




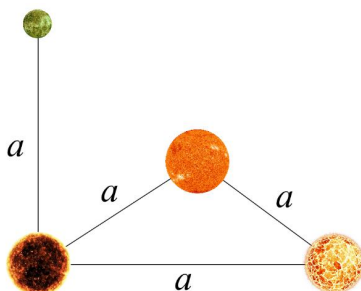
Вариант 17. Пространственное расположение. Солнца и планета лежат на координатных осях.



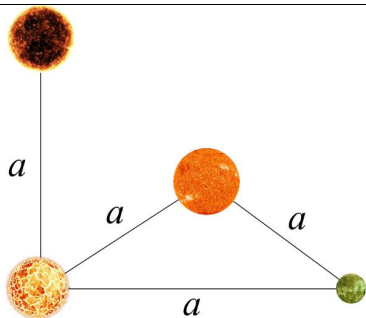
Вариант 18. Пространственное расположение. Солнца и планета лежат на координатных осях.



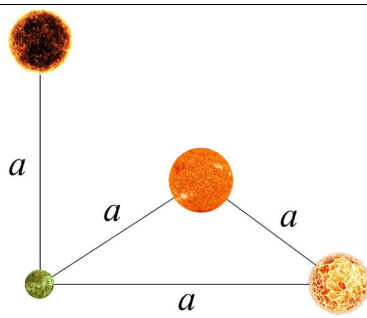
Вариант 19. Пространственное расположение. Солнца и планета лежат в вершинах тетраэдра.



Вариант 20. Пространственное расположение. Солнца лежат в вершинах правильного треугольника.



Вариант 21. Пространственное расположение. Два солнца и планета лежат в вершинах правильного треугольника.



Вариант 22. Пространственное расположение. Два солнца и планета лежат в вершинах правильного треугольника.

## БАЗОВЫЕ СВЕДЕНИЯ

Этот раздел начнем с цитаты из ставшей уже классической книги Г.А.Чеботарева «Аналитические и численные методы небесной механики»: *Небесная механика – раздел астрономии, изучающий движение тел солнечной системы в гравитационном поле. Движение Солнца, звезд и звездных систем изучает звездная астрономия. Так как расстояния между телами солнечной системы очень велики по сравнению с размерами самих тел, то все тела солнечной системы можно рассматривать как материальные точки, притягивающие друг друга по закону Ньютона* [5.11].

Задача повышения точности прогнозирования движения небесных тел является основной задачей небесной механики от момента ее возникновения и до наших дней. Не случайно, что многие идеи и методы вычислительной математики своим происхождению обязаны задачам именно небесной механики. Современный космический эксперимент, а также новые способы получения наблюдений предъявляют новые высокие требования к точности вычисления пространственных положений небесных тел. Основным аппаратом для этого по-прежнему являются численные методы решения дифференциальных уравнений движения этих тел. В этой связи особого внимания заслуживают работы [5.1, 5.3, 5.6], которые формально являясь монографиями по небесной механике, могут служить серьезными учебниками по численным методам. Если говорить о работах более образовательного характера, то познакомиться с различными подходами к компьютерному моделированию движений космических тел можно, например, с помощью учебных пособий [5.4, 5.5, 5.10].

Численное моделирование является мощным средством для исследования орбит небесных тел. Расчеты позволяют прогнозировать движение небесных тел и выявлять обстоятельства их происхождения в прошлом, проектировать космические миссии и оценивать вероятности столкновения астероидов с планетами, предсказывать с высокой точностью затменные явления [5.1].

Количество приложений этого моделирования только увеличивается. В последнее время к этому добавились задачи определения местоположения наземных объектов по навигационным спутниковым системам. Например, в [5.7] проведен сравнительный анализ численных методов интегрирования уравнений движения навигационных спутников ГЛОНАСС. Особое

внимание в сравнительном анализе уделено тому, как рассматриваемые схемы ведут себя в условиях действия на правую часть уравнений движения скачкообразных возмущений, например, соответствующих воздействию на спутник радиационного давления солнечного излучения в момент входа навигационного спутника в тень Земли и выхода из неё.

Даже знакомство с оглавлениями упомянутых выше книг дает представление о разнообразии и широте задач, связанных с моделированием движения небесных тел и систем. В этом пособии мы немножко прикоснемся только к одной из таких задач.

Движение планет или небесных тел – очень интересный объект исследования. Наблюдения за движением планет привели к открытию законов Кеплера, которые, строго говоря, справедливы только для системы двух тел (планеты и Солнца). Тот факт, что эти эмпирические законы удалось объяснить с помощью теории гравитации Ньютона, подтвердил справедливость и универсальность классической теории гравитации. Однако другие задачи, кроме задачи о гравитационном взаимодействии двух тел, так и не нашли аналитических решений в общем случае. Простые системы из трех частиц встречаются практически на всех уровнях физики: гелий (два электрона плюс ядро), вода (один атом кислорода и два атома водорода), планетарные системы, и даже триплеты галактик. Именно эти системы, являющиеся частью так называемых задач  $N$  тел, до сих пор являются благодатной почвой для численных исследований.

Что касается использования языка Питон в астрономических вычислениях, то следует, прежде всего, упомянуть проект *Astropy* [5.20], в рамках которого силами сообщества разрабатывается базовый пакет для астрономии, предназначенный для удобства использования, совместимости и взаимодействия между различными библиотеками Python для астрономических расчетов. Основной пакет *Astropy* содержит функции, предназначенные для профессиональных астрономов и астрофизиков, но может быть полезен всем, кто занимается разработкой программного обеспечения для астрономии. Проект *Astropy* также включает «аффилированные пакеты» на языке Питон, которые не обязательно разрабатываются основной командой разработчиков, но разделяют цели *Astropy* и часто строятся на основе кода и инфраструктуры основного пакета.

Одним из таких аффилированных пакетов является, например, система Gala [5.19], предназначенная для решения задач галактической динамики, т. е. для моделирования гравитирующих компонентов галактик, изучения их структуры и эволюции. Для большинства возникающих здесь задач требуются эффективные численные инструменты, многие из которых требуют одного и того же базового кода (например, для выполнения численного интегрирования орбиты). Питон при этом служит оберткой для высокопроизводительных алгоритмов на языках типа С, что обеспечивает сочетание скорости расчетов с гибкостью и простотой пользовательского интерфейса. API предоставляет основанный на классах интерфейс для быстрой (оптимизированной для С или Cython) реализации таких общих операций как оценка гравитационного потенциала и силы, расчет орбиты, динамические преобразования и индикаторы хаоса для нелинейной динамики.

В настоящем пособии мы не будем пользоваться специализированными пакетами. Несколько простейших задач планетарного движения будут решены стандартными (если не говорить о визуализации) средствами языка Питон.

**Примеры моделирования объектов солнечной системы.** Как уже было отмечено, основная причина планетарного движения – закон гравитации И. Ньютона. Сила, действующая на тело массы  $m$  со стороны тела массы  $M$ , прямо пропорциональна произведению масс и обратно пропорциональна квадрату расстояния между этими телами:

$$F = -G \frac{Mm}{r^2}, \quad (25)$$

где  $G$  – гравитационная постоянная, или постоянная Ньютона:

$$G = 6,674 \cdot 10^{-11} \text{ Н} \cdot \text{м}^2 \cdot \text{кг}^{-2} \quad (26)$$

Поле гравитационных сил имеет потенциал, очевидно равный

$$V = -G \frac{Mm}{r}. \quad (27)$$

Проиллюстрируем сказанное на примере движения Земли вокруг Солнца. В этом случае  $M \gg m$  и можно ограничиться приближением, когда Солнце неподвижно, а перемещается только Земля. Ее движение будет описываться уравнениями

$$\frac{d\vec{r}}{dt} = \vec{v}, \quad \frac{d\vec{v}}{dt} = -\frac{GM}{r^3}\vec{r}. \quad (28)$$

Здесь  $\vec{r}$  – радиус-вектор Земли в системе координат, начало которой расположено в центре Солнца. Обратите внимание, что масса Земли в уравнении ее движения не входит.

Чтобы посмотреть, как происходит такое движение, достаточно запустить программу, текст которой с небольшими изменениями перенесен из [5.21] и приведен во фрагменте кода 33.

Фрагмент кода 33. Движение Земли вокруг неподвижного Солнца

```

1  import vpython as vp
2
3  def leapfrog_dkd(diffeq, r0, v0, dt):
4      dt2 = dt/2.0
5      r1 = r0 + v0*dt2
6      v1 = v0 + diffeq(r1)*dt
7      r1 = r1 + v1*dt2
8      return r1, v1
9
10 def earth_acc(r):
11     s = vp.mag(r)
12     return -GM*r/(s*s*s)
13
14 r = vp.vec(1.0167, 0.0, 0.0)
15 v = vp.vec(0.0, 6.179, 0.0)
16
17 scene = vp.canvas(background=vp.vec(.2, .5, 1),
18                    range=2, autoscale=False)
19 scene.forward = vp.vec(0.2, 1, -0.2)
20 scene.camera.pos = vp.vec(-1, -3, 0.5)
21
22 earth = vp.sphere(pos=r, radius=0.1,
23                  make_trail=True,
24                  texture=vp.textures.earth,
25                  shining=0.1)
26 sun = vp.sphere(pos=vp.vec(0,0,0), radius=0.2,
27                 color=vp.color.yellow,
28                 emissive=True)
29 sunlight = vp.local_light(pos=vp.vec(0,0,0),
30                             color=vp.color.yellow)
31

```

```

32 GM = 4*3.1415926535**2
33 t, T, dt = 0.0, 100.0, 0.0005
34 while t<T:
35     vp.rate(1000)
36     r, v = leapfrog_dkd(earth_acc, r, v, dt)
37     earth.pos = r
38     t += dt

```

Результаты ее работы представлены на рис. 44.

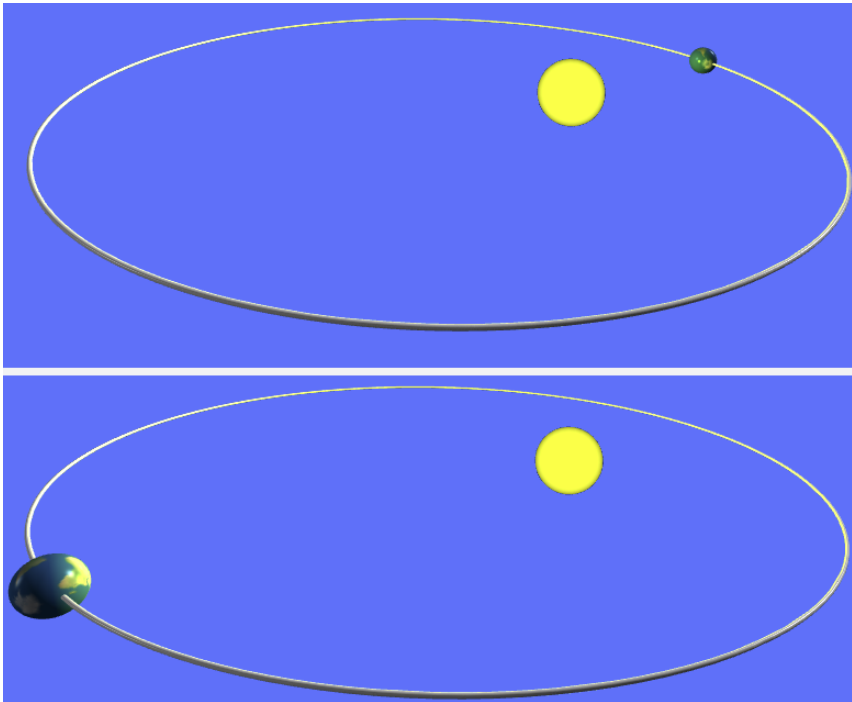


Рис. 44. Движение Земли вокруг Солнца

Приведенный код достаточно типичен для всех программ визуализации динамических процессов. Тем не менее остановимся на некоторых моментах подробнее. Прежде всего, заметим, что для интегрирования уравнений движения в этом случае выбран вариант метода с перешагиванием. Строки 5–7 функции `leapfrog_dkd` однозначно соответствуют трем формулам метода интегрирования (16) главы 3, т. е.

схеме drift-kick-drift. Входными параметрами для `leapfrog_dkd` является функция вычисления правых частей системы, возвращающая вектор ускорений. В случае ее применения к одному телу, это может быть объект типа `vector` пакета `vpython`, для нескольких тел такая функция должна возвращать массив `numpy`. Эти требования обеспечивают возможность выполнения векторных операций (сложения и умножения на числа) с возвращаемым результатом. В данной программе такой функцией служит `earth_acc` (строки 10–12), вычисляющая результат в соответствии со второй формулой в уравнениях (28).

Начальные данные для положения планеты и ее скорости (строки 14–15) соответствуют движению Земли в самой удаленной от Солнца точке (афелии), см. Табл. 7.

Таблица 7. Параметры планет<sup>21</sup> Солнечной системы

Планета	$e$	Орбитальный радиус, а.е.	Период обращения, земных лет	$r_{max}$ , а.е.	$v_{ap}$ , а.е./год	$\frac{M}{M_{\text{Солнца}}}$	$\frac{D}{D_{\text{Солнца}}}$
Меркурий	0.2056	0.3871	0.2408	0.4667	8.198	$1.660 \cdot 10^{-7}$	0.0035
Венера	0.0068	0.7233	0.6152	0.7282	7.338	$2.448 \cdot 10^{-6}$	0.0087
Земля	0.0167	1.0000	1.0000	1.0167	6.179	$3.004 \cdot 10^{-6}$	0.0091
Марс	0.0934	1.5237	1.8808	1.6660	4.635	$3.227 \cdot 10^{-7}$	0.0048
Юпитер	0.0484	5.2029	11.863	5.4547	2.624	$9.551 \cdot 10^{-4}$	0.1025
Сатурн	0.0539	9.5367	29.447	10.051	1.928	$2.859 \cdot 10^{-4}$	0.0861
Уран	0.0473	19.189	84.017	20.097	1.368	$4.355 \cdot 10^{-5}$	0.0364
Нептун	0.0086	30.070	164.79	30.329	1.136	$5.178 \cdot 10^{-5}$	0.0349
Плутон	0.2488	39.482	247.92	49.305	0.776	$6.582 \cdot 10^{-9}$	0.0017

В строках 17–20 происходит настройка области отображения графики; параметры `range`, `forward` и `camera.pos` подобраны для наиболее удачного отображения системы. Строго говоря, их можно вообще опустить, а потом подобрать наилучший для себя вид, используя интерактивные возможности VPython (вращать изображение можно при

<sup>21</sup> В 2006 году Плутон перестал считаться обычной планетой Солнечной системы и переведён в разряд карликовых планет

зажатой правой кнопке мышки, приближать или удалять – колесиком мышки или её перемещением при нажатой левой кнопке и клавише **Alt**, сдвигать влево/вправо и вверх/вниз – перемещением мышки при нажатой левой кнопке и клавише **Shift**).

Основные два отображаемые объекта – сферы `earth` и `sun` – создаются в строках 22–28. Радиусы этих сфер заданы условно, они не имеют ничего общего с радиусами небесных тел (реальный радиус Земли в 109 раз меньше радиуса Солнца). Сфера, изображающая Землю, покрыта одной из стандартных текстур пакета `vpython`; Солнце изображается сферой желтого цвета, внутри которой расположен источник цвета (строки 29–30), позволяющий имитировать день и ночь на Земле<sup>22</sup>.

Заметим, что в сети Интернет существует много бесплатных изображений, которые можно использовать в качестве текстур для планет солнечной системы. Использовать их можно двумя способами: просто загрузить файл и поместить в папку с программой (этот вариант подходит только для VPython 7, т. е. для локального использования), либо в параметре `texture` указать полный адрес файла. Последний вариант будет работать и для GlowScript VPython, но пригоден он только для изображений, расположенных на серверах, поддерживающих т. н. технологию CORS, например, <https://imgur.com>.

В строках 32–33 задаются параметры задачи: произведение гравитационной постоянной на массу Солнца ( $GM$ ), начальное и конечное время, шаг интегрирования. Физический смысл этих величин и их размерность будут уточнены ниже. Наконец, в цикле (строки 34–38) происходит расчет траектории методом перешагивания и корректируется положение сферы, изображающей Землю.

Желающие могут заменить в программе начальные данные Земли на данные любой планеты из Табл. 7 и убедиться, что во всех случаях орбита будет представлять собой замкнутую траекторию (возможно, для этого придется существенно увеличить время  $T$ , а также откорректировать радиусы сфер).

---

<sup>22</sup>Для этого нужно еще добавить отсутствующее в программе вращение Земли вокруг своей оси. В данной же модели «день» длится полгода.



Решение уравнения (28) может быть получено в явном виде; орбиты представляют собой замкнутые и незамкнутые кривые второго порядка. Случай замкнутых орбит схематически представлен на рис. 45. Эти решения описывают движение планеты по вокруг центра 0 (фокуса) по траектории в форме эллипса с полуосями  $a$  и  $b$  ( $b = a\sqrt{1 - e^2}$ ):

$$r = \frac{a(1 - e^2)}{1 - e \cos \varphi}.$$

Афелию ( $r = r_{max}$ ) соответствует значение  $\varphi = 0$ , перигелию ( $r = r_{min}$ ) – значение  $\varphi = \pi$ . Для нашей системы центр располагается практически в центре Солнца.

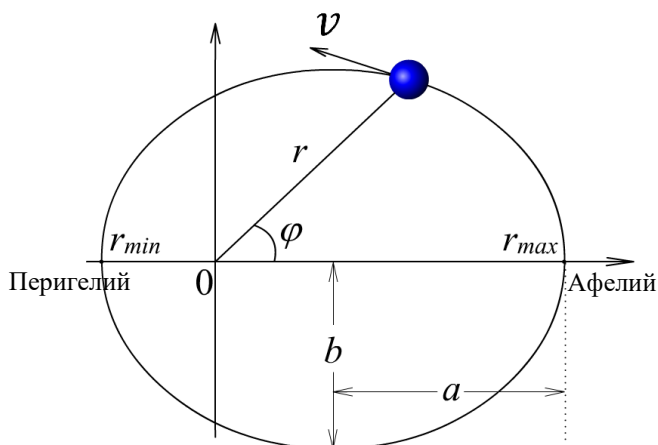


Рис. 45. Эллиптическая орбита

Радиальная скорость планеты ( $dr/dt$ ) обращается в ноль в двух точках траектории – максимально удаленной от центра (афелии) и максимально близкой к нему (перигелии). В этих точках скорость перпендикулярна оси  $x$  и может быть найдена из соображений сохранения энергии системы:

$$v_{af} = \sqrt{\frac{GM}{a} \frac{1 - e}{1 + e}}, \quad v_{пер} = \sqrt{\frac{GM}{a} \frac{1 + e}{1 - e}}.$$

Данные о геометрических характеристиках орбит Солнечной системы приведены в Табл. 7. В качестве единицы расстояния там выбрана астрономическая единица – большая полуось орбиты Земли, или другими словами, среднее расстояние от Земли до Солнца. В качестве единицы времени выбран один год, что соответствует 365.25 дням или, приблизительно,  $3.156 \cdot 10^7$  секундам. Единицей скорости, таким образом является величина 1 а.е./год, что соответствует 4740 м/с.

Осталось выбрать единицу массы. Как уже отмечалось выше, для моделирования часто нужно значение величины  $GM$ , где  $M$  – масса Солнца<sup>23</sup>. Если пренебречь массой Земли по сравнению с массой Солнца, т. е. считать  $M + m \approx M$ , и предположить, что Земля вращается вокруг Солнца по круговой орбите, то закон равенства центростремительного ускорения силе гравитации запишется в виде

$$m \frac{v^2}{r} = \frac{GMm}{r^2}. \quad (29)$$

Если теперь в качестве радиуса выбрать среднее значение  $r = 1$  а.е., в качестве периода обращения – 1 год, то угловая скорость Земли будет 1 а.е./год, а, значит линейная имеет значение  $2\pi$  а.е./год (или 29.77 км/с). Из соотношения (29) получаем

$$GM = v^2 r = 4\pi^2 \frac{\text{а.е.}^3}{\text{год}^2}.$$

Полученное равенство объясняет выбор значения для величины  $GM$  в строке 32 приведенного выше фрагмента кода 33.

Если пренебречь гравитационным влиянием планет Солнечной системы друг на друга, то для создания анимации, визуализирующей эти планеты, достаточно во фрагмент 33 добавить еще строки создания сфер для каждой из планет с соответствующими начальными данными и в цикл движения добавить вычисления координат каждой из них.

Ниже с использованием фрагментов кода симулятора  $N$  тел из статьи [5.18] будет построена модель Солнечной системы, все объекты которой – и планеты и Солнце – оказывают влияние друг на друга. В отличие от

---

<sup>23</sup> Часто обозначаемая  $M_{\odot}$

предыдущего примера вычисления будут существенно основаны на использовании массивов numpy и различных функций работы с ними.

Визуализация солнечной системы осложняется двумя обстоятельствами: размером объектов и расстояниями. Оба они делают практически нереальными те многочисленные красивые изображения, которые выдает любой поисковый сервис по запросу «солнечная система» или «solar system». Соотношения размеров планет и Солнца (не расстояний!) продемонстрированы на рис. 46. Видно, что если еще немного уменьшить масштаб, то Меркурий совсем исчезнет. Еще хуже ситуация с расстояниями. Диаметр Солнца примерно в сто раз меньше, чем расстояние от него до Земли, поэтому если мы захотим, например, чтобы на изображении шириной в тысячу пикселей присутствовали и Солнце, и Земля в реальном масштабе, то диаметр сферы, изображающей Солнце будет равен десяти пикселям, а Землю вообще будет не видно, потому что даже если удесятерить ее диаметр, то изображаться она будет ровно одним пикселем. Если просто увеличить все размеры в сто и более раз, получим еще одну неприятность: увеличенное изображение Солнца может поглотить Меркурий.

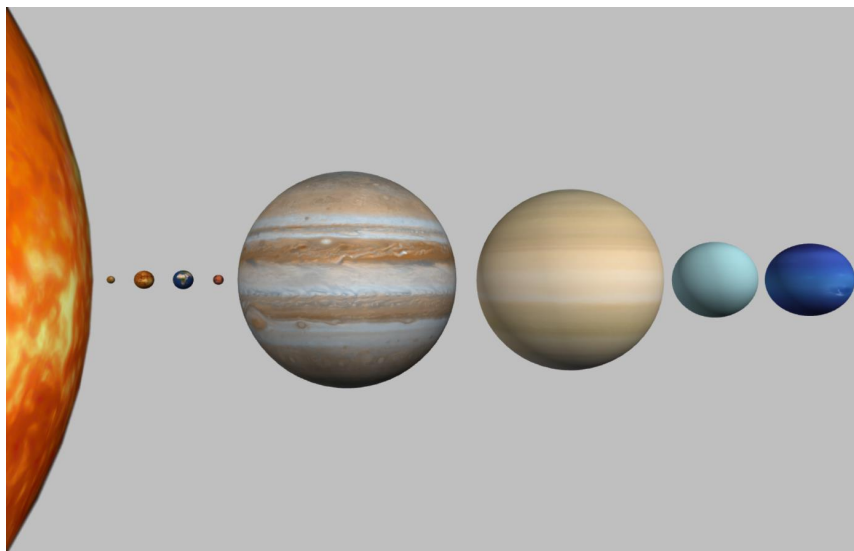


Рис. 46. Иллюстрация соотношения размеров планет Солнечной системы

Решение первой проблемы состоит в использовании массива подгоночных параметров – своего коэффициента масштабирования для каждой планеты. В приведенном ниже фрагменте кода 2 этот массив называется `SCALE`. Умножение радиусов Солнца и планет на эти коэффициенты обеспечивает приблизительно одинаковую видимость всех объектов за счет отказа от реального масштабирования.

Фрагмент кода 34. Параметры для визуализации объектов Солнечной системы

```
1 BODY = ['Солнце', 'Меркурий',
2         'Венера', 'Земля',
3         'Марс', 'Юпитер',
4         'Сатурн', 'Уран',
5         'Нептун']
6 MASS = [1.0, 1.66e-7,
7         2.448e-6, 3.004e-6,
8         3.227e-7, 9.551e-4,
9         2.859e-4, 4.335e-5,
10        5.178e-5]
11 D = [1.0, 0.0035,
12      0.0087, 0.0091,
13      0.0048, 0.1025,
14      0.0861, 0.0364,
15      0.0349]
16 SCALE = [55, 1500,
17          1000, 1000,
18          1500, 250,
19          250, 350, 350]
20 WEB_ADDR = 'https://i.imgur.com/'
21 IMAGES = ['XdRTvzj.jpg', 'SLgVbwD.jpg',
22          'YuK3CzJ.jpeg', 'MG7q4o6.jpg',
23          'Mwsa16j.jpeg', 'RMMtt0K.jpg',
24          '02Kt4gy.jpg', '2kZNVfw.jpg',
25          'lyLpoMk.jpeg']
26 texture = [WEB_ADDR+image for image in IMAGES]
27 texture = dict(zip(BODY, texture))
28 mass = dict(zip(BODY, MASS))
29 diameter = dict(zip(BODY, D))
30 scale = dict(zip(BODY, SCALE))
```

Несколько комментариев к приведенному коду: список `BODY` (строки 1–5) содержит названия моделируемых объектов, список `MASS` (строки 6–10) –

их массы в долях от массы Солнца, `D` (строки 11–15) – диаметры объектов, `SCALE` (строки 16–19) – список коэффициентов масштабирования для более наглядной визуализации, `IMAGES` (строки 21–25) – список имен файлов, расположенных на сайте `imgur.com`<sup>24</sup>, `WEB_ADDR` – строка с адресом сайта, общая для всех расположенных на нем файлов. В строках 26–30 создаются словари, позволяющие получить доступ к характеристикам объекта по его имени.

Пример кода во фрагменте 33 показывает, что для визуализации движения не обязательно создавать сложные классы; вполне можно обойтись и совсем без них. Тем не менее во многих случаях текст даже короткой программы может стать более удобочитаемым с их использованием. Ниже во фрагменте кода 35 приведен возможный вариант реализации такого класса, позволяющего создавать и отображать на экране текстурированную сферу. Реализация использует определенный во фрагменте кода 34 словарь `texture`. Что касается свойства (property) `r` этого класса, то оно означает радиус-вектор тела и введено для того, чтобы упростить доступ к сфере, изображающей тело: если, например, `earth` – объект (экземпляр класса) `Astro_Body`, то для его перемещения можно присвоить новое значение свойству `earth.r`, а не параметру содержащейся в нем сферы, `earth.ball.pos`. В последней строке кода выражение `vec(*value)` служит для «распаковки» массива координат `value` в перечень его элементов, из которых затем создается экземпляр класса `vector`.

Фрагмент кода 35. Вариант реализации класса `Astro_Body`

```
1 from vpython import *
2 import numpy as np
3 class Astro_Body:
4     def __init__(self, name, mass, diameter,
5                 pos=vec(0,0,0)):
6         self.name=name
7         self.mass=mass
8         self.diameter=diameter
9         self._r = np.zeros(3)
10        self.v = np.zeros(3)
11        self.a = np.zeros(3)
```

---

<sup>24</sup> Для их нахождения достаточно было в строке поиска на этом сайте ввести `vpython` или, например, `sun texture`

```

12         self.ball=sphere(radius = diameter/2,
13                             shininess=0)
14         self.ball.texture=texture[name]
15     @property
16     def r(self):
17         return self._r
18     @r.setter
19     def r(self, value):
20         self._r = value
21         self.ball.pos = vec(*value)

```

С использованием введенных выше словарей, констант и класса программа для построения изображения на рис. 46 примет вид, приведенный во фрагменте кода 36.

Параметр `emissive` у «планеты» с номером ноль, т. е. Солнца, установлен в **True** для имитации свечения – в этом случае поверхность достаточно яркая всегда, вне зависимости от количества падающего на нее света. Положение камеры выбрано так, чтобы в «объектив» попали все объекты, которые выстроены вдоль оси  $x$ . Отвечающая за это расположение переменная  $x$  корректируется на каждом шаге с учетом диаметров соседних планет. Эта величина, разумеется, не имеет ничего общего с реальными расстояниями.

Фрагмент кода 36. Визуализация реальных размеров тел Солнечной системы

```

1  scene = canvas(range=2, autoscale=False,
2                  background=color.gray(0.75))
3  scene.camera.pos = vec(0.75,0,0.25)
4  scene.forward = vec(0,0,-1)
5
6  planet = []
7  for name in BODY:
8      planet.append(Astro_Body(name, mass[name],
9                              diameter[name]))
10 planet[0].ball.emissive=True
11 x = 0.6
12 for i in range(1, len(planet)):
13     planet[i].r = np.array([x,0,0])
14     if i<len(planet)-1:
15         x += planet[i].diameter/2
16         x += planet[i+1].diameter/2 + 1.0e-2

```

Во фрагменте кода 37 расстояния между планетами уже более реальны – здесь предполагается, что все орбиты лежат в одной плоскости (в данном случае это будет плоскость  $Oxy$ ) и планеты размещены в своей точке афелия, т.е. на максимальном расстоянии от Солнца. Константа `D0_AU` представляет собой диаметр Солнца, выраженный в астрономических единицах<sup>25</sup>, в результате и диаметры всех планет тоже выражаются в них с учетом масштабирования (строка 15). Для задания точек афелия и скорости в афелии используются данные из таблицы 7. Из строчек 16 и 17 видно, что у каждой планеты ненулевой является только координата  $x$ , а у ее скорости – только компонента  $v_x$ . При использовании фрагмента кода 37 совместно с кодом 36 из последнего лучше убрать строки 1–4, чтобы в области рисования автоматически отобразились все планеты.

Фрагмент кода 37. Расстановка объектов Солнечной системы

```

1 | D0_AU = 0.00929826069
2 | R_MAX = [0.0,      0.4667,
3 |          0.7282,  1.0167,
4 |          1.6660,  5.4547,
5 |          10.051,  20.097,
6 |          30.329]
7 | r_max = dict(zip(BODY, R_MAX))
8 | V_AP = [0.0,      8.198,
9 |          7.338,  6.179,
10 |          4.635,  2.624,
11 |          1.928,  1.368,
12 |          1.136]
13 | v_ap = dict(zip(BODY, V_AP))
14 | for p in planet:
15 |     p.ball.radius *= D0_AU*scale[p.name]
16 |     p.r = vec(r_max[p.name], 0, 0)
17 |     p.v = vec(0, v_ap[p.name], 0)

```

К сожалению, выполнение фрагмента кода 37 показывает, что решение проблемы масштабирования диаметров все-таки не решило проблему расстояний: увидеть сразу все планеты достаточно четко не получается.

В качестве возможного варианта ниже будет использована функция масштабирования расстояний:

<sup>25</sup> AU – международное обозначение астрономической единицы

$$f = \begin{cases} r, & \text{если } r \leq 1, \\ \sqrt[3]{r}, & \text{если } r > 1. \end{cases}$$

где  $r$  – реальное расстояние от планеты до центра, а  $f$  – откорректированное расстояние для визуализации положения планеты. График зависимости  $f(r)$  приведен на рис. 47. Положение Меркурия и Венеры при таком преобразовании не меняется совсем, у Земли – почти не меняется; чем дальше планета от Солнца, тем больше коэффициент приближения, так что изначальный диапазон изменения расстояний  $[0, 30]$  трансформируется в отрезок  $[0, 3.1]$ .

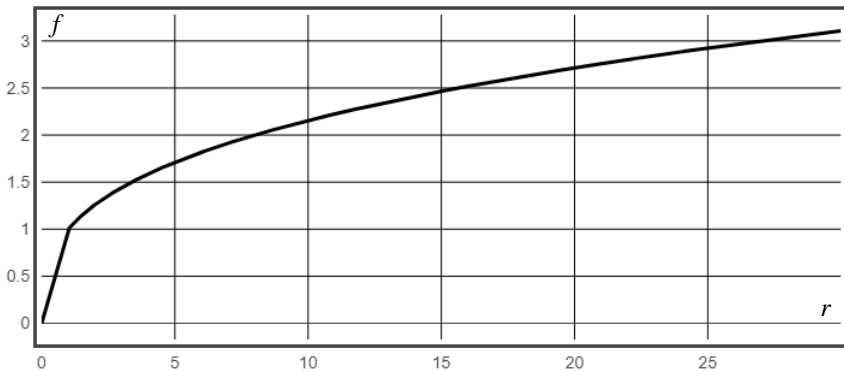


Рис. 47. Используемая функция масштабирования расстояний

Функция, масштабирующая исходный радиус-вектор (объект класса `vector`), приведена во фрагменте кода ниже.

Фрагмент кода 38. Масштабирование расстояний

```

1 def new_dist(r):
2     d = mag(r)
3     if d>1:
4         return d**(1/3)*norm(r)
5     else:
6         return r

```

Если ее использовать для корректировки положения планет, заменив строку 21 фрагмента кода 35 строчкой

```
self.ball.pos = new_dist(vec(*value))
```

то придем к начальному состоянию, изображенному на рис. 48. Еще раз отметим, что реальные положения планет, необходимые для организации



численного моделирования их движения, при этом сохраняются неизменными.



Рис. 48. Модель Солнечной системы после масштабирования размеров планет и расстояний между ними.

Переходим к вычислениям. Будем считать, что положения, скорости и ускорения объектов системы хранятся соответственно в массивах (матрицах)  $r$ ,  $v$ , а размера  $N \times 3$ , где  $N$  – количество объектов системы. При этом  $i$ -я строка матрицы соответствует  $i$ -му телу астрономической системы.

Формула для ускорения  $i$ -го тела, вызванного притяжением к остальным телам, имеет вид

$$\vec{a}_i = \sum_{j \neq i} G m_j \frac{\vec{r}_j - \vec{r}_i}{r_{ij}^3}, \quad (30)$$

где  $r_{ij} = |\vec{r}_j - \vec{r}_i|$ . Отсюда для первой компоненты вектора ускорения получаем

$$a_{ix} = \sum_{j \neq i} G m_j \frac{x_j - x_i}{r_{ij}^3}. \quad (31)$$

Выражения для  $a_{iy}$ ,  $a_{iz}$  получаются из (31) заменой в последнем  $x$  на  $y$  или на  $z$ . Функция для вычисления матрицы ускорений может быть записана в строгом соответствии с данными выражениями. Реализация такой функции приведена во фрагменте кода 39.

Безусловным достоинством функции `getAcc` во фрагменте кода 39 является ее наглядность: каждый компонент вектора ускорения  $i$ -го тела вычисляется по формуле типа (31); код практически не требует каких-то комментариев.

```

1 def getAcc(r, mass, G):
2     n = r.shape[0]
3     a = np.zeros((n,3))
4     for i in range(n):
5         for j in range(n):
6             if j!=i:
7                 dx = r[j,0] - r[i,0]
8                 dy = r[j,1] - r[i,1]
9                 dz = r[j,2] - r[i,2]
10                r_3 = (dx**2+dy**2+dz**2)**(-1.5)
11                a[i,0] += G*(dx*r_3)*mass[j]
12                a[i,1] += G*(dy*r_3)*mass[j]
13                a[i,2] += G*(dz*r_3)*mass[j]
14    return a

```

К сожалению, вычислительная эффективность приведенного кода не очень высока. А поскольку вычисление ускорений на каждом шаге интегрирования уравнений движения по времени является самой затратной по времени операцией, то необходимо принять все меры для ускорения этого процесса. Скорость вычисления ускорений можно существенно повысить, если использовать векторизацию, доступную в пакете `numpy`. Для этого задачу следует переформулировать в терминах матриц и постараться исключить явные циклы. Такой подход в случае языка Питон позволяет ускорить быстроедействие в несколько десятков раз. Обратной стороной повышения быстроедействия является необходимость хранить результаты промежуточных вычислений в виде матриц, что приведет к дополнительным затратам памяти. Для рассматриваемого количества объектов этим можно пренебречь.

Идея векторизации в данном случае состоит в следующем. Введем в рассмотрение следующие  $N$ -мерные векторы-столбцы:

- координат  $x$ ,  $y$  и  $z$  точек тел системы

$$\vec{x} = \begin{pmatrix} x_1 \\ x_2 \\ \dots \\ x_N \end{pmatrix}, \quad \vec{y} = \begin{pmatrix} y_1 \\ y_2 \\ \dots \\ y_N \end{pmatrix}, \quad \vec{z} = \begin{pmatrix} z_1 \\ z_2 \\ \dots \\ z_N \end{pmatrix},$$

- компонент ускорений, вычисляемых по формулам вида (31)

$$\vec{a}_x = \begin{pmatrix} a_{1x} \\ a_{2x} \\ \dots \\ a_{Nx} \end{pmatrix}, \quad \vec{a}_y = \begin{pmatrix} a_{1y} \\ a_{2y} \\ \dots \\ a_{Ny} \end{pmatrix}, \quad \vec{a}_z = \begin{pmatrix} a_{1z} \\ a_{2z} \\ \dots \\ a_{Nz} \end{pmatrix},$$

- масс всех тел системы

$$\vec{m} = \begin{pmatrix} m_1 \\ m_2 \\ \dots \\ m_N \end{pmatrix}.$$

Затем введем  $N \times N$  матрицы  $\Delta x$ ,  $\Delta y$ ,  $\Delta z$ , элементы которых определяются соотношениями:

$$\Delta x_{ij} = x_j - x_i, \quad \Delta y_{ij} = y_j - y_i, \quad \Delta z_{ij} = z_j - z_i,$$

где  $x_i, y_i, z_i$  – координаты  $i$ -го тела системы.

Теперь вычислить вектор-столбец  $\vec{a}_x$  можно в два этапа: сначала умножить каждый из недиагональных элементов матрицы  $\Delta x_{ij}$  на величину  $r_{ij}^{-3}$  (диагональные элементы так и останутся нулями), а затем умножить полученную матрицу  $N \times N$  на вектор  $\vec{m}$ . Именно такая схема реализации функции для вычисления ускорений всех тел, использующей векторизацию операций, приведена во фрагменте кода 40.

Здесь сначала (строки 2–4) создаются векторы столбцы  $x$ ,  $y$  и  $z$  как срезы массива  $r$  всех координат. Далее в строках 5–7 вычисляются матрицы  $dx$ ,  $dy$  и  $dz$  с использованием принятых в `numpy` принципов приведения размерностей: при вычитании из массива  $A$ , имеющего форму  $(1, N)$ , массива  $B$ , имеющего форму  $(N, 1)$  создается массив формы  $(N, N)$ , элементы каждой  $i$ -й строки которого получены поэлементным вычитанием из массива  $A$   $i$ -го элемента массива  $B$ . Поэтому значение элемента матрицы  $(x.T - x)[i, j]$  равно разности  $x[0, j] - x[i, 0]$ , т. е.  $x_j - x_i$ . Операция в строке 8 приводит к созданию матрицы квадратов расстояний между частицами: элемент  $r\_3[i, j]$  равен сумме  $(x_j - x_i)^2 + (y_j - y_i)^2 + (z_j - z_i)^2$ . Выполняемое в строке 9 поэлементное возведение всех таких величин, за исключением нулевых, в степень  $-1.5$ , создает упомянутую выше матрицу  $r_{ij}^{-3}$  (при  $i = j$  ее элементы равны нулю). Имя

переменной  $r\_3$ , как и во фрагменте кода 39, символизирует величину  $r^{-3}$ . Векторы компоненты ускорений получаются поэлементным перемножением матриц разностей координат и  $r\_3$ , умножением полученного результата на гравитационную постоянную, а затем – обычным умножением матрицы на вектор масс (строки 11–13). В строке 14 созданные таким образом векторы столбцы объединяются в единую матрицу.

Фрагмент кода 40. Оптимизированная функция вычисления ускорений

```

1  def getAcc2(r, mass, G):
2      x = r[:,0:1]
3      y = r[:,1:2]
4      z = r[:,2:3]
5      dx = x.T - x
6      dy = y.T - y
7      dz = z.T - z
8      r_3 = dx**2 + dy**2 + dz**2
9      r_3[r_3>0] = r_3[r_3>0]**(-1.5)
10
11     ax = G * (dx * r_3) @ mass
12     ay = G * (dy * r_3) @ mass
13     az = G * (dz * r_3) @ mass
14     a = np.hstack((ax, ay, az))
15     return a

```

Перед тем как переходить к моделированию движения планет, остановимся коротко на вычислении полной энергии системы – величине, которая должна оставаться постоянной, и контроль этого постоянства – важный элемент контроля качества вычислительной модели. Кинетическая энергия системы частиц имеет вид

$$E_{kin} = \sum_i \frac{m_i |\vec{v}_i|^2}{2}.$$

Потенциал поля гравитационных сил для двух частиц имеет вид (27); распространяя его на  $N$  частиц, получаем формулу

$$E_p = - \sum_{i<j} G \frac{m_i m_j}{r_{ij}}.$$

Выражение  $i < j$  показывает, что каждая пара частиц дает вклад в общую энергию только один раз.

Функция вычисления энергии приведена во фрагменте кода 41.

Фрагмент кода 41. Функция вычисления энергии

```
1 def getEnergy(r, v, m, G):
2     # Кинетическая энергия:
3     E_kin = 0.5*np.sum(m*v**2)
4     # Потенциальная энергия:
5     x = r[:,0:1]
6     y = r[:,1:2]
7     z = r[:,2:3]
8     dx = x.T - x
9     dy = y.T - y
10    dz = z.T - z
11    r_1 = np.sqrt(dx**2 + dy**2 + dz**2)
12    r_1[r_1>0] = 1.0/r_1[r_1>0]
13    E_p = -G*np.sum(np.triu((m*m.T)*r_1,1))
14    return E_kin, E_p
```

Здесь использованы те же идеи, что и во фрагменте кода 40. В строке 3 умножение массива масс, имеющего форму  $(N, 1)$  на массив квадратов компонент скоростей, имеющего форму  $(N, 3)$  выполняется по правилам приведения размерностей: поэлементно перемножаются столбцы. Функция `np.sum` вычисляет сумму всех элементов массива, получившегося в результате. Переменная `r_1` обозначает матрицу обратных расстояний между телами  $r_{ij}^{-1}$  (диагональные элементы, как и в предыдущем случае равны нулю). Матрица попарных произведений масс  $m_i m_j$  тоже создается «перемножением» одномерных массивов формы  $(N, 1)$  и  $(1, N)$ ; затем каждый элемент этой матрицы умножается на  $r_{ij}^{-1}$ . Функция `np.triu` возвращает верхнюю треугольную часть получившейся матрицы, чтобы в суммировании участвовали только элементы, удовлетворяющие упомянутому выше условию  $i < j$ .

Осталось только задать значение гравитационной постоянной. Напомним, что у нас в качестве единицы расстояния выбрана астрономическая единица, единицей времени является один год, а единицей массы – масса Солнца. Тогда значение (26), имеющее размерность  $\text{м}^3/(\text{кг}\cdot\text{с}^2)$  должно быть пересчитано следующим образом:

$$G = 6,674 \cdot 10^{-11} \frac{1.9885 \cdot 10^{30}}{149597870700^3} \cdot (365.25 \cdot 24 \cdot 3600)^2 \frac{a.e.^3}{M_{\odot} \cdot \text{год}}$$

что в использованных единицах дает значение 39.47694.

Фрагмент программы, визуализирующий движение планет Солнечной системы, приведен ниже.

Фрагмент кода 42. Настройка и запуск анимации движения планет

```
1 G = 39.47694 # гравитационная постоянная
2 N = 9        # количество небесных тел
3 t = 0        # текущее время
4 t_end = 10   # окончание моделирования (10 лет)
5 dt = 1/365.25/24 # шаг интегрирования (1 час)
6
7 gd = graph(width=800, height=300,
8           xtitle='<i>t</i>',
9           ytitle='<i>Энергия</i>')
10 PE = gcurve(width=2, color=color.black,
11            dot=True, dot_radius=4)
12 KE = gcurve(width=2, color=color.red,
13            dot=True, dot_radius=4)
14 TE = gcurve(width=2, color=color.blue)
15
16 r, v, a = [], [], []
17 for p in planet:
18     r.append(p.r)
19     v.append(p.v)
20     a.append(p.a)
21 p, v, a = np.array(p), np.array(v), np.array(a)
22 m = np.array(MASS).reshape(N,1)
23
24 while t < t_end:
25     rate(500)
26     v += a*dt/2.0
27     r += v*dt
28     for i in range(N):
29         planet[i].r = r[i]
30     a = getAcc2(r, m, G)
31     v += a*dt/2.0
32     e_kin, e_p = getEnergy(r, v, m, G)
33     t += dt
34     PE.plot(t, e_p)
35     KE.plot(t, e_kin)
36     TE.plot(t, e_p+e_kin)
```

Кроме задания параметров моделирования в этом фрагменте создается область отображения графиков зависимости кинетической, потенциальной энергии и их суммы от времени (строки 7–14). Затем на основе информации о начальном положении планет, их начальных скоростях и ускорениях формируются рабочие массивы этих величин размерности  $N \times 3$ .

Для интегрирования уравнений движения используется схема Kick-Drift-Kick, описанная ранее в главе 3, формулы (17). Ей соответствуют строки 26, 27, 30, 31 приведенного кода. В строках 28–29 осуществляется корректировка положений планет на основе вычисленных значений их координат. Кроме этого, на каждом шаге цикла вычисляются значения энергии и корректируется график, их отображающий.

**Задача трех тел.** В процессе своего бурного развития механика космического полета дала жизнь целому ряду задач, пленяющих своеобразной красотой и новизной [5.2]. Одной из них, безусловно, является ставшая классической задача трех тел.

Задача трех тел была сформулирована И. Ньютоном в 1687 году в его «Началах», где он рассматривал движение Земли и Луны вокруг Солнца. В этой задаче ключевым параметром выступало отношение массы Луны к массе Земли, равное 0.0123, что является небольшим, но не пренебрежимо малым числом. Ньютон смог получить приближительное решение, которое согласовывалось с наблюдениями с точностью до 8%.

Специальная форма общей задачи трех тел была изучена Л. Эйлером в 1767 г. Он рассмотрел три тела произвольной (конечной) массы, расположенные на одной прямой. Эйлер показал, что при подходящих начальных условиях эти тела всегда будут оставаться на одной прямой, которая, в свою очередь, будет вращаться вокруг центра масс тел. Движение самих тел будет периодическим, а их траектории – кеплеровскими орбитами: окружностями (в случае равных масс) или эллипсами (Рис. 49). К сожалению, позже выяснилось, что такое движение неустойчиво; малейшие возмущения нарушат симметрию, поэтому такая ситуация не может быть реализована в физической системе.

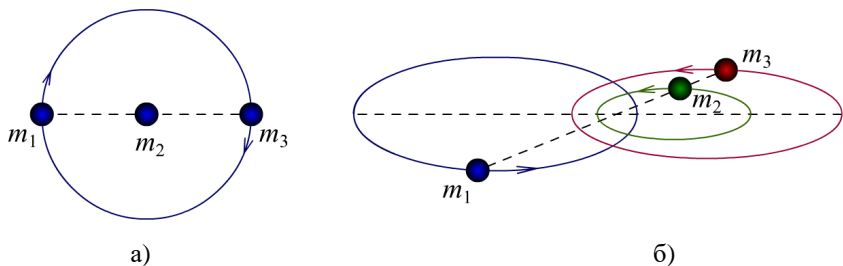


Рис. 49. Иллюстрация решения Эйлера. Три тела находятся на одной прямой и движутся по плоским эллиптическим орбитам: а) случай равных масс; б) массы тел различны.

Примерно в это же время, в 1772 г. Ж. Лагранж обнаружил еще один класс периодических орбит. Тела, расположенные в вершинах правильного треугольника, при определенных начальных условиях могут перемещаться по эллиптическим траекториям вокруг их центра масс так, что образуемый ими треугольник остаётся равносторонним (Рис. 50). В отличие от решения Эйлера, решение Лагранжа может при определенных условиях быть устойчивым, например, если одна из масс полностью доминирует в системе.

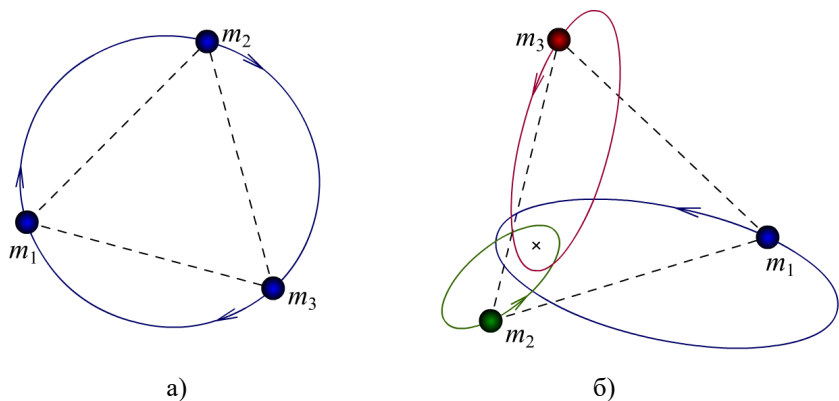


Рис. 50. Иллюстрация решения Лагранжа. Три тела находятся в вершинах равностороннего треугольника и движутся по эллиптическим траекториям, так что треугольник все время остается равносторонним:  
а) случай равных масс; б) массы тел различны.

22 февраля 1906 года немецким астрономом Максом Вольфом в обсерватории Хайдельберг, Германия, был открыт крупный астероид, движущийся вблизи орбиты Юпитера и образующий вместе с Солнцем и



Юпитером равносторонний треугольник. Таким образом, движение Лагранжа было открыто в природе спустя более ста лет после появления его математического описания. Найденный астероид получил имя Ахиллес. Впоследствии было открыто много астероидов, чье движение соответствует формулам Лагранжа. В астрономии сложилась традиция называть такие астероиды именами героев Троянской войны, а сам класс объектов — троянскими астероидами.

Решения Эйлера и Лагранжа теперь известны как частные решения общей задачи трех тел, уравнения которой имеют вид:

$$\frac{d}{dt^2} \vec{r}_1 = \vec{a}_{12} + \vec{a}_{13}, \quad \frac{d}{dt^2} \vec{r}_2 = \vec{a}_{21} + \vec{a}_{23}, \quad \frac{d}{dt^2} \vec{r}_3 = \vec{a}_{32} + \vec{a}_{31}, \quad (32)$$

где  $\vec{a}_{ij}$  – вектор ускорения, приобретаемого  $i$ -м телом под действием гравитационной силы со стороны  $j$ -го тела (другими словами, это сила притяжения, действующая со стороны  $j$ -го тела на  $i$ -е тело, деленная на массу  $i$ -го тела):

$$\vec{a}_{ij} = \frac{Gm_j}{|\vec{r}_i - \vec{r}_j|^3} (\vec{r}_i - \vec{r}_j). \quad (33)$$

При такой записи уравнений движения видно, что они сохраняют свою силу, даже если масса одного из тел пренебрежимо мала, т. е. мы можем положить ее равной нулю при анализе и проведении расчетов.

При моделировании движения объектов нашей солнечной системы возникают разные постановки задачи трех тел, например, для систем Солнце-планета-планета, Солнце-планета-луна или Солнце-планета-астероид. В последнем случае задача значительно упрощается, поскольку малая масса третьего объекта позволяет пренебречь его гравитационным влиянием на Солнце и на планету. Задача трех тел в такой постановке называется *ограниченной* задачей трех тел. Особо изученным является ее частный случай – *ограниченная круговая задача трех тел* – когда два тела с преобладающей массой движутся по окружностям вокруг их общего центра масс. Начало его подробному исследованию также положили Л. Эйлер и Ж. Лагранж, а дальнейший существенный вклад внесли К. Якоби (1836) и Дж. Хилл (1877). Последний, рассмотрев еще более частный случай, когда масса двух тел существенно меньше массы третьего, построил новый класс периодических движений. Почти через двести лет после формулировки

исходной проблемы И. Ньютоном, Дж. Хилл разработал новый подход к решению задачи о движении системы Солнце-Земля-Луна. Разработанная им теория с некоторыми уточнениями, сделанными в 1896 году Э. Брауном, продолжает активно использоваться в небесной механике.

Попытки построить общее решение задачи трех тел предпринимались почти до конца XIX века. В этой связи можно упомянуть, например, имена Ж. Д'Аламбера, Ш.-Э. Делоне, А. Клеро, П.-С. Лапласа и многих других. На задачу было потрачено много лет упорного труда, она вызвала поток блестящих идей и дала много ценных методов и результатов, но построить общее решение так и не удалось.

Крупнейшим достижением в небесной механике стали две монографии А. Пуанкаре: «Новые методы небесной механики» (1892–1899) и «Лекции по небесной механике» (1905–1910). В них он успешно применил результаты своих исследований к задаче о движении трёх тел, детально изучив поведение решения (периодичность, устойчивость, асимптотичность и т. д.). Обобщив теорему Г. Э. Брунса (1887), Пуанкаре доказал, что задача трёх тел принципиально не интегрируема, т. е. ее общее решение нельзя выразить через алгебраические или через однозначные трансцендентные функции координат и скоростей тел [5.9]. Кроме того, он показал, что орбиты системы трех тел чрезвычайно чувствительны к начальным данным.

Это открытие стало фундаментом современной теории хаотических систем. Оно в какой-то степени объясняет тот факт, что за три столетия после формулировки задачи И. Ньютоном никаких других (по сравнению с полученными Эйлером и Лагранжем) семейств периодических орбит так и не было найдено. Новые семейства стали находить, только начиная с 1970 года. Часть из них построена численно, часть обнаружена путем теоретического исследования уравнений.

В качестве примера приведем одно из наиболее известных решений, которому соответствует траектория в форме цифры «восемь». Оно было впервые получено численно К. Муром в 1993 году; строгое доказательство его существования приведено А. Шенсинером и Р. Монтгомери [5.12]. В отличие от других частных решений, оно существует только в случае, когда массы всех трех тел одинаковы. Тела движутся по одной и той же траектории в форме восьмерки, центр которой совпадает с центром масс системы (Рис. 51).

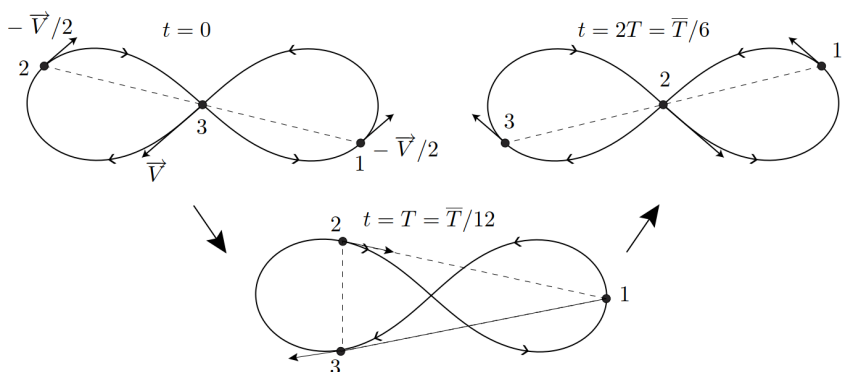


Рис. 51. Движение по «Восьмерке» в разные моменты времени.  
 $\bar{T} = 6.32591398$ . Рисунок из работы [5.12]

Начальные данные для такого движения, вычисленные К. Симо, и приведенные [5.12] содержатся в таблице 8.

Таблица 8. Начальные данные для траектории «Восьмерка»

	масса	$x$	$y$	$v_x$	$v_y$
Тело № 1	1	0.97000436	-0.24308753	0.466203685	0.43236573
Тело № 2	1	-0.97000436	0.24308753	0.466203685	0.43236573
Тело № 3	1	0	0	-0.93240737	-0.86473146

Это решение не стало столь же существенным продвижением теории трех тел, как полученные когда-то Эйлером и Лагранжем, но и оно дало определенный новый импульс ее развитию. В частности, его открытие привело к появлению термина *хореография N тел*, означающему решение задачи, описывающее периодическое движение, когда все тела проходят один и тот же путь в пространстве. Эти хореографии были изучены, например, в работах К. Симо, который обнаружил много маловероятных с точки зрения физической реализации, но чрезвычайно интересных орбит. Числовые данные и анимации некоторых из этих орбит приведены в [5.17].

Более подробно с историей задачи трех тел можно познакомиться, например, в [5.16].

Общая проблема трех тел использовалась при анализе различных задач астрономии и космических полетов, включая такие масштабные процессы как динамика галактик и звездообразование, а также для определения траекторий космических аппаратов, пилотируемых и роботизированных посадочных модулей. Различные техники решения задачи трех тел используются при отборе кандидатов в экзопланеты<sup>26</sup>.

Свидетельством непреходящей актуальности этой задачи может служить следующий факт: только за 2020 год в arXiv<sup>27</sup> разместили около шестидесяти статей, название или аннотация которых содержит словосочетание «three body problem». Тематика опубликованных работ очень разнообразна: использование нейро-сетевых технологий для проведения вычислений, анализ неустойчивостей и хаоса для конкретных систем, расчет орбит спутников Сатурна, решение задач квантовой механики и многое другое.

**Визуализация решения Эйлера задачи трех тел.** Следует отметить, что в ряде публикаций задачей Эйлера называют случай двух неподвижных тел, вокруг которых обращается третье тело, как правило существенно меньшей массы [5.22]. Здесь же будет рассмотрен именно тот вариант, который был описан в предыдущем разделе: в ходе движения три тела все время находятся на одной прямой. Для этого потребуется получить некоторые соотношения для расстояний между телами в начальный момент времени и правильно задать начальные скорости. После этого поведение трех тел будет промоделировано численно соответствии с полученными начальными данными.

В решении Эйлера тела совершают т. н. коллинеарное движение, т. е. в любой момент времени находятся на прямой линии. Будем считать, что тело  $m_2$  лежит между двумя другими телами,  $m_1$  и  $m_3$ . Каждое из тел движется по эллипсу вокруг центра масс системы с одинаковым периодом. Все тела достигают афелия – самой удаленной от центра точки – одновременно, при этом дистанция между телами самая большая. В момент прохождения перигелия эта дистанция становится наименьшей. В

---

<sup>26</sup> Планеты, находящиеся вне Солнечной системы

<sup>27</sup> www.arxiv.org – крупнейший бесплатный архив электронных публикаций научных статей и их препринтов по физике, математике, астрономии, информатике и биологии. Спонсируется и обслуживается Корнеллским университетом США, частью библиотеки которого и считается.

остальные моменты движения расстояние между телами осциллирует от одного экстремума к другому.

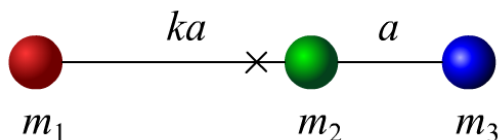


Рис. 52. Задача Эйлера: начальное положение тел.

Для описания теоретического результата, полученного Эйлером, можно перейти к рассмотрению движения во вращающейся системе координат. Именно в ней точки совершают только периодические движения от и навстречу друг другу вдоль соединяющей их прямой. Но чтобы определить начальные данные для такого решения, мы ограничимся лишь моментом времени  $t = 0$  и рассмотрим только одно состояние системы. Предположим, что тела стартуют с оси  $x$ , таким образом,  $y_i = 0$ , и нам нужно определить только горизонтальные координаты  $x_i$ . Чтобы осуществилось коллинеарное движение Эйлера, расстояния между телами должны соответствовать определенным пропорциям. Следуя работе [5.21], обозначим через  $a$  расстояние между телами  $m_2$  и  $m_3$ , а через  $ka$  – между  $m_1$  и  $m_2$  (Рис. 52). Таким образом

$$x_3 - x_2 = a, \quad x_2 - x_1 = ka, \quad x_3 - x_1 = (1 + k)a. \quad (34)$$

Уравнение движение заданного тела, например,  $m_1$ , имеет вид

$$\frac{d^2 x_1}{dt^2} = -m_2 \frac{x_1 - x_2}{|x_1 - x_2|^3} - m_3 \frac{x_1 - x_3}{|x_1 - x_3|^3},$$

где для удобства положили  $G = 1$ . Поскольку расстояния между точками то убывают, то снова увеличиваются, а, следовательно, носят колебательный характер, будем разыскивать решение в виде гармонической функции. Тогда  $d^2 x_1 / dt^2 = -\omega^2 x_1$ . Другим аргументом замены второй производной приведенным выражением является следующая идея: если тело массы  $m_1$  движется вокруг некоторого центра с угловой скоростью  $\omega$ , то действующая на него его центробежная сила  $x_1 m_1 \omega^2$  должна уравниваться силами притяжения к двум другим телам. Подставляя этот результат в предыдущее уравнение и деля обе части на  $-\omega^2$ , получаем

$$x_1 = \frac{1}{\omega^2} \left[ m_2 \frac{x_1 - x_2}{|x_1 - x_2|^3} + m_3 \frac{x_1 - x_3}{|x_1 - x_3|^3} \right]. \quad (35)$$

Аналогично для двух других тел получаем

$$x_2 = \frac{1}{\omega^2} \left[ m_1 \frac{x_2 - x_1}{|x_1 - x_1|^3} + m_3 \frac{x_2 - x_3}{|x_2 - x_3|^3} \right], \quad (36)$$

$$x_3 = \frac{1}{\omega^2} \left[ m_1 \frac{x_3 - x_1}{|x_3 - x_1|^3} + m_2 \frac{x_3 - x_2}{|x_3 - x_2|^3} \right]. \quad (37)$$

Подставляя соотношения (35)–(37) в равенства (34), после несложных преобразований приходим к следующему уравнению для нахождения параметра  $k$ :

$$k^5(m_2 + m_3) + k^4(2m_2 + 3m_3) + k^3(m_2 + 3m_3) - k^2(3m_1 + m_2) - k(3m_1 + 2m_2) - (m_1 + m_2) = 0. \quad (38)$$

Это уравнение имеет нечетный порядок, поэтому имеет хотя бы один вещественный корень. Согласно известному *правилу знаков Декарта* количество положительных корней многочлена с вещественными коэффициентами равно количеству перемен знаков в ряду его коэффициентов (или меньше на четное число). В уравнении (38) эта переменная ровно одна, поэтому и положительный корень только один. Например, если все массы одинаковые, то уравнение (38) приобретает вид

$$2k^5 + 5k^4 + 4k^3 - 4k^2 - 5k - 2 = 0,$$

и его корень, очевидно, равен единице. Этот факт будет использован в программе, где будем выбирать единицу в качестве начального приближения к корню для случая произвольных значений масс.

Расстояние  $a$  может быть выражено через найденное значение  $k$  из соотношения

$$a^3 = \frac{1}{\omega^2} \left[ m_2 + m_3 - \frac{m_1(1 + 2k)}{k^2(1 + k)^2} \right]. \quad (39)$$

Считая теперь  $a$  и  $k$  известными, приходим к следующим выражениям для определения координат тел в начальном состоянии:

$$x_2 = \frac{1}{\omega^2 a^2} \left[ \frac{m_1}{k^2} - m_3 \right], \quad x_1 = x_2 - ka, \quad x_3 = -\frac{m_1 x_1 + m_2 x_2}{m_3}. \quad (40)$$

Последнее из соотношений (40) показывает, что начало отсчета координаты  $x$  расположено в центре масс системы.

Что касается начальных скоростей, то их выбор должен удовлетворять условию равенства нулю полного импульса системы, т. е. выполнения условия

$$\sum_{i=1}^3 m_i v_{iy} = 0$$

(здесь учтено, что в начальный момент времени все скорости имеют только  $y$  – компоненту). С учетом последнего из соотношений (40) это условие будет удовлетворено следующим выбором:

$$v_{1y} = \beta x_1, v_{2y} = \beta x_2, v_{3y} = \beta x_3, \quad (41)$$

где  $\beta$  – произвольное число (масштабный коэффициент), определяющий энергию системы в начальный момент времени. Если  $\beta = \omega$ , то тела будут двигаться по окружностям, в остальных случаях траектории будут эллипсами.

Таким образом получаем следующую схему моделирования эйлеровского коллинеарного движения для заданного набора трех масс. Сначала определяем величину  $k$ , численно решая любым методом уравнение (38). Далее находим начальные данные по формулам (40), (41). Затем решаем уравнения (32) с этими начальными данными одним из известных численных методов. Текст программы, визуализирующей эйлеровское движение трех тел, приведен во фрагменте кода 43 (строго говоря, это даже не фрагмент, а работающая программа).

Фрагмент кода 43. Коллинеарное движение по Эйлеру

```
1 from vpython import *
2 from scipy.optimize import newton
3
4 def set_scene(r, R):
5     scene = canvas(width=600, height=600,
6                   background=color.white)
7     scene.forward = vec(0,0,-1)
8     body, vel = [], []
9     c = [color.red, color.green, color.blue]
10    for i in range(3):
11        body.append(
12            sphere(pos=r[i], radius=R,
13                  color=c[i],make_trail=1))
```

```

14         vel.append(
15             arrow(pos=body[i].pos,
16                  shaftwidth=R/4,color=c[i]))
17         line = curve(color=color.black)
18         com = sphere(pos=vec(0,0,0), radius=R/4)
19         return body, vel, line
20
21 def quintic(k): # Euler's quintic equation
22     return -m1-m2+k*(-3*m1-2*m2+k*(-3*m1-m2+k*(
23         m2+3*m3+k*(2*m2+3*m3+k*(m2+m3))))))
24
25 def init_cond(beta):
26     k = newton(quintic, 1.0)
27     a = (m2+m3- m1*(1+2*k)/(k*(1+k))**2)**(1/3)
28     x2 = (m1/(k*k)-m3)/(a*a)
29     x1 = x2 - k*a
30     x3 = -(m1*x1 + m2*x2)/m3
31     r = [vec(x1,0,0), vec(x2,0,0), vec(x3,0,0)]
32     v = [vec(0,beta*x1,0), vec(0,beta*x2,0),
33         vec(0,beta*x3,0)]
34     return r, v
35
36 def acc(r):
37     r12 = r[0]-r[1]
38     r13 = r[0]-r[2]
39     r23 = r[1]-r[2]
40     s12,s13,s23 = mag(r12), mag(r13), mag(r23)
41     a = [-m2*r12/s12**3 - m3*r13/s13**3,
42         m1*r12/s12**3 - m3*r23/s23**3,
43         m1*r13/s13**3 + m2*r23/s23**3]
44     return a
45
46 R = 0.1
47 m1, m2, m3 = 1, 2, 3
48 dt = 0.001
49 t, T = 0, 20
50 scale = 0.7
51
52 r, v = init_cond(scale)
53 body, vel, line = set_scene(r, R)
54
55 while t<T:
56     rate(1000)

```



```

57     a = acc(r)
58     for i in range(3):
59         v[i] += a[i]*dt
60         r[i] += v[i]*dt
61         body[i].pos = r[i]
62         vel[i].pos = body[i].pos
63         vel[i].axis = v[i]
64         vel[i].length = R*(1+3*mag(v[i]))
65     line.clear()
66     line.append(pos=[r[i] for i in [0,1,2]])
67     t = t + dt

```

Прокомментируем некоторые части приведенного кода.

Во второй строке импортируется функция `newton` пакета `scipy`, которая будет использована для численного решения уравнения (38). Функция, определяющая это уравнение, называется `quintic` и находится в строках 21–23.

Входными параметрами функции `set_scene(r, R)` (строки 4–19) являются список векторов – положений тел в начальный момент времени и радиус сферы, которая изображает тело. Функция используется для создания холста нужного размера и цвета фона. Кроме того, в ней создаются:

- `body` – список сфер заданного радиуса  $R$ , отвечающих за визуализацию тел; их цвета (красный, зеленый, синий) определяются списком `c`;
- `vel` – список стрелок, отвечающих за визуализацию скоростей тел в ходе движения. Стрелки привязаны к центрам сфер. Ориентация и размер стрелки в процессе вычислений корректируется в зависимости направления и величины вектора скорости;
- `line` – линия, которая в ходе движения будет соединять центры сфер, визуально подтверждая, что они остаются на одной прямой (или, наоборот, демонстрируя нарушение этого факта, например, при потере устойчивости)
- `com` – маленькая сфера, отображающая начало координат, которое в этой задаче является центром масс (Center Of Masses) системы.

Для генерирования «правильных», т. е. обеспечивающих коллинеарный характер движения, начальных условий – положений тел на

оси  $x$  и их скоростей – используется функция `init_cond(beta)` (строки 25–34). Параметр `beta` – масштабный коэффициент из соотношений (41). Значение параметра `k` находится путем решения уравнения (38) методом Ньютона (строка 26); в качестве начального приближения использовано значение, равное единице, что соответствует случаю равных масс. В расчетных формулах для величин  $a$  и  $x_2$  (соотношения (39) и (40)) использовано значение параметра  $\omega = 1$ ; это не нарушает общности, поскольку нас интересует только принципиальная возможность коллинеарного движения. Функция возвращает два списка: список начальных (трехмерных) радиусов-векторов тел и список начальных скоростей этих тел.

Функция `acc(r)` (строки 36–44) служит для вычисления ускорений тел. Она возвращает список векторов ускорений, вычисленных в соответствии с уравнениями движения (32).

В строках 46–50 задаются настраиваемые параметры: размер сферы для визуализации тел, массы тел, интервал и шаг интегрирования по времени, масштабный множитель.

В строке 51 в соответствии с заданными параметрами определяются начальные положения и скорости тел, а затем в строке 52 создается сцена – окно для отображения графики и списки объектов для визуализации тел на сцене.

Основной цикл моделирования и визуализации движения содержится в строках 54–66. В каждый очередной момент времени координаты и скорости объектов определяются по вычисленным ускорениям методом Эйлера-Кромера (строки 58–59). После этого корректируются положение сфер, изображающих тела (строка 60), а также положение, ориентация и размер стрелок, визуализирующих скорости). Формула для длины стрелки (строка 63) подобрана с учетом того, что точка привязки (основание) стрелки находится в центре сферы. На каждом шаге цикла линия, соединяющая центры тел, стирается, а потом заполняется очередными данными и, следовательно, отображается заново.

Результаты работы программы для значения  $\beta = 0.65$  в один из моментов времени приведены на рис. 53. Здесь тела совершили один полный оборота и заканчивают второй.

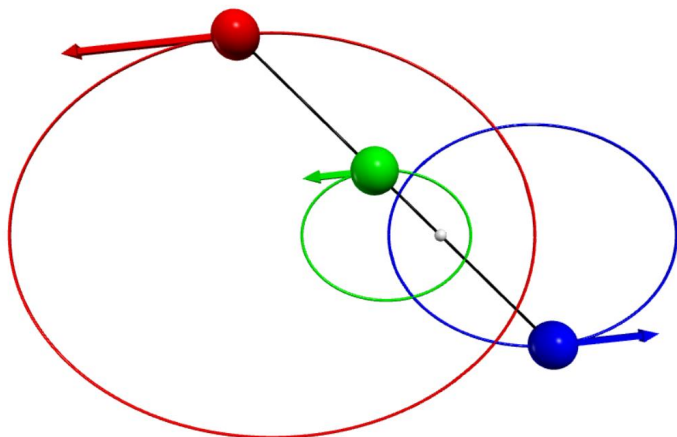


Рис. 53. Коллинеарное движение трех тел

Теоретически доказанная неустойчивость решения Эйлера и его высокая чувствительность к начальным данным может быть продемонстрирована этой же программой. Прежде всего, периодичность орбит сохраняется не больше трех-четырех циклов; их число зависит от начальной энергии и шага интегрирования по времени. Далее, как правило, два из трех тел притягиваются друг к другу и быстро сходят с периодических орбит, образуя что-то вроде бинарной системы.

Чтобы проследить, как задача реагирует на малые изменения параметров, изменим значение одной из скоростей на сотую долю процента, добавив после строки 51 следующий текст

```
v[0] *= 1.0001
```

При шаге интегрирования  $dt=0.0001$  достаточно быстро получим картину движения, изображенную на рис. 54. Коллинеарный характер движения нарушился до того, как тела успели совершить один полный оборот. Такая же ситуация возникает и при малом изменении начальных координат.

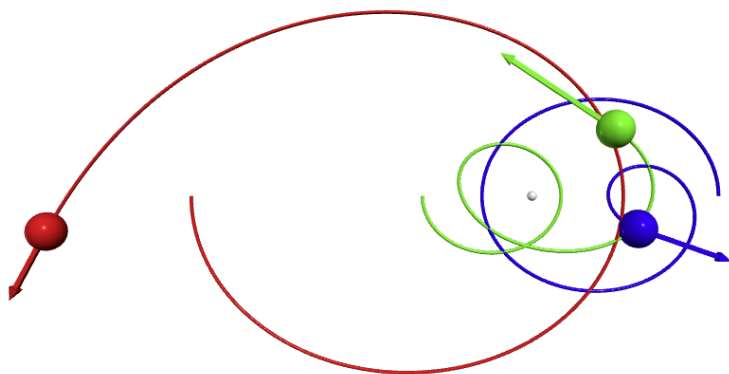


Рис. 54. Неустойчивость коллинеарного движения

## ПРИМЕР ВЫПОЛНЕНИЯ ЗАДАНИЯ

**Переборные задачи при моделировании астрономических систем.** Рассмотренные выше примеры моделирования астрономических объектов могут быть использованы в качестве основы для разработки и выполнения каждого индивидуального проекта. Что касается творческой, или зависящей от интуиции, составляющей работы, то ее продемонстрировать каким-то конкретным примером достаточно трудно. Конечно, возможен переборный вариант выполнения задания: задается некоторый конечный, но достаточно представительный набор векторов, содержащий, скажем, сто элементов, а затем полным перебором всех возможных сочетаний этих векторов в качестве начальных скоростей (т. е. перебором всех  $10^8$  вариантов) отбираются более-менее пригодные. Проблемы тут две. Прежде всего, проверка на пригодность не простой процесс, он тоже требует существенного времени, а значит для осуществления  $10^8$  проверок понадобятся слишком большие вычислительные затраты, даже учитывая мощности современных персональных компьютеров. Ну и во-вторых, составить удачный набор из ста векторов – такой же творческий и не формализуемый процесс, как и само исходное задание. Так что проще, наверное, посмотреть на поведение системы при каких-то конкретных значениях скоростей, а потом аккуратно

скорректировать эти скорости для обеспечения стабильности системы в смысле выполнения условий, сформулированных в задании.

Тем не менее, переборный подход совсем не является чем-то априори некорректным или недопустимым. Именно идея полного перебора использована, например, в работе [5.15] для нахождения периодических движений системы трех тел. И хотя постановка задачи в [5.15] отличается от сформулированной в проектом задании, полученные там результаты будут использованы ниже для демонстрации нескольких примеров численного решения задачи трех тел.

В работе [5.15] сформулирована задача поиска такого расположения трех гравитирующих тел, движение которых из положения покоя будет периодическим, т. е. через определенный промежуток времени тела вернуться в исходное состояние и их скорость в нем снова станет нулевой. Разумеется, условие периодичности гораздо более ограничительное, чем условие существования системы в течение достаточно короткого по астрономическим меркам отрезка времени, сформулированное в проектом задании. Однако алгоритм поиска таких решений оказался вполне реализуемым. Схема его такова.

Во-первых, установим количество неизвестных. Три тела определяют некоторую плоскость, и при отсутствии начальных скоростей их движение будет проходить в пределах этой плоскости. Задача становится двумерной. Выберем и зафиксируем далее любые два тела из трех, назовем их А и В и примем расстояние между ними за единицу расстояния. Оси координат в плоскости расположим так, чтобы координаты этих тел были  $(-0.5, 0)$  и  $(0.5, 0)$ , соответственно. В результате задача о поиске положения трех тел свелась к задаче поиска двух координат  $(x, y)$  третьего тела С в заданной плоскости (рис. 55).

Во-вторых, сузим область поиска неизвестных координат. В силу симметрии задачи можно ограничиться только одной четвертью плоскости, например первой  $(x > 0, y > 0)$ . Авторы [5.15] этим не ограничились: на основе анализа ряда публикаций они выбрали в качестве области возможных значений неизвестных расположенную в этой плоскости часть окружности единичного радиуса с центром в точке  $(-0.5, 0)$ .

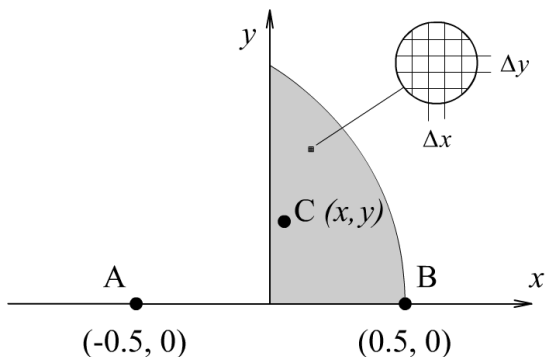


Рис. 55. Схема начального расположения трех тел

В-третьих, определимся с массами. За единицу массы примем массу тела А. В расчетах в работе [5.15] для масс тел В и С использовались значения 0.2, 0.4, 0.6, 0.8 и 1. В качестве значения гравитационной постоянной тоже использовалась единица. В качестве упражнения, полезно оценить, чему тогда соответствует единица времени, если тела А и В имеют массу Земли, а расстояние между ними равно одной астрономической единице. Или, наоборот, если единица времени – это одни сутки, то на каком реальном расстоянии в этом случае находятся тела А и В, имеющие массу, равную, массе Солнца?

Наконец, определим целевую функцию. Координаты  $(x, y)$  искомого тела С должны быть таковы, что для решения задачи Коши о движении гравитирующих тел в некоторый момент времени  $T$  должно обратиться в ноль выражение

$$\Psi(x, y; T) = \sum_{i=1}^3 |\vec{r}_i(T) - \vec{r}_i(0)| + \sum_{i=1}^3 |\vec{v}_i(T) - \vec{v}_i(0)|, \quad (42)$$

где  $\vec{r}_1, \vec{r}_2, \vec{r}_3$  и  $\vec{v}_1, \vec{v}_2, \vec{v}_3$  – радиус-векторы и скорости тела А, В, С соответственно.

Схема дальнейшего анализа была такой. Область поиска была разбита на квадраты сеткой с шагом  $\Delta x = \Delta y = 0.0001$  (см. рис. 55). Для положения тела С  $(x^*, y^*)$  в центре каждого квадрата задача трех тел решалась численно методом DOP853 на отрезке времени  $[0, 200]$ .

Рассмотренное положение отбиралось кандидатом для построения периодической орбиты, если существовало такое значение  $T_* \in (0.1, 200]$ , что  $\Psi(x_*, y_*, T_*) < 0.05$ . Далее найденное положение тела С уточнялось путем решения методом Ньютона уравнения  $\Psi(x, y, T) = 0$ , где в качестве начального приближения к решению выбирался набор  $x_*, y_*, T_*$ . Однако при этом существенно менялся способ вычисления функции  $\Psi(x, y, T)$ . Интегрирование задачи Коши осуществлялось специально разработанным методом сверхвысокой точности, так называемым CNS (clean numerical simulation). Его идея [5.14] состоит, с одной стороны, в использовании в разложениях неизвестных функций вида (7, Глава 3) от сорока до восьмидесяти слагаемых ряда Тейлора, а с другой стороны, проведения вычислений с использованием от 50 до 300 значащих цифр. Орбита считалась удовлетворяющей условию периодичности, если таким способом удавалось сделать значение функции  $\Psi$  меньшим, чем  $10^{-6}$ .

На основе такого поиска авторам удалось обнаружить 234 периодические орбиты, три из которых были известны ранее, а остальные являются новыми. Полученные результаты опубликованы в виде таблицы, содержащей информацию о массах тел, координатах тела С и периоде времени, после которого гравитирующие тела, перемещающиеся из состояния покоя, вернуться в исходное состояние.

### **Двумерная визуализация периодических траекторий.**

Переходим к визуализации. В качестве примера выберем строки с номерами 1 и 31 из таблицы данных [5.15] и поместим их в качестве первой и второй строки в Табл. 9 :

Таблица 9. Параметры системы трех тел

$m_1$	$m_2$	$m_3$	$x$	$y$	$T$
1	1	1	0.0207067154	0.3133550361	2.1740969264
1	0.8	0.8	0.0009114239	0.3019805958	1.8286248401

Начнем с первого случая. Поскольку задача является двумерной, то для изображения траекторий движения тел можно ограничиться их отрисовкой на плоскости объекта graph. Добавляя на график очередную точку, мы будем наблюдать изменение траектории. При этом очень

удобным является использование параметра `dot=True` объекта `gcurve`, позволяющего выделять кружком текущую строящуюся точку графика.

Что касается численного интегрирования, то попробуем поступить аналогично авторам работы [5.15]: используем метод DOP853, который является одним из реализованных в функции `solve_ivp` пакета `scipy`. Проинтегрируем уравнения движения на отрезке  $[0, T]$ , а потом отобразим полученное решение на графике.

Для использования функции `solve_ivp` уравнения движения (32) необходимо записать в виде одного векторного уравнения первого порядка. Это можно сделать разными способами, например так. Введем сначала векторы скорости  $\vec{v}_i = \frac{d}{dt} \vec{r}_i$ . Все используемые векторы в этом случае являются двумерными, поэтому объединим их в один 12-мерный вектор  $\vec{z}$  следующим образом:

$$\vec{z} = (r_{1x}, r_{2x}, r_{3x}, r_{1y}, r_{2y}, r_{3y}, v_{1x}, v_{2x}, v_{3x}, v_{1y}, v_{2y}, v_{3y}),$$

где  $r_{1x}$  – первая компонента радиус-вектора первой точки,  $v_{3y}$  – вторая компонента скорости третьей точки и т. д. Тогда первые шесть уравнений движения в терминах введенного вектора  $z$  примут вид

$$\frac{dz_k}{dt} = z_{k+6}, \quad k = 1 \dots 6.$$

Чтобы записать остальные, обозначим куб расстояния между  $i$ -м и  $j$ -м телами, т. е. величину  $|\vec{r}_i - \vec{r}_j|^3$  через  $s_{ij}$ . В терминах вектора  $\vec{z}$  эти величины примут вид

$$s_{12} = ((z_2 - z_1)^2 + (z_5 - z_4)^2)^{3/2},$$

$$s_{13} = ((z_3 - z_1)^2 + (z_6 - z_4)^2)^{3/2},$$

$$s_{23} = ((z_3 - z_2)^2 + (z_6 - z_5)^2)^{3/2}.$$

Выпишем в качестве примера уравнение для компоненты скорости  $v_{1x}$ :

$$\frac{dv_{1x}}{dt} = G \left( \frac{m_2}{s_{12}} (r_{2x} - r_{1x}) + \frac{m_3}{s_{13}} (r_{3x} - r_{1x}) \right),$$

или



$$\frac{dz_7}{dt} = G \left( \frac{m_2}{s_{12}} (z_2 - z_1) + \frac{m_3}{s_{13}} (z_3 - z_1) \right),$$

Аналогичным образом можно получить выражения и для остальных производных. Код функции, вычисляющей правые части уравнений движения приведен во фрагменте 44. Обратите внимание только, что в нем использована принятая в Питон нумерация индексов массивов с нуля, поэтому вектору  $\vec{r}_2 - \vec{r}_1$  соответствует массив `r_01`, а величине  $s_{23}$  – переменная `s12` и т. д.

Фрагмент кода 44. Функция вычисления правых частей уравнений движения

```

1  def eq(t, z):
2      r_01 = np.array([z[1]-z[0], z[4]-z[3]])
3      r_02 = np.array([z[2]-z[0], z[5]-z[3]])
4      r_12 = np.array([z[2]-z[1], z[5]-z[4]])
5      s01 = np.linalg.norm(r_01)**3
6      s02 = np.linalg.norm(r_02)**3
7      s12 = np.linalg.norm(r_12)**3
8      f0 = m1/s01*r_01 + m2/s02*r_02
9      f1 = -m0/s01*r_01 + m2/s12*r_12
10     f2 = -m0/s02*r_02 - m1/s12*r_12
11     return np.concatenate((z[6:],
12                             [f0[0], f1[0], f2[0]],
13                             [f0[1], f1[1], f2[1]]))

```

Описание целого класса для создания и показа небесного тела в данном случае не является обязательным – «отобразить тело» означает всего лишь поставить точку на графике. Однако для некоторого единообразия используем класс `Astro_Body` и в этом примере (фрагмент кода 45). По умолчанию тело считается имеющим единичную массу и нулевую скорость; оно расположено в начале координат и отображается черным цветом. За отображение траектории тела отвечает объект `tr` (класса `gcurve`): метод `show` выводит в область графика точку, имеющую координаты тела (строка 15). Учитывая значение параметра `dot` (строка 13) объекта `tr`, эта последняя выведенная точка будет иметь форму закрашенного круга радиуса в 4 пиксела.

Фрагмент кода 45. Класс для 2D-отображения небесного тела

```

1 class Astro_Body:
2     def __init__(self, name, mass=1, radius=1,
3                 r = vector(0,0,0),
4                 v = vector(0,0,0),
5                 color = color.black):
6         self.name = name
7         self.mass = mass
8         self.radius = radius
9         self.r = r
10        self.v = v
11        self.a = vector(0,0,0)
12        self.tr = gcurve(width=3, color=color,
13                        dot=True, dot_radius=4)
14    def show(self):
15        self.tr.plot(self.r.x, self.r.y)

```

В разделе инициализации (фрагмент кода 46) зададим значения гравитационной постоянной, масс тел, цветов, используемых для их отображения, а также создадим список векторов начальных положений объектов на плоскости. Параметры ширины и высоты области отображения создаваемого двумерного графика ( $x_{\min}$ ,  $x_{\max}$ ,  $y_{\min}$ ,  $y_{\max}$ ) не будем задавать заранее – тогда область автоматически подстроится под эти максимальные и минимальные значения.

Фрагмент кода 46. Инициализация графики

```

1 G = 1
2 MASSES = [1.0, 1.0, 1.0]
3 COLORS = [color.black, color.red, color.green]
4 posA = vec(-0.5, 0, 0)
5 posB = vec(0.5, 0, 0)
6 posC = vec(0.0207067154, 0.3133550361, 0)
7 pos = [posA, posB, posC]
8
9 gd = graph(width=550, height=600,
10          xtitle='<i>x</i>', ytitle='<i>y</i>')
11
12 sun = []
13 for i in range(3):
14     sun.append(Astro_Body(f'S{i}', MASSES[i],
15                          r=pos[i], color=COLORS[i]))
16     sun[i].show()

```

После создания списка «солнц» – объектов типа `Astro_Body` – на экране отобразятся три точки, соответствующие их начальным положениям. Остается запустить анимацию движения.

Осуществим ее в два этапа (см. фрагмент кода 47). Сначала численно решим задачу Коши на отрезке  $[0, T]$ , где  $T$  – значение периода, взятое из таблицы 9. В строках 1–4 задаются параметры, требуемые для этого решения, в частности – начальное значение массива  $z$ , содержащего координаты и скорости трех тел. Вызов функции `solve_ivp` завершает этот этап. Его результатом, с учетом значения `dense_output=True`, является объект `solution.sol`, позволяющий вычислить все значения массива  $z$  в интересующий нас момент времени  $t$ . На втором этапе остается в цикле с некоторым произвольным шагом по времени (этот шаг не связан с точностью решения, а определяет лишь гладкость построения траекторий) вычислять значения массива  $z$  и использовать первые шесть из них (координаты тел) для построения траектории.

Фрагмент кода 47. Численное решение и его визуализация

```

1 | m0, m1, m2 = MASSES
2 | t0, T = 0, 2.1740969264
3 | z0 = np.array([-0.5, 0.5, posC.x,
4 |               0, 0, posC.y,
5 |               0, 0, 0, 0, 0, 0])
6 | solution = solve_ivp(eq, [t0, T], z0,
7 |                    method='DOP853',
8 |                    rtol=1.0e-9, atol=1.0e-12,
9 |                    dense_output=True)
10 | dt = 0.0005
11 | t = 0
12 | while t<=T:
13 |     [x0, x1, x2, y0, y1, y2] = solution.sol(t)[:6]
14 |     sun[0].r = vec(x0,y0,0)
15 |     sun[1].r = vec(x1,y1,0)
16 |     sun[2].r = vec(x2,y2,0)
17 |     for i in range(3):
18 |         sun[i].show()
19 |     t = t + dt
20 |     sleep(dt/100)

```

Результаты работы программы – траектории движения и положение тел в конце периода – приведены на рис. 56. Интересно отметить, что в

отличие от движения по восьмерке, построенные траектории не являются замкнутыми орбитами: каждое тело перемещалось по сложной криволинейной траектории из начальной точки в конечную, а потом по ней же возвратилось назад.

Особого внимания заслуживает строчка 8 во фрагменте кода 47, где задаются параметры численного метода, связанного с погрешностью вычислений. По умолчанию они имеют другие значения:  $rtol=1e-3$ ,  $atol=1e-6$ . Как отмечалось выше, задача трех тел является очень чувствительной и к начальным данным, и к погрешностям метода. Использование значений, принятых по умолчанию, т. е. вычисления с меньшей точностью, приводят к совершенно другим результатам: система разлетается (см. рис 57).



Рис. 56. Траектории движения гравитирующих тел

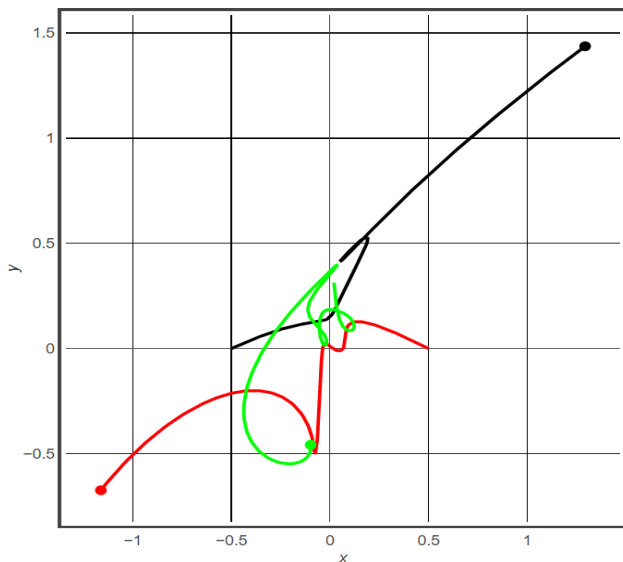


Рис. 57. Расчет траекторий с недостаточной точностью

### Трехмерная визуализация периодических траекторий.

Параметры системы, приведенные во второй строке таблицы 9, будут использованы для построения трехмерной модели системы. Для численного решения попробуем использовать метод Эйлера-Кромера и оценим, насколько он применим в данной задаче. Библиотеки `numpy` и `scipy` подключаться не будут, правые части уравнений движения, т. е. ускорения тел, будем вычислять исключительно средствами пакета `python`.

Во фрагменте кода 48 приведен текст функции, вычисляющей вектор ускорения  $i$ -го тела, возникающего вследствие действия гравитационных сил притяжения к двум другим телам. Условный оператор в строках 3–11 использован как самое наглядное (хотя, скорее всего, и не самое *pythonic*) средство определения, какие тела действуют на заданное. В строках с 12 по 15 вычисляются векторы, соединяющие заданное тело с остальными, и квадраты их длин. Функция `norm`, использованная для вычисления возвращаемого значения, вычисляет единичный вектор, сонаправленный с заданным.

```

1  def g_acc(i):
2      body = sun[i]
3      if i==0:
4          body1 = sun[1]
5          body2 = sun[2]
6      elif i==1:
7          body1 = sun[0]
8          body2 = sun[2]
9      else:
10         body1 = sun[0]
11         body2 = sun[1]
12     r_1=body1.r-body.r
13     r_2=body2.r-body.r
14     r12=mag2(r_1)
15     r22=mag2(r_2)
16     return G*(body1.mass/r12*norm(r_1)+
17             body2.mass/r22*norm(r_2))

```

Реализация класса `Astro_Body` (фрагмент кода 49) в этом случае будет во многом аналогична приведенной во фрагменте кода 45.

Фрагмент кода 49. Класс для 3D-отображения небесного тела

```

1  class Astro_Body:
2      def __init__(self, name, mass=1, radius=1,
3                  r=vector(0,0,0),
4                  v=vector(0,0,0),
5                  color=color.white):
6          self.name = name
7          self.mass = mass
8          self.radius = radius
9          self.r = r
10         self.v = v
11         self.a = vector(0,0,0)
12         self.ball=sphere(pos=self.r,
13                          radius = self.radius,
14                          color=color,
15                          make_trail=True,
16                          interval=100)
17     def show(self):
18         self.ball.pos = self.r

```

По умолчанию тело также считается имеющим единичную массу и нулевую скорость и расположенным в начале координат. Важным отличием данного случая является тот факт, что параметры масса, радиус-вектор и скорость существенно используются не только для визуализации, но и для проведения расчетов. За отображение тела в 3D теперь отвечает объект `sphere`, ее радиус по умолчанию равен единице. В разделе инициализации (фрагмент 50) при создании объектов он выбран равным 0.02 (константа  $R$ ). Метод `show` просто меняет координаты центра сферы, что автоматически приводит к ее отрисовке. Значение параметра `make_trail=True` приводит к тому, что траектория движения сферы остается видимой на экране все время движения.

Фрагмент кода 50. Инициализация трехмерной визуализации

```

1 | G = 1
2 | COLORS = [color.red, color.blue, color.orange]
3 | MASSES = [1, 0.8, 0.8]
4 | R = 0.02
5 | scene = canvas(width=600, height=600,
6 |               background = color.cyan,
7 |               range=0.6, autoscale=False)
8 | scene.forward = vector(0,0,-1)
9 |
10 | posA = vec(-0.5, 0, 0)
11 | posB = vec(0.5, 0, 0)
12 | posC = vec(0.0009114239, 0.3019805958, 0)
13 | pos = [posA, posB, posC]
14 |
15 | sun = []
16 | for i in range(3):
17 |     sun.append(Astro_Body(f'S{i}', MASSES[i],
18 |                          radius=R, r = pos[i],
19 |                          color=COLORS[i]))

```

Численная схема расчета и визуализации траекторий движения приведена во фрагменте кода 51. В качестве метода интегрирования используется метода Эйлера-Кромера: в цикле (строки 4–6) вычисляются ускорения всех точек системы в момент времени  $t$  и на их основе определяются скорости в момент времени  $t+dt$ . Следующим циклом (строки 7–9) определяются и визуализируются положения точек тела в момент времени  $t+dt$ . Цикл `while` (строка 3) обеспечивает выполнение

расчетов на интервале времени, соответствующем одному периоду орбитального движения, так что, если параметр (шаг) интегрирования выбран достаточно малым, в результате выполнения программы тела вернутся в исходное положение.

Фрагмент кода 51. Расчеты траекторий методом Эйлера-Кромера

```
1 dt = 0.000025
2 t, T = 0, 1.8286248401
3 while t < T:
4     for i in range(3):
5         sun[i].a = g_acc(i)
6         sun[i].v += sun[i].a*dt
7     for i in range(3):
8         sun[i].r += sun[i].v*dt
9         sun[i].show()
10    t = t + dt
```

Рис. 58 демонстрирует, что при заданном шаге интегрирования методом Эйлера-Кромера тела действительно вернулись в исходное положение. Однако эксперименты показали, что если продолжать вычисления с этим же шагом, то повторить траектории не получается: ошибка, накопившаяся за первый период вычислений, приводит к малому отклонению результата от начального состояния. Но даже это малое отклонение в силу высокой чувствительности задачи к начальным данным полностью разрушит картину траекторий уже к середине второго периода (см. рис. 59).

Ситуация с методом восьмого порядка точности DOP853, использованным в предыдущем разделе во многом аналогична: при использовании приведенных во фрагменте кода 47 высоких параметров точности численное решение полностью теряет периодический характер при  $t = 3T$ .



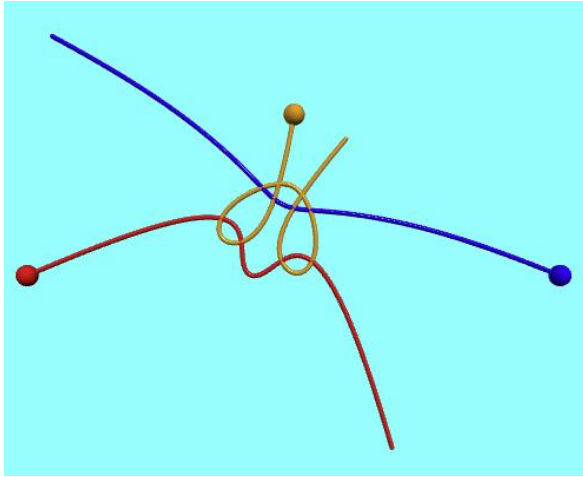


Рис. 58. Трехмерная визуализация периодического движения;  $t = T$

Разработанная программа практически без изменений может быть использована для отображения упомянутой хореографии «Восьмерки» (см. рис. 51). Требуется только откорректировать начальные положения в соответствии с Таблицей 8 и при инициализации тел добавить параметр  $v=\text{vector}()$ , координаты которого заполнить данными из этой же таблицы. В качестве времени окончания расчетов выберем значение  $\bar{T} = 6.32591398$  из работы [5.12]. В результате получаем траектории, изображенные на рис. 60. Для большей наглядности параметр  $R$  (радиус сфер) в программе увеличен в пять раз, а камера немного отодвинута ( $\text{range}=1.5$ ).

Интересно отметить, что данный набор начальных условий гораздо менее чувствителен к погрешностям метода: визуально тела продолжают двигаться по той же самой восьмерке в течение трех периодов даже при шаге интегрирования  $dt = 0.001$ . Если же говорить не о визуальных, а о числовых характеристиках, то значение заданной соотношением (42) целевой функции  $\Psi$ , вычисленной при  $t = k\bar{T}$  и интегрировании методом Эйлера-Кромера с шагом 0.0001 не превышает 0.01 для значений  $k$  в диапазоне  $1 \div 20$ .

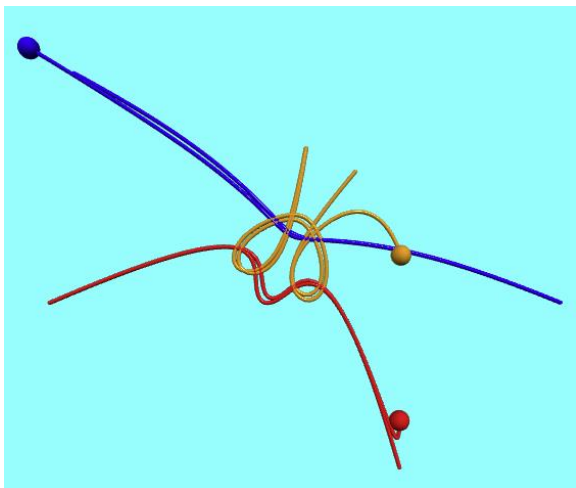


Рис. 59. Нарушение периодичности движения вследствие накопившейся вычислительной ошибки;  $t = 1.5 T$

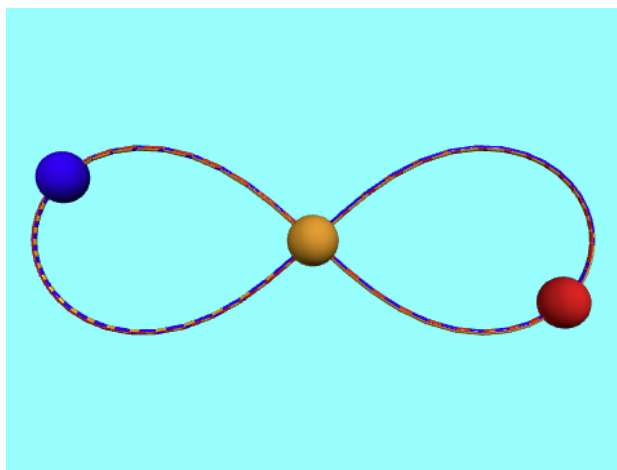


Рис. 60. 3D-визуализация движения по восьмерке.  
Метод Эйлера-Кромера,  $\Delta t = 0.001$

В заключение отметим, что переборная схема поиска периодических движений системы трех тел была использована ее авторами и для решения задач, близких по постановке к сформулированным в проектных заданиях.

В новом варианте [5.13] три тела в начальном состоянии располагались на одной прямой, расстояние между двумя принималось за единицу; требовалось найти координату третьего тела и начальные скорости тел, перпендикулярные соединяющей их прямой, так, чтобы движение системы оказалось периодическим. В результате продолжительных вычислений с использованием мощностей национального суперкомпьютерного центра КНР авторам удалось найти семейство из 13315 устойчивых периодических орбит системы из трех тел. Для расчетов ими был использован находящийся в настоящий момент на четвертом месте в мире по производительности суперкомпьютер Тяньхэ-2 (16 тысяч узлов, каждый из которых включает два 12-ядерных процессора Intel Xeon E5-2692).

Практически все наблюдаемые периодические тройные системы являются иерархическими (подобными системе Солнце–Земля–Луна). Традиционно считалось, что неиерархические тройные системы являются неустойчивыми и поэтому должны распадаться на стабильную двойную систему и одну звезду. Поэтому обнаруженное авторами работы [5.13] семейство из тысяч устойчивых периодических орбит неиерархических тройных систем с неравными массами может существенно поменять традиционные взгляды, в том числе и на область поиска новых астрономических систем.

## ЛИТЕРАТУРА К ГЛАВЕ 5

- 5.1. Авдюшев В. А. Численное моделирование орбит небесных тел. – Томск: Издательский Дом Томского государственного университета, 2015. – 336 с.
- 5.2. Белецкий В. В. Очерки о движении космических тел. – М.: Либроком, 2017. – 432 с.
- 5.3. Бордовицына Т. В. Современные численные методы в задачах небесной механики. – М.: Наука. Главная редакция физико-математической литературы, 1984. – 136 с.
- 5.4. Бутиков Е. И. Компьютерное моделирование движений космических тел [Электронный ресурс]. – СПб, 2016. – 303 с. – URL: <http://butikov.faculty.ifmo.ru/MotionsRus.pdf> (дата обращения: 01.02.2021)
- 5.5. Емельянов Н. В. Практическая небесная механика [Электронный ресурс]. – М.: Физический факультет МГУ, 2018. – 270 с. – URL:

<http://Infm1.sai.msu.su/neb/kaf/pcm/PCMcorr2020SE.pdf> (дата обращения: 01.02.2021)

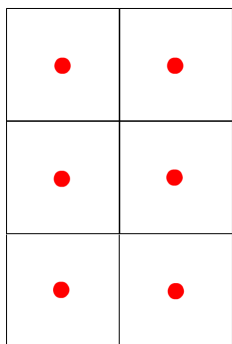
- 5.6. Заусаев А. Ф., Романюк М. А. Численные методы в задачах математического моделирования движения небесных тел в Солнечной системе. – Самара: Самар. гос. техн. ун-т, 2017. – 265 с.
- 5.7. Карауш А. А. Выбор численного метода интегрирования дифференциальных уравнений для задач спутниковых навигационных технологий // Доклады Томского государственного университета систем управления и радиоэлектроники. – 2014. № 31. – С. 174–178.
- 5.8. Лю Цысинь. Задача трех тел. – М.: Fanzon, 2017. – 464 с.
- 5.9. Маркеев А. П. Задача трёх тел и её точные решения // Соросовский образовательный журнал. – 1999. № 9. – С. 112–117.
- 5.10. Мартынова А. И., Орлов В. В., Рубинов А. В., Соколов Л. Л., Никифоров И. И. Динамика тройных систем. – СПб.: Изд-во С.-Петерб. ун-та, 2010. – 216 с.
- 5.11. Чеботарев Г. А. Аналитические и численные методы небесной механики. – М., Л.: Наука, 1965. – 368 с.
- 5.12. Chenciner A., Montgomery R. A remarkable periodic solution of the three-body problem in the case of equal masses // *Annals of Mathematics – Second Series*. – 2000. – Vol. 152, № 3. – Pp. 881–901. doi: 10.2307/2661357
- 5.13. Li X., Li X., Liao S. One family of 13315 stable periodic orbits of non-hierarchical unequal-mass triple systems // *Sci. China Phys. Mech. Astron.* – 2021. – Vol. 64. – P. 219511. doi: 10.1007/s11433-020-1624-7
- 5.14. Li X., Liao S. Clean numerical simulation: a new strategy to obtain reliable solutions of chaotic dynamic systems // *Applied Mathematics and Mechanics (English Edition)*. – 2018. – Vol. 39, № 11. – Pp. 1529–1546. doi: 10.1007/s10483-018-2383-6
- 5.15. Li X., Liao S. Collisionless periodic orbits in the free-fall three-body problem // *New Astronomy*. – 2019. – Vol. 70. – Pp. 22–26. doi: 10.1016/j.newast. 2019.01.003
- 5.16. Musielak Z. E., Quarles B. The three-body problem // *Reports on Progress in Physics*. – 2014. – Vol. 77, № 6. – P. 065901. doi: 10.1088/0034-4885/77/6/065901
- 5.17. Montgomery R. *N*-body choreographies // *Scholarpedia*. – 2010. – Vol. 5, № 11. – P. 10666. doi: 10.4249/scholarpedia.10666
- 5.18. Mocz P. Create Your Own N-body Simulation (With Python) [Электронный ресурс]. – URL: <https://medium.com/swlh/create-your-own->

- n-body-simulation-with-python-f417234885e9 (дата обращения: 01.02.2021)
- 5.19. Price-Whelan A.M. Gala: A Python package for galactic dynamics // The Journal of Open Source Software. – 2017. – Vol.2, № 18. doi: 10.21105/joss.00388
- 5.20. The Astropy Project [Электронный ресурс]. – URL: <http://www.astropy.org> (дата обращения: 01.02.2021)
- 5.21. Wang J. Computational modeling and visualization of physical systems with Python. – Hoboken, NJ: John Wiley & Sons, 2015. – 475 p.
- 5.22. Wild W.J. Euler's three body problem // Am. J. Phys. – 1980. – Vol. 48, № 4. – Pp. 297301. doi: 10.1119/1.12144

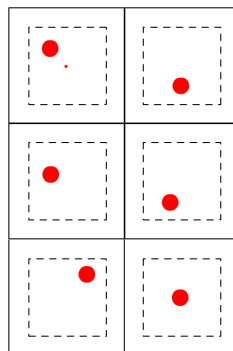
## ГЛАВА 6. ПРОЕКТНОЕ ЗАДАНИЕ «ЗНАКОМСТВО С МОЛЕКУЛЯРНОЙ ДИНАМИКОЙ»

### ФОРМУЛИРОВКА ЗАДАНИЯ

Для заданного варианта расположения атомов (А или Б, см. рис. 61) в кубической ячейке  $10 \times 10 \times 10$  разработать программу, моделирующую движение атомов заданного вещества. Величину временного интервала моделирования выбрать исходя из вычислительной мощности имеющегося компьютера, обеспечив шаг интегрирования по времени не более, чем  $0.002$  от характерной единицы времени<sup>28</sup>.



Вариант А. Расстояние между центрами атомов  $0.95\sigma$



Вариант Б. Расстояние между центрами квадратов –  $1.2\sigma$ , длина стороны зоны случайного размещения атома  $0.8\sigma$

Рис. 61. Схемы расположения атомов<sup>29</sup>:

Вычисления провести для двух типов граничных условий:

- Периодические граничные условия (бесконечная среда)
- На верхней и нижней гранях ячейки задана температура  $T_1$ ; комментарий по выбору ее значения см. ниже.

<sup>28</sup> Желательно достижение временного интервала в 3–5 пикосекунд.

<sup>29</sup> В процессе моделирования рекомендуется дополнительно оценить, произойдет ли заметное изменение результатов, если расстояние между атомами уменьшить или увеличить в полтора-два раза.

В зависимости от номера варианта и соответствующего ему вещества в таблице 10 в качестве потенциала межатомного взаимодействия использовать тот, данные о параметрах которого приведены в таблицах 11 или 12.

Таблица 10. Варианты проектных заданий

Номер варианта	Атом	Схема расположения атомов
1	He	А
2	Ni	Б
3	Ne	А
4	Kr	Б
5	Pb	А
6	Ag	Б
7	Xe	А
8	Mo	Б
9	F	А
10	Cl	Б
11	K	А
12	Na	Б
13	H <sub>2</sub>	А
14	Br	Б
15	Cr	А
16	Fe	Б
17	Ba	А
18	O <sub>2</sub>	Б
19	Ag	А
20	N <sub>2</sub>	Б

Таблица 11. Параметры потенциала Ленарда-Джонса<sup>30</sup>

Атом	$\varepsilon / k_B, \text{K}$	$\sigma, \text{нм}$
Ar	119.8	0.341
H <sub>2</sub>	37	0.293
He	10.2	0.228
N <sub>2</sub>	37.3	0.331
O <sub>2</sub>	118	0.358
Ne	47.0	0.272
Kr	164.0	0.383
Xe	218.2	0.406
CO	100	0.376
CO <sub>2</sub>	189	0.449
CH <sub>4</sub>	149	0.378
C <sub>2</sub> H <sub>4</sub>	199	0.452
CF <sub>4</sub>	152	0.47
C <sub>2</sub> H <sub>6</sub>	243	0.395
C <sub>3</sub> H <sub>8</sub>	242	0.564
F	52.8	0.283
Cl	173.5	0.335
Br	257.5	0.354

Таблица 12. Параметры потенциала Морза<sup>31</sup>

Металл	$\alpha, \text{нм}^{-1}$	$\sigma, \text{нм}$	$\varepsilon, \text{эВ}$	$\varepsilon / k_B, \text{K}$
Pb	11.836	0.3733	0.2348	2725
Ag	13.690	0.3115	0.3323	3856
Ni	14.199	0.2780	0.4205	4880
Cu	13.588	0.2866	0.3429	3979
Al	11.646	0.3.253	0.2703	3137

<sup>30</sup> Данные из работ [6.6] и [6.11].<sup>31</sup> Данные из работы [6.9]



Ca	8.0535	0.4569	0.1623	1883
Sr	7.3776	0.4988	0.1513	1756
Mo	15.079	0.2976	0.8032	9321
W	14.116	0.3032	0.9906	11495
Cr	15.721	0.2754	0.4414	5122
Fe	13.885	0.2845	0.4174	4844
Ba	6.5698	0.5373	0.1416	1643
K	4.9767	0.6369	0.05424	629
Na	5.8993	0.5336	0.06334	735
Cs	4.1569	0.7557	0.04485	520
Rb	4.2981	0.7207	0.04644	539

В обеих таблицах  $k_B$  – постоянная Больцмана,  $1.380649 \cdot 10^{-23}$  Дж/К.

Начальную температуру в ячейке  $T_0$  и температуру на гранях  $T_1$  выбирать следующим образом<sup>32</sup>:

- при использовании потенциала Леннарда-Джонса осуществить моделирование перехода жидкости в газообразное состояние: начальную температуру выбрать немного ниже температуры кипения заданного вещества, значение температуры  $T_1$  должно быть в полтора-два раза выше температуры кипения;

- при использовании потенциала Морза осуществить моделирование перехода из твердого состояния в жидкое: начальную температуру выбрать немного ниже температуры плавления заданного вещества, значение температуры  $T_1$  должно быть в полтора-два раза выше температуры плавления.

Для интегрирования уравнений можно воспользоваться любым вариантом метода Верле или перешагивания.

В качестве результата представить:

- 1) График зависимости энергии системы частиц в ячейке от времени
- 2) График зависимости температуры в ячейке от времени
- 3) Визуализацию системы средствами VPython

---

<sup>32</sup> В комментариях в тексте программы привести ссылку на найденные в сети данные о значениях температуры плавления или температуры кипения.

**Молекулярная динамика.** Следуя [6.12], в настоящем пособии методом молекулярной динамики (ММД) будем называть такой способ компьютерного моделирования, при котором эволюция набора взаимодействующих между собой частиц (атомов, молекул, кластеров и т. д.) определяется путем численного интегрирования уравнений их движения. Таким образом, в молекулярной динамике используются классические законы механики; в частности, каждый  $i$ -й атом системы из  $N$  атомов движется в соответствии с законом Ньютона

$$m_i \vec{a}_i = \vec{F}_i, \quad (43)$$

где  $m_i$  – масса этого атома,  $\vec{a}_i = d^2\vec{r}_i/dt^2$  – его ускорение, а действующая на атом сила  $\vec{F}_i$  обусловлена влиянием остальных атомов системы.

Таким образом, схема ММД имеет детерминированный характер: на основе информации о начальных положениях и скоростях всех частиц системы последующая эволюция этой системы во времени *в принципе*<sup>33</sup> может быть полностью определена. Если говорить чуть менее строго, то частицы в компьютерной модели будут двигаться, наталкиваясь друг на друга, обтекая соседей (если система жидкая), участвуя в колебательном движении вместе со своим окружением, возможно, испаряясь из системы, если есть свободная поверхность и т. п., то есть, в итоге, ведя себя практически так же, как атомы и молекулы в реальном веществе.

Результатом расчетов служит траектория системы частиц в  $6N$ -мерном фазовом пространстве ( $3N$  координат и  $3N$  компонент скоростей или, что более приятно в физике, импульсов). Однако сама по себе эта траектория обычно не очень существенна. ММД в известной степени это метод статистической механики. Физические величины, в этом смысле, представляют собой некоторые усредненные значения по той или иной конфигурации. Траектория, рассчитанная методом молекулярной динамики, как раз и предоставляет такой набор конфигураций. Осреднение при этом может проводиться как по ансамблю частиц, так и по времени

---

<sup>33</sup> Как было показано в главе 3, погрешность используемых численных методов и ошибки округления могут приводить к существенным отличиям расчетных траекторий от реальных.

вычислительного эксперимента. В последнем случае измерение той или иной физической величины на основе компьютерного моделирования сводится к обыкновенному арифметическому осреднению многочисленных мгновенных значений этой величины, найденных в ходе работы программы.

С развитием средств вычислительной техники ММД приобретает все большую актуальность, поскольку за счет количественного усложнения компьютерной модели он может позволить получать качественно новые результаты. Как принципиально дискретный метод, он не имеет недостатков континуальных моделей, проявляющихся при нарушении сплошности вещества или в результате дискретности его внутренней структуры. ММД по сравнению с методами, основанными на концепции сплошной среды, требует значительно меньше априорных предположений о свойствах материала. Использование даже простейшего потенциала взаимодействия (например, типа Леннарда-Джонса) позволяет моделировать такие сложнейшие эффекты, как пластичность, образование трещин, разрушение, температурное изменение свойств материала, фазовые переходы. Для описания каждого из этих эффектов в рамках сплошной среды требуется отдельная теория, в то время как при использовании ММД эти эффекты получаются автоматически, в результате интегрирования уравнений движения. [6.4]

Современные сферы применения ММД очень широки и разнообразны. Следуя работам [6.4, 6.5, 6.7, 6.12, 6.16] приведем некоторые из таких областей.

- ММД возник как средство анализа задач, связанных с жидкостями. И до настоящего времени исследование структуры различных жидкостей и происходящих в них процессов является важным приложением молекулярной динамики. Современные модели межатомных взаимодействий позволяют изучать новые, в том числе, многокомпонентные системы, а технологии анализа неравновесных процессов дают возможность исследовать явления переноса, например такие как вязкость и теплопроводность.
- Другое традиционное приложение моделирования на основе молекулярной динамики – изучение влияния дефектов кристаллической структуры на механические свойства твердых тел – по-прежнему представляет большой практический интерес и

является актуальной темой физики твёрдого тела. Улучшенные потенциалы позволяют существенно приблизить расчеты в этой сфере к реальным процессам.

- Моделирование средствами молекулярной динамики позволяет понять микроскопические механизмы процесса разрушения твёрдых тел, сформулировать новые технологические принципы работы с конструкционными материалами.
- В 80-х годах прошлого века начался бум в физике поверхностей, связанный с появлением новых экспериментальных инструментов с микроскопическим разрешением. Компьютерное моделирование с использованием ММД позволяет лучше понять такие явления как адгезия и трение, поверхностная диффузия и залечивание поверхностей, а также усовершенствовать технологии огранки, придания шероховатости и др.
- ММД широко применяется для изучения структуры ударных волн в конденсированных средах. Пространственный и временной масштаб физико-химических процессов, происходящих в ударном скачке, идеально подходит для изучения этих процессов методами молекулярной динамики.
- Базовым объектом, или частицей, в ММД могут выступать не только атомы и молекулы, но и целые их объединения – кластеры. Интересной особенностью таких агломераций во многих случаях является существенное отличие их характеристик от макроскопических свойств твердого вещества, что прежде всего связано с влиянием поверхности и анизотропией свойств. Моделирование металлических кластеров представляет интерес, например, для химической промышленности, где такие кластеры выступают в качестве катализаторов важных химических реакций.
- ММД позволяет изучать динамику крупных макромолекул, включая биологические системы, такие как белки, нуклеиновые кислоты (ДНК, РНК), биомембраны. При этом динамика может играть ключевую роль в процессах, которые влияют на функциональные свойства биомолекулы, а, следовательно, ее учет очень важен в задачах оптимального управления такими процессами, например, при производстве лекарств. Влияние моделирования на основе ММД на молекулярную биологию и открытие лекарств резко возросло в

последние годы. Имитационное моделирование позволяет отобразить поведение белков и других биомолекул с полной атомарной детализацией и с очень точным временным разрешением. Компьютерная проверка физико-химических свойств молекулы в фармацевтической промышленности обходится существенно дешевле ее синтеза.

- Молекулярная динамика сегодня позволяет моделировать объемы материала размером до кубического микрометра, что соответствует примерно миллиарду частиц (куб 1000 x 1000 x 1000 частиц). Таким образом, практически любые наноструктуры могут быть смоделированы с чрезвычайно высокой степенью точности. Поэтому ММД является важным теоретическим инструментом современной наномеханики.

**Краткая историческая справка.** Полное описание всех ранних разработок ММД, безусловно, выходит за рамки данного пособия. Ниже будут упомянуты только несколько ключевых работ, появившихся в 50-х и 60-х годах, которые можно рассматривать как своеобразные важные вехи развития молекулярной динамики.

- Первой работой, описывающей моделирование на основе молекулярной динамики считается статья Олдера и Уэйнрайта [6.9], опубликованная в 1957 году. В ней была построена и изучена фазовая диаграмма системы твердых сфер, моделирующих, в частности, твердую и жидкую фазы вещества. Частицы при таком описании взаимодействуют посредством мгновенных столкновений и перемещаются между столкновениями как свободные твердые тела. Расчеты выполнены на UNIVAC и IBM 704. Для системы из 32 частиц анализ семи тысяч соударений занимал около часа.
- Работа Дж. Б. Гибсона, А. Н. Голанда, М. Милграма и Г. Х. Виньярда из Брукхейвенской национальной лаборатории, опубликованная в 1960 г. [6.13], вероятно, является первым примером проведения расчетов молекулярной динамики с непрерывным потенциалом на основе конечно-разностного интегрирования по времени. Расчет для системы из 500 атомов был выполнен на IBM 704 и занимал около минуты на каждый временной шаг. Тематика статьи связана с изучением дефектов, вызванных радиационным повреждением, что хорошо соответствует временам холодной войны.

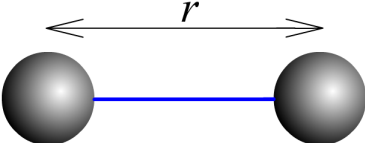
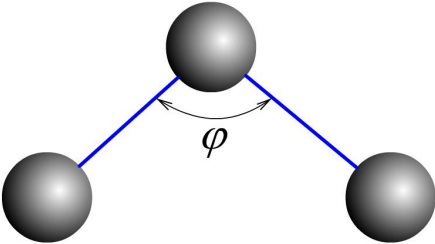
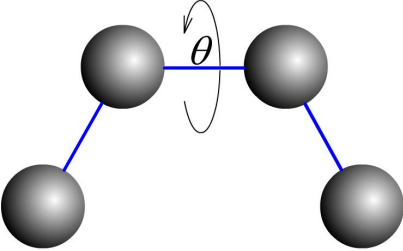
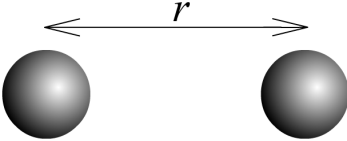
- Среди пионеров молекулярной динамики безусловного упоминания заслуживает имя Анизура Рахмана из Аргоннской национальной лаборатории. В своей знаменитой статье 1964 года [6.18] с помощью компьютера CDC 3600 он исследовал ряд свойств жидкого аргона, используя потенциал Леннарда-Джонса в системе, содержащей 864 атома. Наследие разработанного А. Рахманом компьютерного кода можно проследить во многих действующих по всему миру программах молекулярной динамики.
- В 1967 и 1968 годах Лу Верле опубликовал [6.19, 6.20] две части статьи, посвященной изучению фазовой диаграммы аргона с помощью потенциала Леннарда-Джонса и вычислению корреляционных функций для проверки теории жидкого состояния. В его работах использован алгоритм интегрирования уравнений движения системы частиц по времени, известный в настоящее время как *метод Верле*. Кроме того, Верле предложил еще одну схему ускорения расчетов, известную как *список соседей Верле* (Verlet neighbor list).

**Потенциалы молекулярной динамики.** В предыдущих частях этой главы уже несколько раз упоминались потенциалы, использование которых позволяет решать тот широкий круг задач, которые стоят перед молекулярной динамикой. Речь идет о том, что сила в уравнениях (43) представляет собой производную (точнее говоря, градиент) некоторой потенциальной функции:

$$\vec{F}_i = - \frac{\partial U(\vec{r}_1, \vec{r}_2, \dots, \vec{r}_N)}{\partial \vec{r}_i} \quad (44)$$

Подробное изложение теории межмолекулярного взаимодействия выходит далеко за рамки данного пособия. Интересующимся данным вопросом можно порекомендовать книгу [6.3], которая содержит детальное описание взаимодействий между молекулами на больших, средних и малых расстояниях, а также сравнительный анализ модельных потенциалов, используемых в современных квантово-химических расчетах и при компьютерном моделировании в физике, химии и молекулярной биологии. Ниже в табл. 13, следуя учебному пособию [6.6], схематически изображены основные силы и соответствующие им потенциальные поля, существующие в любой молекуле, а также межмолекулярные (межатомные) силы.

Таблица 13. Основные взаимодействия, учитываемые ММД

Схема химической связи	Возможный потенциал
	$\frac{1}{2}k_b(r - r_0)^2$
	$\frac{1}{2}k_\varphi(\varphi - \varphi_0)^2$
	$\frac{1}{2}k_\theta(1 + \cos(n\theta - \delta))$
	$\frac{A}{r^{12}} - \frac{B}{r^6}$

Как видно из приведенной таблицы, при моделировании средствами молекулярной динамики, как правило, учитываются энергия связей между парами атомов в молекуле, угловых связей в системе трех атомов и торсионных связей в четырехатомных системах. Кроме химических связей важным, а во многих задачах даже определяющим, является участие несвязанных ван-дер-ваальсовых взаимодействий. Если атомы обладают

ещё и зарядом, то в рассмотрение вводят электростатические силы и потенциалы. Результатом учета всех таких полей является универсальная схема, на основе которой ММД позволяет одинаково эффективно моделировать самые разнообразные системы – от простых атомно-молекулярных фрагментов до сложных полимерных систем и белковых макромолекул [6.6].

Для описания межмолекулярного взаимодействия в ММД используют различные потенциалы парного взаимодействия. В качестве примера рассмотрим сначала один из самых первых таких потенциалов, т. н. степенной потенциал  $m - n$ , введенный в рассмотрение в 1903 году Густавом Ми и носящий в настоящее время имя своего автора. Выражение функции энергии или **потенциал Ми** имеет вид

$$U_{mn} = \varepsilon C_{mn} \left[ \left( \frac{\sigma}{r} \right)^n - \left( \frac{\sigma}{r} \right)^m \right], n > m \quad (45)$$

где  $r$  – расстояние между центрами частиц, а  $\sigma$  – «эффективный» или «характерный» диаметр атома, трактуемый как расстояние, на котором уравниваются силы отталкивания и притяжения между атомами [6.6]. Точнее говоря, параметр  $\sigma$  показывает, насколько близко могут подойти две несвязанные частицы, и поэтому называется радиусом Ван-дер-Ваальса. Он равен половине межъядерного расстояния между этими частицами.

Коэффициент  $C_{mn}$  задается выражением

$$C_{mn} = \left( \frac{n}{m} \right)^{\frac{m}{n-m}} \left( 1 - \frac{m}{n} \right)^{-1}.$$

В этом случае в точке

$$r_{min} = \sigma \left( \frac{n}{m} \right)^{\frac{1}{n-m}}.$$

потенциальная энергия принимает минимальное значение, равное  $-\varepsilon$ . Таким образом, параметр  $\varepsilon$  имеет смысл глубины потенциальной ямы в равновесном состоянии. Зависимость потенциала (45) от расстояния схематически изображена на рис. 62.



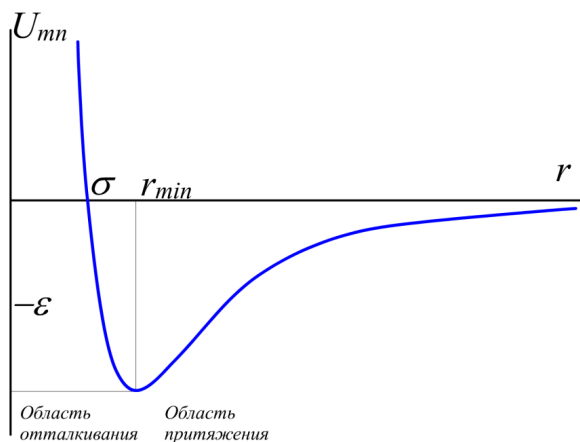


Рис. 62. Характерный вид потенциала  $m - n$

Наиболее известным и распространенным при решении задач методами молекулярной динамики является частный случай потенциала Ми, потенциал 6–12, известный как **потенциал Леннарда-Джонса**. Он назван по имени английского физика, сэра Эдварда Леннарда-Джонса, чьи работы 1924 года по использованию данного потенциала для исследования термодинамических свойств инертных газов послужили основой для дальнейшего его широкого применения к исследованию различных систем. Подставляя значения  $m = 6$  и  $n = 12$  в (45), получаем

$$U = 4\varepsilon \left[ \left( \frac{\sigma}{r} \right)^{12} - \left( \frac{\sigma}{r} \right)^6 \right]. \quad (46)$$

Как отмечено в [6.8], данный потенциал хорошо подходит для описания сферически симметричных молекул, каковыми и являются инертные газы. Более удивительным для исследователей стал тот факт, что и для несферических молекул, таких, как  $\text{H}_2$ ,  $\text{N}_2$ ,  $\text{O}_2$ , а также для молекул с такой же удлинённой формой, как у бензола, использование сферически симметричного потенциала вида (46) во многих случаях приводило к корректным результатам. Итогом многочисленных обсуждений и споров стал вывод о том, что степень такой пригодности существенно зависит от того, насколько сильно возбуждаются вращательные степени свободы молекул при обычных температурах [6.8].

Потенциал Леннарда-Джонса является двухпараметрическим, поэтому его возможности для описания макроскопических параметров моделируемого материала весьма ограничены. Как правило, с его помощью можно удовлетворительно вычислить только один такой материальный параметр, например модуль упругости или скорость распространения продольных волн. Обратите внимание, что здесь не случайно упомянуты параметры, относящиеся не к жидкостям и газам, а к твердым веществам. Потенциал Леннарда-Джонса активно используется и для моделирования твердой фазы, см., например, работу [6.15]. К несомненным достоинствам потенциала относится его вычислительная простота, не требующая вычисления иррациональных и трансцендентных функций. Он широко применяется как классический модельный потенциал, особенно в работах, в которых основной задачей является не получение точных количественных результатов, а исследование общих физических закономерностей [6.1].

Еще один общеупотребительный потенциал был предложен американским физиком Филипом Морзом<sup>34</sup> (Philip Morse) в 1929 году.

**Потенциал Морза** имеет вид

$$U = \varepsilon(e^{-2\alpha(r-\sigma)} - 2e^{-\alpha(r-\sigma)}) \quad (47)$$

Выражение (47) содержит три положительных параметра:  $\alpha$ ,  $\varepsilon$ , и  $\sigma$ . В своей работе Ф. Морз подобрал их значения для целого ряда атомов, используя спектроскопические данные. В 60-х и 70-х годах прошлого века появился целый ряд теоретических и экспериментальных работ (например, [6.14, 6.17]), позволивших скорректировать или уточнить эти значения. Работы в этой области науки продолжаются и до сих пор [6.10].

Характерный вид кривой (47) аналогичен приведенному на рис. 62. Она также содержит участок убывания, один минимум и, оставаясь отрицательной, возрастает, стремясь к нулевому значению.

В отличие от потенциала Леннарда-Джонса, потенциал Морза предназначался, в первую очередь, для моделирования связей в двухатомной молекуле, поэтому параметр  $\sigma$  имеет смысл равновесного

---

<sup>34</sup> В русскоязычной литературе (см. например, Википедию) более употребительным является другой вариант написания – не Морз, а *Морзе*, видимо по аналогии с написанием фамилии автора знаменитой морзянки, который тоже был американцем и тоже «пострадал» от неудачного транскрибирования.

расстояния, т. е. расстояния, на котором в ноль обращается не энергия, а сила – производная энергии по расстоянию. Сам же потенциал обращается в ноль на расстоянии, задаваемом соотношением  $r = \sigma - \ln(2)/\alpha$ .

Как и для потенциала Леннарда-Джонса, минимальное значение потенциал (47), достигаемое в точке в точке  $r = \sigma$ , равно  $-\varepsilon$ , таким образом, и в этом случае параметр  $\varepsilon$  имеет смысл глубины потенциальной ямы.

Дальнодействующая часть потенциала Морза хуже согласуется с опытными данными, так как в этой области расстояний обратная степенная зависимость работает лучше экспоненциальной. При  $r = 0$  потенциал Морза имеет конечное значение, равное  $\varepsilon e^{\alpha\sigma}(e^{\alpha\sigma} - 2)$ . Из таблицы 12 видно, что величина  $\alpha\sigma$  находится в диапазоне от 3 до 4.5, т. е. коэффициент при параметре  $\varepsilon$  достаточно велик. Однако бесконечно большое значение потенциала в нуле гораздо лучше соотносится с основными физическими представлениями, как теоретическими, так и экспериментальными. Тем не менее, потенциал Морза вполне удовлетворительно описывает колебательные энергетические уровни молекул, поскольку для них важна область расстояний в окрестности минимума. Этим определена и область наибольшей применимости потенциала Морза, связанная с исследованиями различных кристаллических свойств твердых тел, поскольку эти свойства наиболее чувствительны к тем диапазонам расстояний, на которых потенциал Морза удовлетворительно описывает реальные межатомные взаимодействия [6.5].

## ПРИМЕР ВЫПОЛНЕНИЯ ЗАДАНИЯ

Очень условно задачи молекулярной динамики можно разделить на три группы: «конструирование» молекул из заданного набора атомов и анализ свойств получившихся веществ, анализ стационарных состояний или динамики релаксации системы к равновесному состоянию и, наконец, изучение существенно неустановившихся процессов и необратимых явлений.

Проектное задание этой главы относится ко второй группе. Схематически анализ таких задач можно представить в виде следующей последовательности действий:

- Предварительный этап (выбор потенциала, задание параметров, обезразмеривание, уточнение уравнений и пр.)
- Задание начальных положений частиц. Обычно в качестве таковых выбирается либо некоторое случайное распределение, соответствующее тем или иным физическим принципам или предположениям, либо расположение частиц в узлах кристаллической решетки заданного типа. При этом расстояние между частицами может выбираться из некоторых априорных предположений (как в случае проектного задания) или вычисляться на основе тех или иных физических данных (например, на основе информации о количестве частиц и плотности, как в приведенном ниже примере).
- Задание начальных скоростей частиц. Как правило, скорости задаются на основе информации о начальной температуре в моделируемом объеме.
- Цикл по времени:
  - $t = t + \Delta t$
  - вычисление положений частиц и скоростей (один шаг численного интегрирования)
  - корректировка положений частиц и скоростей с учетом граничных условий
  - сохранение информации, вычисление осредненных по ансамблю частиц величин и т. п.
- Завершение работы, вычисление средних по интервалу времени характеристик системы.

В качестве основы для примера будет использован приведенный и подробно прокомментированный в [6.6] код моделирования Леннард-Джонсовой жидкости, параметры которой соответствует аргону. Для данного пособия этот код был сначала переведен с языка FORTRAN на Питон, а затем получившаяся программа на Питоне была откорректирована с учетом особенностей этого языка с целью повышения производительности. Тем не менее, основные блоки и логическая структура кода полностью сохранена.

В соответствии с приведенной выше схемой анализа проведем предварительный анализ задачи, в частности, выберем размерные масштабные единицы для физических параметров и запишем уравнения движения в безразмерном виде.

Прежде всего, сформулируем уравнения движения для системы частиц в случае сферически симметричного потенциала, т. е. потенциала, зависящего только от расстояния  $U = U(r)$ . Равенство

$$\vec{F} = -\nabla U$$

означает, что компоненты вектора силы в декартовой системе координат  $x_1, x_2, x_3$  вычисляются по формулам

$$F_i = -\frac{\partial U}{\partial x_i} = -\frac{dU}{dr} \frac{\partial r}{\partial x_i}.$$

Найдя производную

$$\frac{\partial r}{\partial x_i} = \frac{\partial \sqrt{x_1^2 + x_2^2 + x_3^2}}{\partial x_i} = \frac{1}{r} x_i,$$

для силы, соответствующей сферически симметричному потенциалу, получаем

$$\vec{F} = -\frac{1}{r} \frac{dU}{dr} \vec{x}, \quad (48)$$

где вектор  $\vec{x}$  соединяет моделируемую частицу с воздействующей на нее.

Переходим к системе из  $N$  частиц. Сила, действующая на  $i$ -ую частицу системы, запишется в виде

$$\vec{F}_i = \sum_{j=1, j \neq i}^N \vec{F}_{i \leftarrow j},$$

где символом  $\vec{F}_{i \leftarrow j}$  обозначена сила, действующая со стороны  $j$ -й частицы на  $i$ -ю частицу. Ясно, что

$$\vec{F}_{i \leftarrow j} = -\vec{F}_{j \leftarrow i}. \quad (49)$$

Если радиус-векторы  $i$ -ой и  $j$ -ой частиц равны соответственно  $\vec{x}_i$  и  $\vec{x}_j$  (см. рис. 63), то выражение для соединяющего эти частицы вектора  $\vec{x}_{ij}$  принимает вид  $\vec{x}_j - \vec{x}_i$ . Вводя обозначение  $r_{ij} = |\vec{x}_{ij}|$ , с учетом формулы (48) находим

$$\vec{F}_{i \leftarrow j} = -\frac{1}{r_{ij}} \frac{dU}{dr} \Big|_{r=r_{ij}} \vec{x}_{ij}. \quad (50)$$

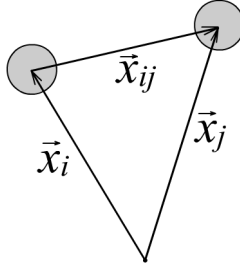


Рис. 63. Радиус-векторы частиц системы

Уточним теперь полученное выражение для силы в случае использования потенциала Леннарда-Джонса. Дифференцируя выражение потенциала (46) и подставляя в (50), получаем

$$\vec{F}_{i \leftarrow j} = \frac{48\varepsilon}{r_{ij}^2} \left[ \left( \frac{\sigma}{r_{ij}} \right)^{12} - \frac{1}{2} \left( \frac{\sigma}{r_{ij}} \right)^6 \right] \vec{x}_{ij}.$$

В результате приходим к следующей системе дифференциальных уравнений движения частиц:

$$m \frac{d^2 \vec{x}_i}{dt^2} = \sum_{j=1, j \neq i}^N \frac{48\varepsilon}{r_{ij}^2} \left[ \left( \frac{\sigma}{r_{ij}} \right)^{12} - \frac{1}{2} \left( \frac{\sigma}{r_{ij}} \right)^6 \right] \vec{x}_{ij}. \quad (51)$$

Обратите внимание, что поскольку в данном примере речь идет об одном веществе – аргоне, то все массы в уравнениях (51) одинаковы.

Приведем уравнения движения к безразмерному виду. В качестве единицы длины выберем величину параметра  $\sigma$ . Безразмерный радиус-вектор и безразмерные расстояния введем с помощью соотношений

$$\vec{\chi}_i = \frac{1}{\sigma} \vec{x}_i, \quad \rho_{ij} = \frac{1}{\sigma} r_{ij}. \quad (52)$$

Что касается единицы времени, то для ее выбора представим время  $t$  в виде

$$t = \tau t_*,$$

где  $\tau$  – безразмерное времени, а  $t_*$  – неизвестный пока параметр, имеющий размерность времени. Подставляя данное выражение для времени вместе с безразмерными представлениями расстояний (52) в уравнения движения (51) и деля обе части последних на  $m$ , приходим к соотношениям

$$\frac{d^2 \vec{\chi}_i}{d\tau^2} = \frac{\varepsilon t_*^2}{m\sigma^2} \sum_{j=1, j \neq i}^N \frac{48}{\rho_{ij}^2} \left[ \left( \frac{1}{\rho_{ij}} \right)^{12} - \frac{1}{2} \left( \frac{1}{\rho_{ij}} \right)^6 \right] \vec{\chi}_{ij}.$$

Естественным значением для параметра  $t_*$  будет такое, которое обратит безразмерный коэффициент  $\varepsilon t_*^2 / (m\sigma^2)$  в единицу, т. е.

$$t_* = \sigma \sqrt{\frac{m}{\varepsilon}}. \quad (53)$$

Данную величину будем называть *характерным временем* системы частиц. Для аргона из таблицы 11 находим

$$\sigma = 0.341 \cdot 10^{-9} \text{ м},$$

$$\varepsilon = 119.8 k_B = 119.8 \cdot 1.380649 \cdot 10^{-23} \text{ Дж} = 1.654 \cdot 10^{-21} \text{ Дж}.$$

Масса атома аргона может быть найдена умножением его относительной атомной массы (39.948) на атомную единицу массы (углеродную единицу), равную  $1.66054 \cdot 10^{-27}$  кг. В результате получаем

$$m_{Ar} = 6.6337 \cdot 10^{-26} \text{ кг}.$$

Подставляя найденные числовые значения в соотношение (53) для величины характерного времени данной задачи получаем

$$t_* = 2.16 \cdot 10^{-12} \text{ с} = 2.16 \text{ пс}.$$

Подводя итоги, получаем, что для решения задачи необходимо численно интегрировать безразмерные уравнения вида

$$\frac{d^2 \vec{\chi}_i}{d\tau^2} = \sum_{j=1, j \neq i}^N \frac{48}{\rho_{ij}^2} \left[ \left( \frac{1}{\rho_{ij}} \right)^{12} - \frac{1}{2} \left( \frac{1}{\rho_{ij}} \right)^6 \right] \vec{\chi}_{ij}, \quad (54)$$

при этом значению временного параметра  $\tau = 1$  будет соответствовать реальное время приблизительно равное двум пикосекундам; таким образом, типичный шаг интегрирования ММД, равный одной фемтосекунде ( $10^{-15}$  с), для нашей задачи будет означать интегрирование по параметру  $\tau$  с шагом 0.0005.

Во фрагменте кода 52 содержится задание основных параметров, соответствующих оригинальной программе. Открываемый на запись файл `result.txt` (строка 4) предназначен для сохранения информации о положениях атомов на очередном шаге интегрирования (точнее говоря, вывод в этот файл совершается с определенной периодичностью). В

отличие от оригинальной программы, в данном случае отсутствуют тонкие настройки этого вывода, соответствующие формату PDB.

За количество атомов системы отвечает параметр `natom` (строка 7). Эта величина не может быть произвольной, поскольку дальше предполагается определенная структура расположения атомов. Как будет показано ниже число `natom/4` должно быть кубом некоторого натурального числа. Таким образом, для анализа влияния количества атомов на время работы программы в качестве значения параметра `natom` можно выбирать, например, числа 500, 2048, 4000 и т. д.

Массивы координат `x_`, скоростей `v_`, ускорений `a_` и сил имеют общую структуру, представляя собой матрицы размера `natom x 3`. В строках 11 и 13 для координат и ускорений резервируется место в памяти. Массив скоростей создается ниже во фрагменте 54; он инициализируется случайными значениями. Сразу уточним, что на самом деле для оптимизации используемой схемы интегрирования в массиве `v_` будут храниться скорости, умноженные на шаг интегрирования по времени  $\Delta t$ , а в массиве `a_` – ускорения, умноженные на  $\Delta t^2/2$ .

Входные значения плотности (строка 18), температуры (строка 19) и шага интегрирования (строка 20) соответствуют оригинальному коду из [6.6]. Параметр плотности будет использован ниже во фрагменте 53 для определения размера куба, занимаемого частицами. Значение `temp=1.5` соответствует температуре  $1.5\varepsilon/k_B$ , т. е. примерно 180К.

Переменные `stepsq` и `stepsqh` введены для большей компактности кода и будут использованы ниже.

Фрагмент кода 52. Задание основных параметров задачи

```
1 import numpy as np
2
3 # Файл хранения информации о каждом шаге моделирования.
4 f = open('result.txt', mode='w')
5
6 # Число атомов в системе
7 natom = 256
8
9 # Выделение памяти под основные массивы данных
10 # массив радиус-векторов частиц
11 x_ = np.empty(natom*3).reshape(natom, 3)
```



```

12 # массив "ускорений"
13 a_ = np.empty(natom*3).reshape(natom,3)
14
15 # входные безразмерные параметры:
16 # плотность, температура и
17 # шаг интегрирования системы по времени
18 densy = 0.9
19 temp = 1.5
20 step = 0.001          # delta t
21 stepsq = step*step    # (delta t)^2
22 stepsqh = 0.5*stepsq # (delta t)^2 / 2

```

Фрагмент кода 53 представляет собой практически необработанный перевод с Фортрана на Питон оригинального текста программы. В строках 1 и 2 вычисляется объем куба, занимаемого частицами, на основе информации о количестве частиц и заданного значения плотности и вычисляется размер стороны этого куба (а также вспомогательная величина, равная половине этой стороны). В строках 5–9 происходит вычисление количества единичных ячеек – гранецентрированных кубиков (рис. 64), расположенных вдоль одного ребра основного куба, занимаемого частицами. В переменной *dist* (строка 10) сохраняется половина длины ребра базовой ячейки.

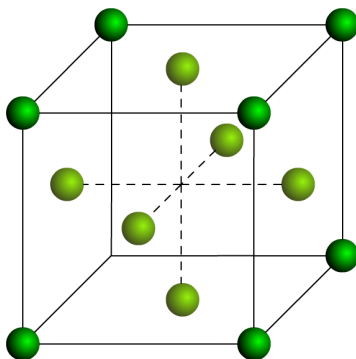


Рис. 64. Базовая ячейка гранецентрированной кубической решетки

В строках 12–15 задаются координаты «стартовой четверки» – одного из углов куба и трех ближних к нему атомов, т. е. атомов, лежащих в центрах граней базовой ячейки, содержащих этот угол. Далее в строках 16–26 положения новых атомов строятся на основе параллельного переноса этих стартовых позиций.

```

1 vol = natom/densty
2 cube = vol**(1/3)
3 cubeh = 0.5*cube
4
5 nunit=(natom/4.)*(1/3)+0.1
6 ncheck=4*(nunit**3)
7 while ncheck<natom:
8     nunit += 1
9     ncheck=4*(nunit**3)
10 dist=0.5*cube/nunit
11
12 x_[0]=np.array([0.0, 0.0, 0.0])
13 x_[1]=np.array([0.0, dist, dist])
14 x_[2]=np.array([dist, 0.0, dist])
15 x_[3]=np.array([dist, dist, 0.0])
16 m=0
17 kct=0
18 for i in range(int(nunit)):
19     for j in range(int(nunit)):
20         for k in range(int(nunit)):
21             for ij in range(4):
22                 if kct<natom:
23                     tmp = np.array([k, j, i], dtype=np.float64)
24                     x_[ij+m]=x_[ij]+2.0*dist*tmp
25                     kct+=1
26                 m+=4

```

В молекулярной динамике основными являются два способа задания начальных скоростей частиц. В первом из них опираются на известные физические теории распределения молекул по скоростям; наиболее часто используемым при этом является распределение Максвелла. Второй способ, который применен во фрагменте кода 54, начинается с задания скоростям случайных значений (строка 2). Вычисляемая в строке 4 сумма представляет собой полную (безразмерную) кинетическую энергию получившейся системы. Для того, чтобы сделать температуру в ячейке равной заданной величине  $T$ , необходимо воспользоваться соотношением, связывающим температуру и кинетическую энергию

$$T = \frac{1}{3Nk_B} \sum_{i=1}^N m_i |\vec{v}_i|^2,$$

и пронормировать получившиеся случайные значения скоростей, умножив их на коэффициент

$$\sqrt{\frac{3NT}{\sum_{i=1}^N |\vec{v}_i|^2}}$$

В строках 5–6 вычисляется этот коэффициент (factor) с учетом принятой схемы обезразмеривания, а в строке 7 осуществляется корректировка массива скоростей.

Фрагмент кода 54. Задание начальных скоростей

```

1 # равномерно распределенные случайные значения
2 v_ = np.random.uniform(-1,1,(natom,3))
3 # нормировка амплитуд скоростей
4 velsq=np.sum(v**2)
5 aheat=3.e0*natom*stepsq*temp
6 factor=np.sqrt(aheat/velsq)
7 v_*=factor

```

Функция вычисления массива сил приведена во фрагменте кода 55.

При этом под силами понимаются правые части уравнений (54), т. е. фактически безразмерные ускорения. Тем не менее, в соответствии с оригинальной программой будем называть их силами. Массив сил имеет размерность `natom x 3`, при этом в строке с номером  $i$  находятся три компоненты вектора силы, действующей на  $i$ -ую частицу.

Фрагмент кода 55. Вычисление массива сил

```

1 def forces():
2     f_ = np.zeros(natom*3).reshape(natom,3)
3     for i in range(natom-1):
4         xij = x_[i] - x_[i+1:]
5         xij[xij<-cubeh]+=cube
6         xij[xij>cubeh]-=cube
7         rsq=np.sum(xij*xij, axis=1)
8         rsqinv=1.0/rsq
9         r6inv=rsqinv*rsqinv*rsqinv
10        force_=np.einsum('ki,k->ki', xij,
11                          rsqinv*48*r6inv*(r6inv-0.5))
12        f_[i] += np.sum(force_, axis=0)
13        f_[i+1:] -=force_
14    return f_

```

Строка 4 приведенного кода позволяет без использования циклов создать массив разностей координат  $i$ -й точки и всех остальных точек, следующих за ней в массиве, т. е. массив векторов  $\vec{x}_{ij}, j = i + 1, \dots, N$ .

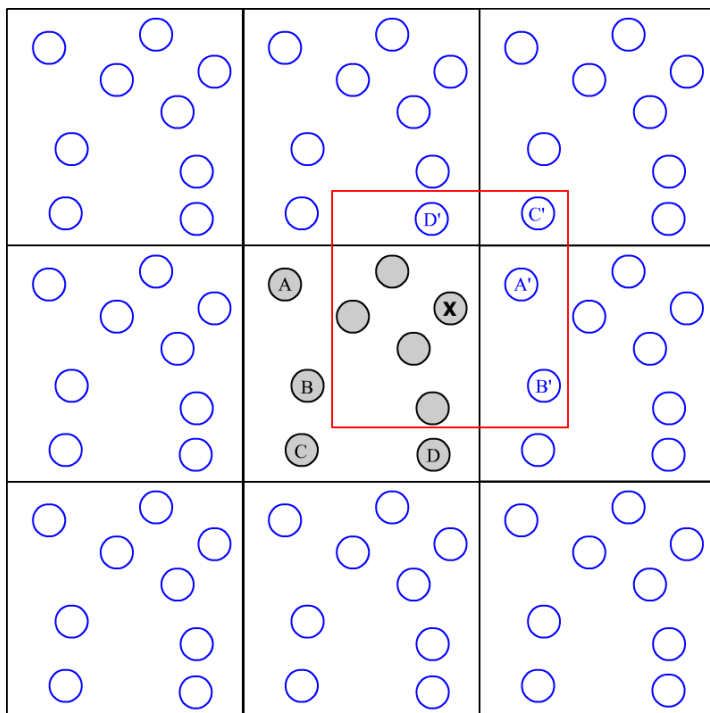


Рис. 65. Периодичность среды при вычислении сил

Отдельного внимания заслуживают строки 5 и 6 приведенного фрагмента, поскольку именно использованный в них прием позволяет считать малый изучаемый объем сколько-нибудь представительным. В молекулярной динамике при вычислении сил полагают, что рассматриваемая система частиц не изолирована, а повторяется периодически по всем пространственным направлениям. Эту идею иллюстрирует рисунок 65. Девять атомов изучаемой системы изображены серым цветом. При вычислении сил, действующих на атом, отмеченный крестиком, учитывается ровно восемь атомов. Но кроме атомов исходной системы рассматриваются и все их «образы», получившиеся в результате периодического продолжения рассматриваемой ячейки. В итоге вместо

атома А при вычислении сил будет использован атом А', вместо атома В – атом В' и т. д. При таком подходе существенным является еще допущение о том, что используемый потенциал является короткодействующим, а значит для вычисления сил в построенной периодической среде действительно достаточно ограничиваться атомами, «попавшими внутрь красного квадрата», или, более строго, лежащими внутри куба со стороной, равной длине изучаемой ячейки.

Программная реализация этой идеи достаточно проста: если расстояние по какой-либо из координат между атомами больше половины длины стороны куба, в котором расположена изучаемая система, то вторая частица перемещается симметрично в одну или другую сторону вдоль этой координаты.

Создаваемый в строке 7 массив `rsq` представляет собой массив квадратов расстояний  $\rho_{ij}^2$ , массив `rsqinv` – массив обратных к ним величин (строка 8), массив `rbinv` содержит величины  $\rho_{ij}^{-6}$ . Во всех случаях  $j > i$ .

В строках 10–12 реализуется вычисление сумм, присутствующих в уравнении (54). Строка 13 соответствует соотношению (49).

Интегрирование уравнений движения системы частиц по времени представлено в завершающем фрагменте кода 56. Схема интегрирования перенесена без изменений из оригинальной программы; она может быть записана следующим образом:

$$\begin{aligned} \vec{a}(t) &= \vec{F}(t), \\ \vec{v}(t + \Delta t) &= \vec{v}(t) + \Delta t \vec{a}(t), \\ \vec{x}(t + \Delta t) &= \vec{x}(t) + \vec{v}(t + \Delta t) \Delta t + \vec{a}(t) \frac{\Delta t^2}{2}. \end{aligned} \tag{55}$$

Умножим теперь первое из уравнений (55) на  $\Delta t^2/2$ , второе – на  $\Delta t$  и будем использовать переменную `a_` для величины  $\frac{1}{2} \vec{a} \Delta t^2$ , переменную `v_` для  $\vec{v} \Delta t$ , переменную `x_` для  $\vec{x}$ . Тогда три строки численной схемы (55) будут строго соответствовать строкам 6, 8 и 10 из фрагмента кода 56.

Очень важными являются строки 12–13, в которых осуществляется учет граничных условий. Здесь также предполагается безграничность среды и периодические свойства по всем направлениям.

Фрагмент кода 56. Интегрирование уравнений движения

```

1 # количество шагов по времени
2 maxeq = 500
3 kstep = 5
4 for ktime in range(1, maxeq+1):
5 # ускорения, умноженные на delta_t^2/2
6     a_ = forces()*stepsqh
7 # скорости, умноженные на delta_t
8     v_ += 2.0*a_
9 # положения частиц
10    x_ += v_+a_
11 # учет граничных условий
12    x_[x_<0] += cube
13    x_[x_>cube] -= cube
14 # вывод информации в файл и на экран
15    if ktime%kstep==0:
16        print(f'step = {ktime}\nx:\n {x_}', file=f)
17        print(f'step = {ktime}/{maxeq}')
18 print('dynamics successfully ended')
19 f.close()

```

Тогда, если атом покидает наш куб (например, выходит за правую границу, как частица А на рис. 66, или опускается за нижнюю границу, как частица В), то на ее место тут же приходит ее «копия» из периодически продолженной ячейки. В программе у частиц, вылетевших за пределы куба, необходимо соответствующим образом откорректировать координаты.

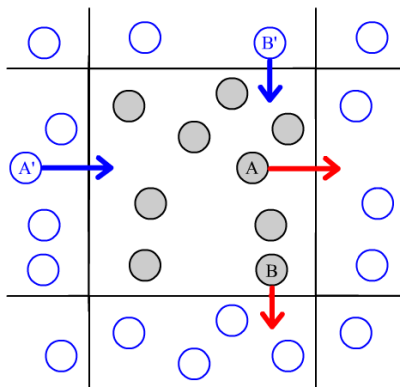


Рис. 66. Периодические граничные условия

Другим возможным случаем граничных условий является заданная на стенке куба постоянная температура  $T_1$ . Тогда принято считать [238] что частица при ударе о стенку на мгновение прилипает к ней, приобретает температуру стенки, а затем отлетает от стенки в произвольном направлении. Модуль вектора скорости отлетающей частицы может быть найден из условия

$$\frac{m|\vec{v}|}{2} = \frac{3}{2}k_B T_1.$$

На каждом шаге интегрирования необходимо проверять, не вылетела ли частица за стенку термостата, а если это произошло, то размещать такую частицу на стенке, находить модуль ее скорости и случайным образом выбирать направление этого вектора внутрь ячейки.

## ЛИТЕРАТУРА К ГЛАВЕ 6

- 6.1. Агафонов А. Н., Еремин А. В. Метод классической молекулярной динамики в моделировании физико-химических процессов: учеб. пособие [Электронный ресурс]. – Самара: Изд-во Самарского университета, 2017. – 68 с. – URL: <http://repo.ssau.ru/handle/Uchebnye-posobiya/Metod-klassicheskoi-molekulyarnoi-dinamiki-v-modelirovanii-fizikohimicheskikh-processov-Elektronnyi-resurs-ucheb-posobie-68492>  
Агафонов А.Н. Метод.pdf (дата обращения 01.02.2021)
- 6.2. Аксенова Е. В., Кшевецкий М. С. Вычислительные методы исследования молекулярной динамики [Электронный ресурс]. – СПб.: Изд-во СПбГУ, 2009. – 50 с. – URL: <http://stat.phys.spbu.ru/Metod/VychMethodMolDyn.pdf> (дата обращения 01.02.2021)
- 6.3. Каплан И. Г. Межмолекулярные взаимодействия. Физическая интерпретация, компьютерные расчеты и модельные потенциалы. – М.: БИНОМ. Лаборатория знаний, 2012. – 394 с.
- 6.4. Кривцов А. М., Кривцова Н. В. Метод частиц и его использование в механике деформируемого твердого тела // Дальневосточный математический журнал ДВО РАН. – 2002. – Т. 3, № 2. – С. 254–276
- 6.5. Селезнев А.А. Основы метода молекулярной динамики: Учебно-методическое пособие [Электронный ресурс]. – Саров, СарФТИ. 2017 г. 72 с. – URL: <http://sarfti.ru/wp-content/uploads/2014/05/Селезнев-А.А.->

Основы-метода-молекулярной-динамики.doc (дата обращения 01.02.2021)

- 6.6. Холмуродов Х.Т. МД-моделирование в химических исследованиях: от атомных фрагментов до молекулярных соединений. – Дубна, 2011. – 100 с.
- 6.7. Холмуродов Х. Т., Алтайский М. В., Пузынин И. В., Дардин Т., Филатов В.П. Методы молекулярной динамики для моделирования физических и биологических процессов [Электронный ресурс]. // Физика элементарных частиц и атомного ядра. – 2003. – Т. 34, вып. 2. – С. 474–515. – URL: <http://www1.jinr.ru/Pepan/v-34-2/v-34-2-5.pdf> (дата обращения 01.02.2021)
- 6.8. Цянь Сюэ-Сень. Физическая механика. – М.: Мир, 1965. – 544 с.
- 6.9. Alder В. J. , Wainwright Т. Е. Phase Transition for a Hard Sphere System // J. Chem. Phys. – 1957. – Vol. 27. – Pp. 1208–1209. doi: 10.1063/1.1743957
- 6.10. Benassi E. The Zero Point Position in Morse’s potential and accurate prediction of thermal expansion in metals // Chemical Physics. 2018. Vol. 515, Pp. 323–335. doi: 10.1016/j.chemphys.2018.09.005
- 6.11. Donald A. McQuarrie, John D. Simon. Physical Chemistry: A Molecular Approach. – University Science Books, 1997. – 1360 с.
- 6.12. Ercolessi F. A molecular dynamics primer [Электронный ресурс]. Spring College in Computational Physics, ICTP, Trieste, June 1997. – URL: <http://www.cems.uvm.edu/~gmirchan/classes/EE274/Images/Articles/MD%20Primer.pdf> (дата обращения 01.02.2021)
- 6.13. Gibson J. В., Goland A. N., Milgram M., Vineyard G. H. Dynamics of Radiation Damage // Physical Review. – 1960. – Vol. 120, issue 4. – Pp. 1229–1253. doi: 10.1103/PhysRev.120.1229
- 6.14. Girifalco L. A., Weizer V. G. Application of the Morse Potential Function to Cubic Metals // Physical Review. – 1959. – Vol. 114, issue 3. Pp. 687–689. doi: 10.1103/PhysRev.114.687
- 6.15. Halicioglu T., Pound G. M. Calculation of potential energy parameters form crystalline state properties // Physica Status Solidi (a). – 1975. – Vol. 30(2). – Pp. 619–623. doi:10.1002/pssa.2210300223
- 6.16. Hollingsworth S. A., Dror R.O. Molecular Dynamics Simulation for All // Neuron. – 2018. – Vol 99, issue 6. – Pp. 1129–1143. doi: 10.1016/j.neuron.2018.08.011



- 6.17. Pamuk H. Ö., Halicioğlu T. Evaluation of Morse parameters for metals // *Physica Status Solidi (a)*. – 1976. – Vol. 37(2). – Pp. 695–699. doi: 10.1002/pssa.2210370242
- 6.18. Rahman A. Correlations in the Motion of Atoms in Liquid Argon // *Physical Review*. – 1964. – Vol. 136, issue 2A. – Pp. A405–A411. doi: 10.1103/PhysRev.136.A405
- 6.19. Verlet L. Computer "Experiments" on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules // *Physical Review*. – 1967. – Vol. 159, issue 1. – Pp. 98–103. doi: 10.1103/PhysRev.159.98
- 6.20. Verlet L. Computer "Experiments" on Classical Fluids. II. Equilibrium Correlation Functions // *Physical Review*. – 1968. – Vol. 165, issue 1. – Pp. 01–214. doi: 10.1103/PhysRev.165.201

## ОГЛАВЛЕНИЕ

<b>ПРЕДИСЛОВИЕ .....</b>	<b>3</b>
<b>ГЛАВА 1. ИСПОЛЬЗОВАНИЕ ПАКЕТА VPYTHON ДЛЯ СОЗДАНИЯ 3D-АНИМАЦИЙ.....</b>	<b>7</b>
<b>Введение.....</b>	<b>7</b>
<b>Первое знакомство .....</b>	<b>8</b>
Установка.....	8
Холст.....	9
Анимация.....	12
<b>Математический аппарат VPython .....</b>	<b>13</b>
Математические функции.....	13
Операции с векторами.....	15
Построение графиков.....	16
<b>Базовые 3D-объекты.....</b>	<b>22</b>
Цилиндр.....	22
Сфера/Эллипсоид.....	25
«Простая» сфера.....	26
Параллелепипед.....	26
Стрелка.....	28
Присоединение стрелки.....	29
Спираль.....	30
Пирамида и конус.....	30
Кольцо.....	31
Пространственная ломаная.....	32
Экструзия.....	34
Вывод текстовой информации.....	35
<b>Общие характеристики объектов .....</b>	<b>38</b>
Работа с цветом.....	38
Прозрачность.....	39
Текстуры.....	40
Дополнительные атрибуты и методы.....	42
<b>Холст: свойства и события .....</b>	<b>45</b>
Создание холста.....	45
Основные характеристики холста.....	46
Представление содержимого холста на экране.....	47
Освещение сцены.....	50
Интерактивные элементы.....	50
<b>Литература к главе 1.....</b>	<b>57</b>

<b>ГЛАВА 2. ПРОЕКТНОЕ ЗАДАНИЕ</b>	
<b>«ВИЗУАЛИЗАЦИЯ МОЛЕКУЛЫ»</b>	<b>58</b>
<b>Формулировка задания</b>	<b>58</b>
<b>Базовые сведения</b>	<b>61</b>
Химическая информатика	61
Хранение и визуализация структурных данных	63
Формат XYZ	65
Формат MOL	65
Формат SDF	67
Формат PDB	69
Формат CML	73
<b>Пример выполнения задания</b>	<b>75</b>
Получение файла со структурой молекулы	75
Обсуждение структуры программы	78
Класс Atom	79
Класс Bond	82
Класс Molecule	86
<b>Литература к главе 2</b>	<b>89</b>
<b>ГЛАВА 3. МОДЕЛИРОВАНИЕ ДВИЖЕНИЯ</b>	
<b>МЕХАНИЧЕСКИХ СИСТЕМ</b>	<b>91</b>
<b>Численные методы решения задачи Коши</b>	<b>91</b>
Постановка задачи	91
Метод Эйлера и его модификации	94
Методы второго порядка	98
Решение задачи Коши с использованием библиотеки <code>scipy</code>	103
<b>Примеры моделирования механических систем</b>	<b>109</b>
Колесания тележки на нелинейно-упругой пружине	109
Полет снаряда	117
<b>Литература к главе 3</b>	<b>128</b>
<b>ГЛАВА 4. ПРОЕКТНОЕ ЗАДАНИЕ</b>	
<b>«ПРУЖИННАЯ МОДЕЛЬ ТВЕРДОГО ТЕЛА»</b>	<b>130</b>
<b>Формулировка задания</b>	<b>130</b>
<b>Пример выполнения задания</b>	<b>143</b>
<b>Литература к главе 4</b>	<b>154</b>
<b>ГЛАВА 5. ПРОЕКТНОЕ ЗАДАНИЕ</b>	
<b>«НЕБЕСНАЯ МЕХАНИКА»</b>	<b>156</b>
<b>Формулировка задания</b>	<b>156</b>
<b>Базовые сведения</b>	<b>161</b>
Примеры моделирования объектов солнечной системы	163
Задача трех тел	182
Визуализация решения Эйлера задачи трех тел	187

<b>Пример выполнения задания .....</b>	<b>195</b>
Переборные задачи при моделировании астрономических систем. ....	195
Двумерная визуализация периодических траекторий. ....	198
Трехмерная визуализация периодических траекторий. ....	204
<b>Литература к главе 5.....</b>	<b>210</b>
<b>ГЛАВА 6. ПРОЕКТНОЕ ЗАДАНИЕ</b>	
<b>«ЗНАКОМСТВО С МОЛЕКУЛЯРНОЙ ДИНАМИКОЙ» .....</b>	<b>213</b>
<b>    Формулировка задания.....</b>	<b>213</b>
<b>    Базовые сведения.....</b>	<b>217</b>
Молекулярная динамика. ....	217
Краткая историческая справка. ....	220
Потенциалы молекулярной динамики. ....	221
<b>    Пример выполнения задания .....</b>	<b>226</b>
<b>    Литература к главе 6.....</b>	<b>238</b>
<b>ОГЛАВЛЕНИЕ .....</b>	<b>241</b>

*Учебное издание*

**КАРЯКИН Михаил Игорьевич**

**Визуализация механических систем,  
процессов и явлений:  
проектные задания с использованием VPython**

Подписано в печать 28.06.2021 г.

Бумага офсетная. Формат 60×84<sup>1/16</sup>. Тираж 30 экз.

Усл. печ. лист. 14,18. Уч. изд. л. 9,0. Заказ № 8067.

Издательство Южного федерального университета.

Отпечатано в отделе полиграфической, корпоративной и сувенирной продукции  
Издательско-полиграфического комплекса КИБИ МЕДИА ЦЕНТРА ЮФУ.  
344090, г. Ростов-на-Дону, пр. Стачки, 200/1, тел (863) 243-41-66.

