

Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное
учреждение высшего образования
ПЕТРОЗАВОДСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

В. Б. Пикулев, С. В. Логинова

**Язык Python в приложении
к экспериментальным исследованиям**

Учебное электронное пособие

Петрозаводск
Издательство ПетрГУ
2019

УДК 004
ББК 32.973.2
ПЗ23

Издается по решению редакционно-издательского совета
Петрозаводского государственного университета

Рецензенты:

А. Д. Фофанов, доцент, доктор физ.-мат. наук;

Л. В. Щеголева, доцент, доктор техн. наук

Пикулев, Виталий Борисович.

ПЗ23 Язык Python в приложении к экспериментальным исследованиям [Электронный ресурс]: учебное электронное пособие / В. Б. Пикулев, С. В. Логинова; М-во науки и высшего образования Рос. Федерации, Федер. гос. бюджет. образоват. учреждение высш. образования Петрозавод. гос. ун-т. — Электрон. дан. — Петрозаводск: Издательство ПетрГУ, 2019. — 1 электрон. опт. диск (CD-R); 12 см. — Систем. требования: PC, MAC с процессором Intel 1,3 ГГц и выше; Windows, MAC OSX; 256 Мб; видеосистема: разрешение экрана 800×600 и выше; графический ускоритель (опционально); мышь или другое аналогичное устройство. — Загл. с этикетки диска.

ISBN 978-5-8021-3548-8

В пособии на практических примерах рассмотрены возможности применения языка Python в приложении к физическому эксперименту от программирования устройств сбора данных до обработки, хранения и визуализации результатов измерений. Пособие предназначено для студентов физико-технического направления подготовки, для инженеров и научных работников, занимающихся автоматизацией эксперимента и компьютерной обработкой результатов научных исследований. Пособие адресуется тем, кто владеет навыками программирования, но еще не знаком с языком Python.

УДК 004
ББК 32.973.2

ISBN 978-5-8021-3548-8

© Пикулев В. Б., Логинова С. В., 2019

© Петрозаводский государственный
университет, 2019

Оглавление

Оглавление	3
Введение	4
Глава 1. Введение в программирование на Python	5
1.1 Среда разработки программ.....	5
1.2 Основные языковые конструкции.....	6
1.3 Построение графиков: библиотека matplotlib	11
Глава 2. Получение и визуализация данных эксперимента	19
2.1 Подключение устройства сбора данных NI USB 6351	19
2.2 Сохранение данных в файл.....	22
2.3 Загрузка данных из файла и их визуализация.....	24
2.4 Работаем с осциллографом на Python.....	25
2.5 Комментарии.....	27
Глава 3. Обработка результатов эксперимента	29
3.1 Сглаживание экспериментальных данных.....	29
3.2 Разбиение спектральных кривых на гауссианы.....	30
3.3 Работа с файлами данных формата JDX.....	36
Глава 4. Автоматизация спектроскопического эксперимента	39
4.1 Проектирование управляющей программы	39
4.2 Взаимодействие с контроллером Arduino	40
4.3 Работа с платой сбора данных National Instruments	43
4.4 Режим измерения	43
4.5 Просмотр и сохранение результатов измерений	46
4.6 Организация работы с базой данных	48
4.7 Заключительные комментарии	52
Список литературы	53

Введение

Сфера разработки программного обеспечения предполагает очень быстрое освоение современных концепций и понятий, в этой связи формирование конкурентоспособного выпускника не может быть ориентировано «на перспективу», фактически он должен быть конкурентоспособен уже на стадии своего обучения. Можно даже сказать, что программист, способный быстро войти в суть решаемой проблемы, быстро обучиться и быстро применить полученные знания на практике, в настоящее время является наиболее востребованным для индустрии информационных технологий. Нелишне напомнить, что современная система получения знаний уверенно смещается в сторону дистанционного обучения, а обилие видеоуроков в свободном доступе, описывающих «первые шаги» в каком-либо языке программирования, способствует этому процессу. Язык *Python* в этом смысле не только не исключение — а характерный пример языка, который в настоящее время большинство школьников, студентов, а тем более — разработчиков со стажем — осваивают самостоятельно и в сжатые сроки.

Если взглянуть на рейтинг 2019 года самых востребованных языков программирования от авторитетных групп [1] или [2], в первой тройке языков программирования мы видим *Python*. Этот язык постепенно набирал популярность, полагаясь на ключевое утверждение «простое лучше, чем сложное». Иными словами, это язык, который сочетает продуманный и лаконичный синтаксис с огромным количеством возможностей, предоставляемых многочисленными модулями и библиотеками.

Следует заметить, что для технического направления подготовки студентов программирование не является самоцелью, а выступает как средство решения какой-либо физической или технологической задачи. Студенты должны знать, какие программные средства нужно применить, чтобы наиболее эффективно решить поставленную задачу. Подчас даже построение графика для научной статьи может стать проблематичным, причём наиболее известный студентам пакет *Microsoft Excel* оказывается не всегда применим. Бакалаврам обычно не хватает практики использования таких совершенных систем инженерных научных расчётов, как *MATLAB*, *SciLab*, *MathCAD*, *Wolfram Alpha*. Создание виртуальных инструментов *LabView* также требует опыта и освоения специфического стиля программирования. Кроме того, *MATLAB* и *LabView* требуют покупки дорогостоящих лицензий.

Преимущества языка *Python* состоят в том, что, во-первых, он является свободно распространяемым программным продуктом, при этом для него в настоящее время разработано огромное количество модулей, что делает его не менее функциональным, чем *MATLAB*. В частности, практически однотипными командами можно быстро оформить двумерные и трехмерные графики, провести статистическую обработку данных. При этом можно писать программу в привычном для себя редакторе, а среда *Python* идеальна для отладки созданного алгоритма. Во-вторых, на этом языке довольно легко создавать программы, управляющие периферийным оборудованием: осциллографом, вольтметром, платой сбора данных и т. п. Единственное существенное ограничение — для процессов реального времени будут необходимы откомпилированные фрагменты кода, созданные, например, на языке Си. И, наконец, *Python* удобен для создания распределённой автоматизированной системы, включающей в себя сбор данных с датчиков и периферийного оборудования по различным сетевым протоколам и интерфейсам, работу с базами данных и, наконец, публикацию динамических страниц на web-сайте [3]. На каждом рабочем месте, под любой операционной системой, *Python* может быть легко настроен в соответствии с теми задачами, которые будет решать конкретный пользователь.

Авторы, имея опыт работы с различными языками программирования, а также с пакетами *MATLAB* и *LabView*, хотели бы показать в данном пособии, что язык *Python* является очень удобным средством в решении актуальных задач автоматизации физического эксперимента и обработки результатов измерений. Освоение современного и перспективного языка *Python* молодыми специалистами станет надёжным фундаментом в сфере их профессиональной деятельности.

Глава 1. Введение в программирование на Python

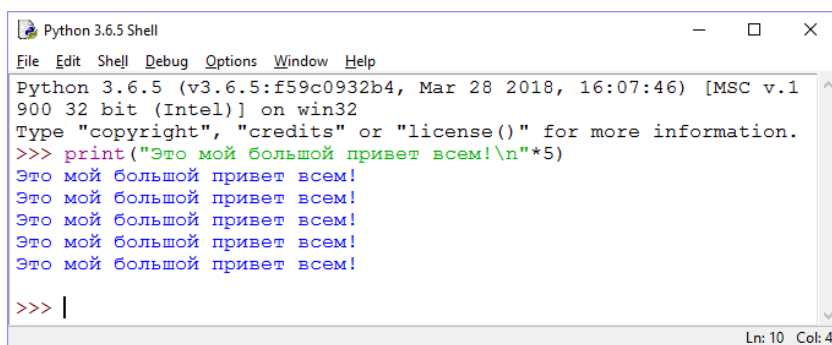
1.1 Среда разработки программ

Среда Python может быть установлена под различные операционные системы. Инсталлятор языка и подробные инструкции можно получить на сайте поддержки [4]. Имея опыт использования Python под операционными системами Microsoft Windows, Debian Linux и Raspbian (для микрокомпьютера Raspberry Pi), в данном пособии мы ограничим описание практических примеров Python средой Windows как наиболее распространённой среди пользователей, включая в их число и студентов, работающих в научных лабораториях. Для других операционных систем некоторые из приведенных примеров могут потребовать переработки, а некоторые модули и пакеты следует заменить на их эквиваленты в этих системах.

Также следует обратить внимание, что все рассматриваемые примеры написаны для *третьей* версии языка Python. Мы использовали актуальную на момент выпуска пособия версию Python 3.6.5. Следовательно, пользователи, еще не ушедшие с Windows XP, также могут встретиться с рядом ошибок при установке пакетов и запуске предложенных примеров. Поэтому, во избежание ненужных проблем, в качестве базовой платформы мы предлагаем использовать 32- или 64-разрядную Windows начиная с версии 7.

После установки Python вы получаете в свое распоряжение среду разработки IDLE, которая включает в себя терминальное окно *Python Shell*, где можно комфортно работать в режиме командной строки (рис. 1) и достаточно простой текстовый редактор, в котором удобно писать и отлаживать небольшие программы. Для запуска редактора достаточно выбрать пункт меню *File* → *New File* и можно начинать. Запускается программа на выполнение с помощью клавиши F5 или из меню *Run* → *Run Module*. Использование встроенного редактора, как и самой IDLE, не является обязательным: можно использовать любую универсальную среду разработчика программ или мощный редактор типа *Atom* или *Microsoft Code*. Файл с кодом программы обычно имеет расширение *.py*.

Второй доступный способ работы с Python использует режим командной строки Windows (команда операционной системы *cmd*). В окне Windows-терминала введите команду *python*, и оболочка языка запустится, что можно увидеть по характерному символу приглашения *>>>* (рис. 2). Если «команда не найдена» — следует добавить в переменную окружения PATH путь к каталогу, где установлен Python. Выход из оболочки — команда *exit()*.



```
Python 3.6.5 Shell
File Edit Shell Debug Options Window Help
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 16:07:46) [MSC v.1
900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> print("Это мой большой привет всем!\n"*5)
Это мой большой привет всем!
Это мой большой привет всем!
Это мой большой привет всем!
Это мой большой привет всем!
Это мой большой привет всем!
>>> |
```

Рис. 1. «Hello, world!» в оболочке Python

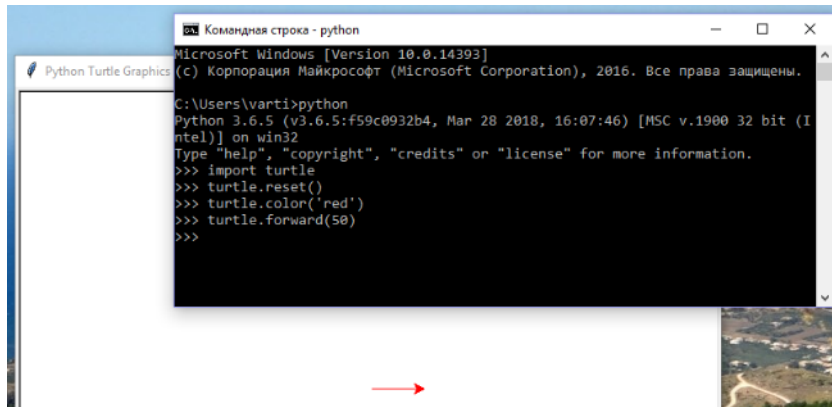


Рис. 2. Запуск Python из командной строки

Помимо обычного интерфейса Python, для использования системы в качестве своего рода интеллектуального блокнота удобно использовать расширение *iPython* — это интерактивная оболочка, поддерживающая дополнительный набор команд, цветовую подсветку кода и режим подсказок при наборе команд (рис. 3). Особенно удобно использовать оболочку *iPython* вместе с графической библиотекой *matplotlib*, о которой будет рассказано в разделе 1.3.

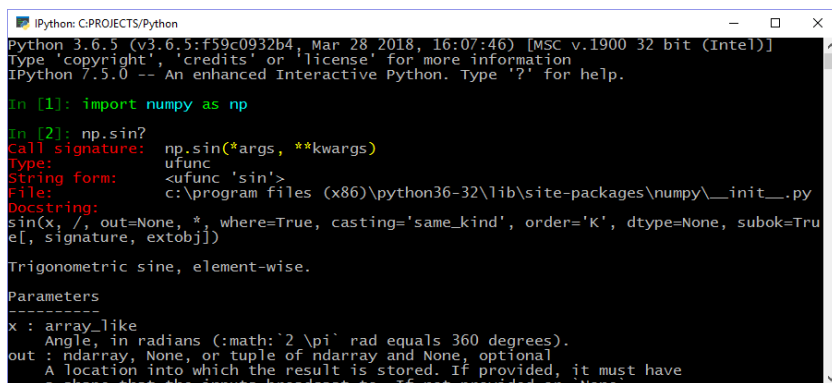


Рис. 3. Оболочка iPython: вызов справки по функции

Для установки этой оболочки, равно как и всех других пакетов, упоминаемых в данном пособии, следует воспользоваться командной строкой Windows в режиме администратора: в командной строке следует набрать:

```
> pip install ipython
```

и соответствующее расширение языка будет установлено. Для запуска оболочки можно создать ярлык Windows, в свойствах которого описать объект запуска приблизительно таким образом: «C:\Windows\System32\cmd.exe /k ipython». Если вы хотите еще большего сходства с интеллектуальной записной книжкой (или же вам не нравится работать в режиме командной строки), можно установить расширение *Jupyter Notebook* и после этого интерактивно взаимодействовать с Python в окне Интернет-браузера.

1.2 Основные языковые конструкции

Python является регистрозависимым и строкозависимым языком. Для объединения команд не используются фигурные скобки или какие-либо ключевые слова — в Python зависимость команд определяется фиксированными отступами от начала строки. Рассмотрим пример кода, реализующего поиск простых чисел (чисел, которые делятся только на единицу и на самих себя):

```

1  i = 2; i_max = 100
2  print('Простые числа от',i,' до ', i_max)
3  while i <= i_max:
4      j = 2
5      prime = True
6      while j <= i/j:
7          if i%j == 0:
8              prime = False
9              break
10         j += 1
11     if prime: print (i, end='\t')
12     i += 1

```

Результат работы программы:

```

Простые числа от 2 до 100
2  3    5    7    11   13   17   19   23
   29   31   37   41   43   47   53   59
   61   67   71   73   79   83   89   97

```

В этом алгоритме используются два цикла *while*: первый включает в себя строки с 3 по 12, второй, вложенный в первый, содержит строки с 6 по 10. Отступы (по умолчанию 4 пробела) позволяют точно идентифицировать блоки команд и, одновременно, красиво оформить текст программы. Блок операторов внутри конструкции *if* (строки 7—9) также определен с помощью отступов. Для второго условия *if*, содержащего всего одну команду, можно обойтись без отступов (строка 11). Но двоеточие, разграничивающее условие и выполняемое действие является обязательным (строки 3, 6, 7, 11).

Точка с запятой после команд не нужна, но может использоваться в качестве разделителя команд, если они написаны на одной строке (пример — строка 1). Символ '\t' в операторе вывода *print* — это символ табуляции.

Python является языком с *достаточно строгой* динамической типизацией. В частности, числа в Python представлены классами *int* (длинные целые), *float* (вещественные) и *complex* (комплексные). Явное объявление типа переменной отсутствует. При выполнении программы переменная может получить или изменить тип в зависимости от присвоенного значения. Значения логического типа *True* и *False* соответствуют целочисленным значениям **1** и **0**. Поэтому условие в строке 7 можно переписать так:

```
if not(i%j):
```

Математическая операция *%* возвращает остаток от деления, оператор *break* прерывает ближайший к нему (внутренний) цикл, Си-подобная конструкция $i += 1$ является краткой записью выражения $i = i + 1$. На самом деле переменная *prime* в коде лишняя: следующий пример выглядит гораздо короче, но работать будет аналогично предыдущему:

```

1  for i in range(2, 101):
2      j = 2
3      while j <= i/j:
4          if not(i%j): break
5          j += 1
6      else: print (i, end='\t')

```

Здесь внешний цикл реализован с помощью конструкции *for*. В языке Python цикл *for* всегда реализует перебор элементов некоторого набора, указанного после ключевого слова *in*. Например, это может быть какой-либо список. В данном случае такую структуру создает функция *range*, в качестве аргументов этой функции заданы начало и конец целочисленной последовательности (шаг по умолчанию равен единице).

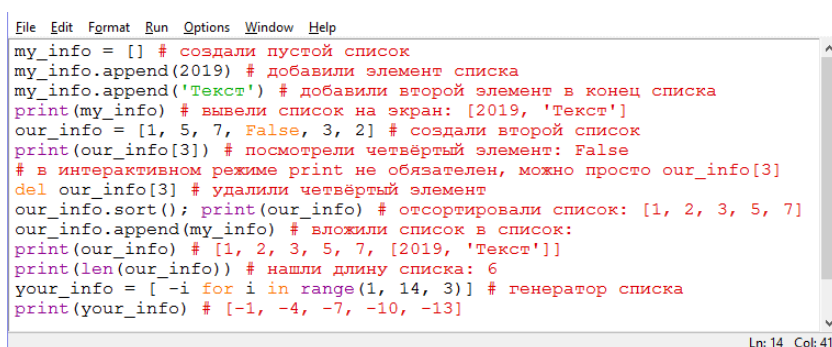
Конструкция *else* в строке 6 относится не к оператору *if* (для оператора *if* в данном контексте она не нужна), а к оператору *while* и означает выполнение действия только в случае, если цикл был завершен «корректно», т. е. без использования *break*.

И, наконец, пример короткой, самой быстрой из предложенных и специфичной именно для Python реализации алгоритма поиска простых чисел:

```
1 import math
2 print('2\t')
3 for i in range(3,101,2):
4     if all(i%j for j in range(2, 1+int(math.sqrt(i)))):
5         print (i, end='\t')
```

Здесь нам потребовалось импортировать модуль математических функций (строка 1), из которой понадобился корень квадратный для более корректного ограничения перебираемых делителей. Поскольку, кроме числа 2, других четных чисел среди простых нет, функция *range* теперь выдаст нам только нечётные числа (третий аргумент — шаг). Конструкция *if all* проверяет каждый элемент последовательности на выполнение указанного условия, а сама последовательность значений формируется с помощью генератора списка *for* в строке 4.

Заметим, что в Python нет характерного для других языков программирования понятия *массив*. Есть списки, кортежи, множества и словари, которые в целом гораздо функциональнее обычных массивов. Наиболее близким к «классическому» массиву является *список* (рис. 4).



```
File Edit Format Run Options Window Help
my_info = [] # создали пустой список
my_info.append(2019) # добавили элемент списка
my_info.append('Текст') # добавили второй элемент в конец списка
print(my_info) # вывели список на экран: [2019, 'Текст']
our_info = [1, 5, 7, False, 3, 2] # создали второй список
print(our_info[3]) # посмотрели четвёртый элемент: False
# в интерактивном режиме print не обязателен, можно просто our_info[3]
del our_info[3] # удалили четвёртый элемент
our_info.sort(); print(our_info) # отсортировали список: [1, 2, 3, 5, 7]
our_info.append(my_info) # вложили список в список:
print(our_info) # [1, 2, 3, 5, 7, [2019, 'Текст']]
print(len(our_info)) # нашли длину списка: 6
your_info = [-i for i in range(1, 14, 3)] # генератор списка
print(your_info) # [-1, -4, -7, -10, -13]
```

Рис. 4. Работа со списками, редактор IDLE Python

Список является изменяемой последовательностью, т. е. можно различным образом модифицировать элементы списка. Следует быть внимательным при присваивании списка другой переменной — на самом деле в этом случае происходит передача ссылки на один и тот же список. Для выбора фрагмента списка (что называется *срезом*) используется следующий синтаксис (отрицательный индекс означает смещение от конца списка):

```
>>> our_info
[1, 2, 3, 5, 7, [2019, 'Текст']]
>>> our_info[2:4]
[3, 5]
>>> our_info[3:]
[5, 7, [2019, 'Текст']]
>>> our_info[-1][0]
2019
>>>
```

Скопировать список в другой список можно, передав ему срез:

```
>>> copy_info = our_info[:]
>>> copy_info[1] = 100
>>> print(our_info)
[1, 2, 3, 5, 7, [2019, 'Текст']]
>>> print(copy_info)
[1, 100, 3, 5, 7, [2019, 'Текст']]
```

В отличие от списка, *кортеж* является неизменяемой последовательностью. Кортеж оформляется в круглые скобки:

```
>>> days = ('ПН', 'ВТ', 'СР', 'ЧТ', 'ПТ', 'СБ', 'ВС')
>>> days[1]
```



```
'BT'  
>>> my_info = ('Один элемент',) # запятая обязательна!
```

Работа с кортежами похожа на работу со списками, только изменять структуру кортежа нельзя, т. е. нельзя добавлять, удалять и изменять на уровне кортежа значения его элементов. Кортеж, как и список, может быть автоматически сгенерирован из строкового значения:

```
>>> char_tuple = tuple('Строка')  
>>> char_tuple  
( 'c', 'т', 'р', 'о', 'к', 'а' )
```

Словарь очень напоминает ассоциативный массив в других языках программирования и представляет собой неупорядоченную коллекцию значений произвольных типов. Словарь состоит из набора пар «ключ: значение». Для задания словаря используются фигурные скобки:

```
>>> my_d = {}  
>>> my_d['test'] = 'проверочное задание'  
>>> my_d['петроглиф'] = 'древний рисунок на камне'  
>>> my_d[1000] = 'thousand'  
>>> my_d  
{ 'test': 'проверочное задание', 'петроглиф': 'древний рисунок на камне', 1000:  
'thousand' }  
>>> my_d['test']  
'проверочное задание'  
>>> my_d[1] # такого ключа нет  
KeyError: 1
```

В качестве ключа может выступать любой неизменяемый тип данных, в том числе и кортеж. Словарь является обновляемым: можно изменить значение для ключа, можно добавить и удалить любую пару «ключ: значение»:

```
>>> my_d['петроглиф'] = 'доисторический рисунок на камне'  
>>> del my_d[1000]  
>>> my_d.keys() # выводит все значения ключей  
dict_keys(['test', 'петроглиф'])  
>>> for k, v in my_d.items():  
    print('ключ: %s, значение: %s' % (k, v))
```

```
ключ: test, значение: проверочное задание  
ключ: петроглиф, значение: доисторический рисунок на камне
```

Здесь использован цикл *for* по словарю и вывод отформатированной строки для функции *print*. Уже в последнем примере можно заметить, что приёмником результата (слева от знака равенства или от ключевого слова *in*) в Python могут выступать несколько переменных, перечисленных через запятую. Одновременное присваивание нескольким переменным является очень удобной особенностью этого языка:

```
>>> a, b, c = 0, 'значение', False  
>>> print(a, b, c)  
0 значение False  
>>> b, a = a, b # меняем местами значения переменных a и b  
>>> print(a, b, c)  
значение 0 False  
>>> a, b, c = c, a, b # можно и так  
>>> del c # удалили имя переменной
```

Знак ***, стоящий перед переменной, означает, что в эту переменную войдут все оставшиеся справа значения:

```
>>> a, *b = 'zzz', ['это', 'список'], 5+3j  
>>> print(a, '\n', b)  
zzz  
[['это', 'список'], (5+3j)]
```

Чтобы присвоить нескольким переменным одно значение, достаточно написать так:

```
>>> x = y = z = 0
```

Тип *множество* в Python содержит неповторяющийся уникальный набор данных. Для задания множества также используются фигурные скобки. Ниже показан пример преобразования списка во множество и обратно для устранения повторяющихся значений:

```
>>> my_list = [1, 17, 25, 1, 31, 17, 19, 25, 1]
>>> my_set = set(my_list)
>>> print(my_set)
{1, 17, 19, 25, 31}
>>> my_list = list(my_set)
>>> print(my_list)
[1, 17, 19, 25, 31]
```

Для множества, как и для других вышеперечисленных последовательностей, можно использовать *for*-генератор:

```
>>> D = {i**2 for i in range(10)}
```

Функции в Python определяются с помощью ключевого слова *def*. Принадлежащие функции команды также выделяются отступами. Приведем простейший пример функции, вычисляющей факториал числа *n*:

```
def fact(n):
    f = 1
    while n > 1:
        f *= n; n -= 1
    return f
```

После ключевого слова *return* указывается имя переменной, значение которой возвращается в основную программу. Проверим работу созданной функции:

```
>>> fact(1)
1
>>> fact(3)
6
>>> fact(100)
9332621544394415268169923885626670049071596826438162146859296389521759999322991
560894146397615651828625369792082722375825118521091686400000000000000000000000
```

Функции поддерживают рекурсию. Ниже дана реализация вычисления факториала с использованием вызова функцией самой себя:

```
def fact(n = 0):
    if n == 0:
        return 1
    return fact(n-1) * n
```

В скобках в первой строке для примера дано значение по умолчанию для *n*. Количество вызовов *return* внутри функции не лимитируется, поскольку всё равно выполнится только один из них.

Приведем пример функции, которая вычисляет (точно) возраст в годах по дате рождения. Правда, без модуля *datetime* здесь не обойтись:

```
from datetime import datetime as dt

def age(birthDateStr):
    birthDate = dt.strptime(birthDateStr, "%d.%m.%Y")
```

```
tdelta = dt.today() - birthDate
return int(tdelta.days/365.243)
```

Дата рождения задается в виде обычной текстовой строки, которая затем преобразуется в формат даты-времени. Результат возвращается в целочисленном формате. Проверим работу этой функции:

```
>>> age('07.11.1952')
66
>>> dt.today().date().strftime('%d.%m.%Y')
'08.06.2019'
>>> D='23.08.2018'; age(birthDateStr = D)
0
```

В последнем случае для примера мы воспользовались возможностью Python явно указывать параметр функции по имени при передаче ему значения (ключевой параметр). Это удобно, если параметров функции много, для них установлены значения по умолчанию, а мы хотим изменить значение лишь одного из параметров.

В следующем примере аргументов у функции может быть произвольное количество, а ключевое слово *return* вообще отсутствует:

```
def printAll(*args):
    for item in args:
        print(item)
```

Результат вызова функции:

```
>>> L = ['зима', 'лето']
>>> printAll(True, 12345, L)
True
12345
['зима', 'лето']
```

1.3 Построение графиков: библиотека *matplotlib*

Построение разнообразных графиков на языке Python на самом деле требует весьма небольшого кода, если обратиться за помощью к пакету *matplotlib*. Эта библиотека воспроизводит визуализацию данных в стиле пакета MATLAB, так что пользователи, работавшие с этим пакетом или с его общедоступной альтернативой SciLab, смогут легко перейти на использование Python. Для работы с библиотекой в первую очередь следует её установить (командная строка Windows в режиме администратора):

```
> pip install matplotlib
```

Кроме того, здесь и далее нам потребуется библиотека *numpy*, которую также потребуется установить:

```
> pip install numpy
```

Ниже представлена простейшая программа, которая строит график функции $\sin(x)/x$ на интервале от -10 до 10 . На скриншоте (рис. 5) показан результат ее работы.

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3 X = np.arange(-10, 10, 0.1)
4 Y = np.sin(X)/X
5 plt.plot(X, Y)
6 plt.show()
```

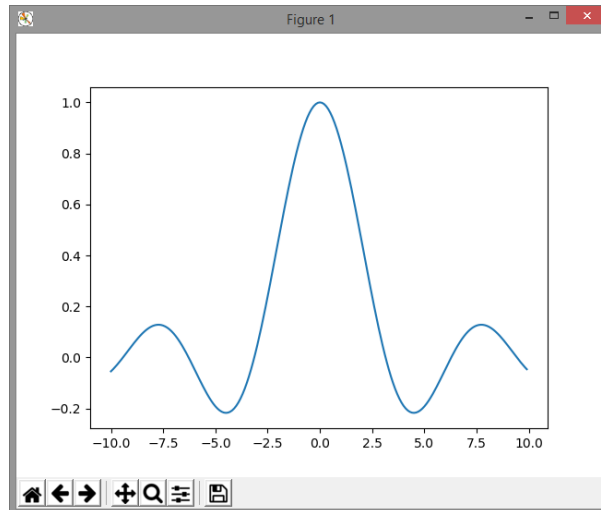


Рис. 5. Результат построения графика функции

Объясним некоторые особенности кода. В строках 1 и 2 импортируются библиотеки *matplotlib* и *numpy*. Если класс *pyplot*, очевидно, необходим для отображения графиков, необходимость *numpy* заключается в математической обработке массивов также в стиле пакета MATLAB (SciLab). Популярной особенностью MATLAB является возможность очень лаконично, без каких-либо вложенных циклов, описывать операции с векторами и матрицами. Это, безусловно, в стиле Python. Именно поэтому для удобства обработки данных здесь и далее будем пользоваться возможностями этой библиотеки, содержащей все необходимые функции для работы с многомерными массивами. Однако сказанное не означает, что *pyplot* не построит график для обычного списка Python. Чуть ниже мы приведём пример использования *pyplot* без *numpy*.

А пока вернемся к коду рассматриваемого примера. В строке 3 создается одномерный *numpy*-массив из монотонно увеличивающихся значений от -10 до 10 (не включая верхнюю границу) с шагом 0.1 . То есть, массив будет таким: $[-10.0, -9.9, -9.8, -9.7, \dots, 9.7, 9.8, 9.9]$. Он содержит:

```
>>> len(x)
200 # элементов
```



Ранее говорилось, что в Python имеется функция *range*, которая создает перечисление по заданным условиям. Однако, к сожалению, *range* работает только с целыми числами, что для этого примера неприемлемо.






Далее, массив Y — это значения нашей функции. В строке 4 мы вычисляем $\sin(X)/X$ для всех значений X . Именно для такой простой записи нам потребовалась библиотека *numpy*. Если вы захотите поэкспериментировать и попытаетесь использовать синус из базовой библиотеки Python — модуля *math*, код в строке 4 придется делать более громоздким, например, таким:

```
Y = np.array([math.sin(x)/x for x in x])
```

И, наконец, в строке 5 вызывается метод *plot* с двумя аргументами X и Y . Это единственная команда, которая самостоятельно формирует графическое окно — поле вывода графика, рассчитывает масштаб по осям координат и выводит график функции. Все настройки окна выполнены в данном случае по умолчанию, но их будет легко модифицировать по вашему усмотрению.

Если вы будете использовать *iPython*, график тотчас же появится на экране. Если же вы работаете в среде *IDLE Python* или в командной строке Python, обязательно потребуется строка 6.

Обратите внимание на кнопки с пиктограммами в нижней строке окна графика. Дело в том, что само окно интерактивно, и, пока оно не закрыто, можно масштабировать график по своему усмотрению. После нажатия на кнопку  появляется курсор в форме креста, с помощью которого на поле графика можно левой клавишей мышки выделить прямоугольную область, которая и будет увеличена. После нажатия на  с помощью мышки можно сдвигать изображение графика.

С помощью кнопки  можно вернуться к исходному виду графика. Клавиши   используются для движения по истории созданных видов. Клавиша  открывает окно настройки расположения поля графика в графическом окне. И, наконец, клавиша  вызывает диалог сохранения изображения в файл на диске в выбранном формате (*png*, *pdf*, *eps*, *svg* и др.).

Для ответа на вопрос, можно ли построить график этой же функции, не используя библиотеку *numpy*, запустите представленный ниже пример.

```
1 import matplotlib.pyplot as plt
2 import math as m
3
4 X = [-9.999999999 + 0.1*i for i in range(200)]
5 Y = [m.sin(x)/x for x in X]
6 plt.plot(X, Y)
7 plt.show()
```

Весьма показательным для оценки качества вычислений является введенное нами по необходимости смещение в массиве *X*, чтобы предотвратить формальное деление на нуль в строке 5. В предыдущем примере неопределённость $0/0$ была устранена автоматически. В дальнейших примерах мы будем преимущественно использовать *numpy*.

Заметим, что библиотека *matplotlib* позволяет использовать весьма разнообразные способы графического представления данных, в том числе создавать диаграммы различного вида, трехмерные анимированные графики, различным образом позиционировать несколько графиков на одном поле вывода, управлять отображением осей координат и др. В нашем случае мы ограничимся обзором базовых возможностей библиотеки, которые потребуются нам для решения дальнейших задач. Давайте поэкспериментируем с отображением графика с использованием синтаксиса, типичного для MATLAB. Изменим строку 6 так:

```
plt.plot(X, Y, 'r')
```

Получили кривую красного цвета. В качестве третьего аргумента возможны варианты *y*, *m*, *s*, *r*, *g*, *b*, *k*. Введем маркеры для точек графика:

```
plt.plot(X, Y, 'k+')
```

Линия исчезла, но появились маркеры черного цвета в виде крестиков. Возможны варианты *.*, *o*, *x*, *+*, ***, *s*. Пурпурная штриховая линия может быть задана так:

```
plt.plot(X, Y, 'm--')
```

Возможны варианты: *-*, *:*, *-.*, *--*. И вот еще один пример:

```
plt.plot(X, Y, 'k:o')
```

Третий параметр использует строковый формат вида «[цвет] [линия] [маркер]». Результат (с использованием интерактивных возможностей графика) приведен на рис. 6.

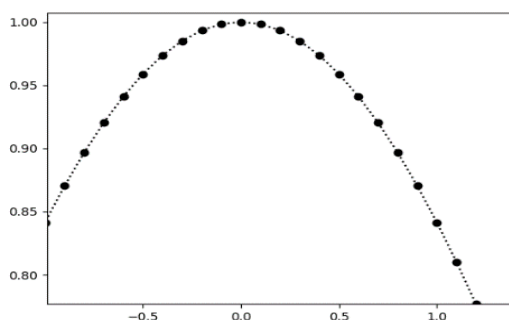


Рис. 6. Дизайн кривой графика функции

С точки зрения MATLAB возможности дизайна отображаемых кривых были бы исчерпаны вышеописанными примерами, но только не для Python. Замечательная особенность этого языка заключается в возможности передавать значения в функцию с явным указанием имен параметров. Таким образом, предыдущую строку программы можно было переписать так:

```
plt.plot(x, y, color='black', marker='o',
         linestyle='dotted')
```

Порядок перечисления имен параметров и их количество не важны. Попробуем изменить дизайн кривой так, чтобы это была зеленая штриховая линия толщиной 1.2 pt. с красными ромбовидными маркерами с тонкой синей окантовкой. Итак:

```
plt.plot(x, y, linestyle='dashed', color='green',
         linewidth = 1.2, marker='d', markersize = 6,
         markeredgecolor = 'blue', markerfacecolor = 'red',
         markeredgewidth = 0.5)
```

Построить несколько графиков на одном поле можно с помощью последовательных вызовов команды *plot*:

```
1  ...
2  x1 = np.arange(-10, 10, 0.1)
3  y1 = np.sin(x1)/x1
4  x2 = np.arange(0, 10, 0.2)
5  y2 = np.log(x2) # вычисляется натуральный логарифм
6
7  plt.grid() # включаем сетку графика
8  plt.plot(x1, y1, '#AA0000', label='кривая sin(x)/x')
9  plt.plot(x2, y2, '#00AA00', label='кривая ln(x)')
10 plt.legend(loc='upper left') # отображаем легенду
11 plt.show() # показываем результат
```

Результат представлен на рисунке 7. В упрощенном варианте вместо строк 8 и 9 можно, как в MATLAB, использовать только один вызов *plot*:

```
plt.plot(x1, y1, 'red', x2, y2, 'blue')
```

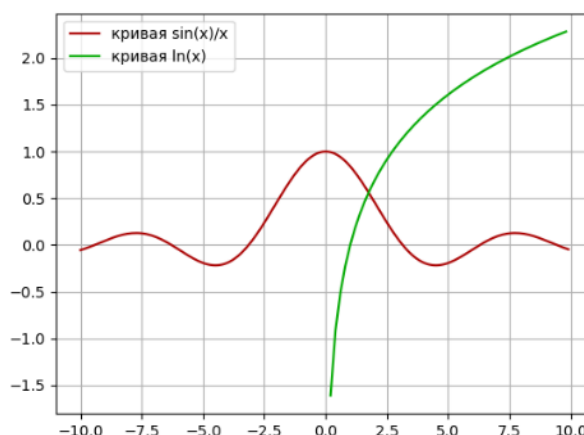


Рис. 7. Несколько графиков с легендой

Поработаем с видом осей координат. В следующем примере отобразим графики функций в логарифмическом формате по оси ординат, точно установим пределы по осям координат и дадим осям формальные наименования (рис. 8).

```
1  ...
2  plt.grid() # отображаем сетку графика
3  plt.ylim(1, 1E4) # задаём пределы по оси y
```

```

4 plt.yscale('log') # логарифмические координаты по Y
5 plt.plot(x1, Y1, '#AA0000', label='кривая 1+x*x')
6 plt.plot(x2, Y2, '#00AA00', label='кривая exp(x)')
7 plt.legend(loc='upper left') # отображаем легенду
8 plt.xlabel('Ось X')
9 plt.ylabel('Ось Y')
10 plt.show() # показываем результат

```

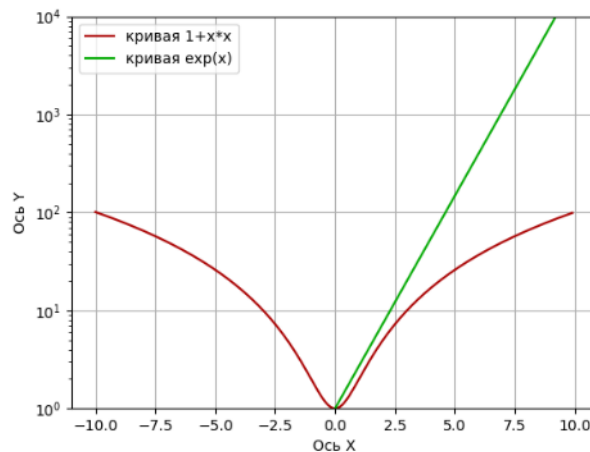


Рис. 8. Логарифмическая ось, подписи по осям

Оценка погрешностей измерений, безусловно, является важной составляющей экспериментальных исследований. Рассмотрим на следующем примере, как библиотеки *numpy* и *matplotlib* справляются с такими рутинными вычислительными задачами. Изобразим график усредненных значений некоего набора данных с указанием ошибки измерений в каждой точке (рис. 9).

```

1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 n_rows = 20 # число "экспериментальных" точек
5 n_cols = 10 # число измерений на каждой точке
6 X = np.linspace(0, 5, n_rows) # значения по абсциссе
7 Y = np.zeros((n_rows, n_cols)) # исходная нулевая матрица
8 for i in range(n_cols): # заполняем матрицу
9     Y[:,i] = np.sin(X) + 0.3 * np.random.randn(20)
10 Y_mean = Y.mean(axis=1) # среднее по каждой строке
11 Y_std = Y.std(axis=1, ddof=1) # стандартное отклонение
12 plt.figure(figsize=(9, 6)) # размер поля вывода
13 plt.errorbar(X, Y_mean, yerr=Y_std, fmt='o-g',
14             capsize=4, elinewidth=1, ecolor='red',
15             label='Искажённая функция')
16 plt.plot(X, np.sin(X), 'b', label = 'Истинная функция')
17 plt.legend()
18 plt.axes([.18, .18, .35, .25]) # врезка графика
19 plt.plot(X, Y) # результаты модельного эксперимента
20 plt.text(0, -1, 'исходные данные')
21 plt.show()

```

В данном примере мы эмулируем экспериментальные данные, вычисляя $\sin(x)$ с добавлением случайной погрешности (строка 9). Количество вычисляемых точек — 20, число повторений вычислений для каждой точки — 10. В результате у нас получается *numpy*-матрица результатов, для которой мы можем применять необходимые статистические функции. Для расчета среднеарифметического значения $\bar{Y}_i = \frac{1}{N} \sum_{k=1}^N Y_k(X_i)$ для N измерений в каждой i точке используем метод *mean* (строка 10). Для расчета стандартного отклонения в каждой точке $s_i = \sqrt{\frac{1}{N-1} \sum_{k=1}^N (Y_k(X_i) - \bar{Y}_i)^2}$ возьмем метод *std* (строка 11). Параметр *axis=1* в нашем случае указывает, что среднее берется по строкам. Параметр *ddof* определяет статистическую схему

расчета, его значение 1 соответствует несмещенной оценке дисперсии (т. е. в знаменателе стоит значение $N - 1$, а не N).

Таким образом, Y_mean и Y_std — это одномерные массивы, которые будут содержать средние значения и ошибки для всех точек. Осталось отобразить эти результаты на графике, что с успехом делает метод *errorbar* (строка 13), который выводит график средних значений (зеленая кривая) вместе с интервалами ошибок (красные линии) для каждой точки. Помимо параметра *yerr* можно использовать и *xerr* для отрисовки интервала ошибок по шкале абсцисс, но для нашей задачи это не требуется.

Для отображения графика исходной функции (синяя кривая) используем обычный метод *plot* (строка 16). Для того чтобы показать исходный набор значений массива Y , на основном графике мы использовали врезку, которая была реализована с помощью метода *axes* (строка 18). Параметрами метода выступает список из четырех значений: левый нижний угол врезки по осям X и Y , и размеры врезки dX , dY в условных единицах от 0 до 1 (единица — максимальная условная длина поля графика). Установив новые координаты, с помощью метода *plot* выводим всю матрицу Y на поле врезанного графика (строка 19). Метод *text* выводит надпись, позицию левого нижнего угла которой следует задать в списке аргументов метода в текущей системе координат (строка 20). Для того чтобы столь сложный составной график не проиграл в разрешении, до начала всех отрисовок задаем необходимый размер поля вывода (строка 12).

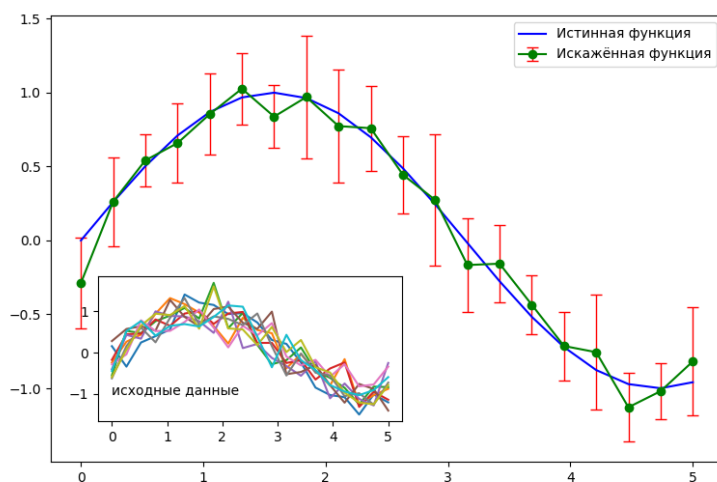


Рис. 9. Изображение интервалов ошибок, врезка графика

При статистическом анализе результатов реальных экспериментов более корректным является расчет стандартной ошибки среднего (средней квадратичной погрешности) [5]:

$$\bar{s}_i = \sqrt{\frac{1}{N(N-1)} \sum_{k=1}^N (Y_k(X_i) - \bar{Y}_i)^2}$$
. В этом случае достаточно умножить результат в строке 11 на $\frac{1}{\sqrt{N}}$, либо написать собственную функцию, вычисляющую s_i и оценку математического ожидания по заданному набору значений, как, например, это сделано в следующей программе:

```

1 import numpy as np
2
3 def mnk(v):
4     N = len(v)
5     M = np.sum(v)/N

```



```

6     D = 0
7     for R in v:
8         D += (R-M)**2
9     D = np.sqrt(D/N/(N-1))
10    return M, D
11
12    A = [1., 2.5, 0.73, 4., 3.]
13    res = mnk(A)
14    print('среднее арифм. = %f, ср.кв.откл. = %f' % res)

```

И, наконец, закончим этот достаточно беглый обзор возможностей библиотеки *matplotlib* кодом, рисующим трехмерные графики функций. Из всего многообразия вариантов 3D-графиков остановимся на изображении поверхности функции двух переменных. Пример самой простой реализации:

```

1  import matplotlib.pyplot as plt
2  from mpl_toolkits.mplot3d import Axes3D
3  import numpy as np
4
5  fig = plt.figure()
6  ax = fig.add_subplot(111, projection='3d')
7
8  # Подготовка данных
9  X = np.arange(-5, 5, 0.25)
10 Y = np.array(X)
11 X, Y = np.meshgrid(X, Y)
12 Z = X**2 + Y**2
13
14 # Построение графика
15 ax.plot_surface(X, Y, Z)
16 plt.show()

```

Параметры контейнера для вывода графика задаются в строке 6. Аналогично MATLAB, для расчета значений функции $Z(X, Y)$ для всего набора значений аргументов (строки 9 и 10) предварительно надо расширить векторы X и Y в матрицы, чем и занимается метод *meshgrid()*. Для отрисовки графика будем использовать метод *plot_surface* с параметрами, заданными по умолчанию. Результат приведен на рисунке 10.

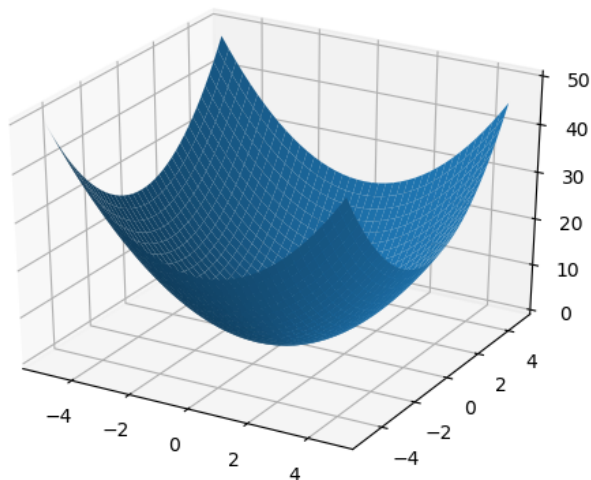


Рис. 10. График функции двух переменных

Окно с графиком остается интерактивным — можно с помощью мышки повернуть систему координат на любой угол. В качестве экспериментов с дизайном поверхности можно попробовать заменить строку 15 на следующую (выбрана цветовая палитра):

```

ax.plot_surface(X, Y, Z, rstride=1, cstride=1,
               cmap='viridis', edgecolor='none')

```

либо на такую (поверхность как трехмерный контур):

```
ax.contour3D(x, y, z, 50, cmap='binary')
```

Вывод позиций некоторого множества точек (рис. 11), имеющих координаты X , Y , Z , можно реализовать с помощью метода *scatter*:

```
1 import matplotlib.pyplot as plt
2 from mpl_toolkits.mplot3d import Axes3D
3 import numpy as np
4
5 def gendata(n, vmin, vmax):
6     return vmin + (vmax - vmin) * np.random.rand(n)
7
8 fig = plt.figure()
9 ax = fig.add_subplot(111, projection='3d')
10 # Построение графика
11 for m in range(5):
12     x = gendata(50, -5, 5)
13     y = gendata(50, -5, 5)
14     z = x**2 + y**2
15     ax.scatter(x, y, z, marker='o')
16
17 plt.show()
```

В этом коде реализована функция *gendata*, которая возвращает случайные вещественные числа в заданном диапазоне значений. Она, в свою очередь, использует метод *numpy.random.rand* (строка 6), который генерирует вектор заданного размера из псевдослучайных чисел в диапазоне от 0 до 1.

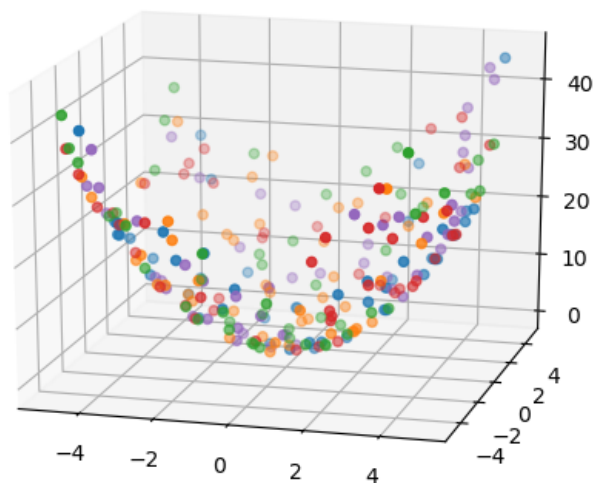


Рис. 11. График в виде набора точек

Глава 2. Получение и визуализация данных эксперимента

2.1 Подключение устройства сбора данных NI USB 6351

Рассмотрим пример, связанный с автоматизацией регистрации температуры в каком-либо физическом эксперименте (или, как вариант, за вашим окном). Для этого нам потребуется любая термопара, подключенная к аналого-цифровому преобразователю (АЦП). Такое электронное устройство мы можем легко найти в стандартной плате сбора данных. Компания National Instruments специализируется на выпуске соответствующих устройств, которые уже давно стали фактическим стандартом для экспериментальных исследований. Однако для работы с этими устройствами обычно используется система LabView, которую мы и собираемся заменить программой на Python.

Мы использовали устройство DAQ (Data acquisition, система сбора данных) NI USB 6351. Это устройство, как следует из его обозначения, использует USB-интерфейс для взаимодействия с компьютером. Для сопряжения с внешними устройствами (датчиками, периферийным оборудованием) имеется: 16 аналоговых входов с временным разрешением 10 нс и входным напряжением до ± 10 В (АЦП), 2 аналоговых выхода (16-разрядные цифро-аналоговые преобразователи), 24 цифровых ввода-вывода, четыре 32-битных счетчика-таймера до 25 МГц, генератор базовых частот. Из этого изобилия возможностей мы будем использовать только один аналоговый двуполярный вход AI0 (клеммы + и -), к которым и подключим термопару.

Первый этап состоит в установке актуальных драйверов поддержки оборудования National Instruments. Их следует скачать и установить с сайта [6]. Драйверы относятся к свободно распространяемому программному обеспечению, необходима только регистрация на сайте. Возможно, вместе с драйверами вы разрешите инсталлятору установить ряд дополнительных программ от National Instruments, которые обычно весьма полезны на стадии отладки приложений, работающих с DAQ.

На втором этапе проверяем подключение и работу устройства с использованием загруженной вместе с драйверами программы NI Device Monitor (рис. 12). Устройство сбора данных в нашей системе установилось с именем *Dev1*, для работы нам нужен аналоговый вход *ai0*.

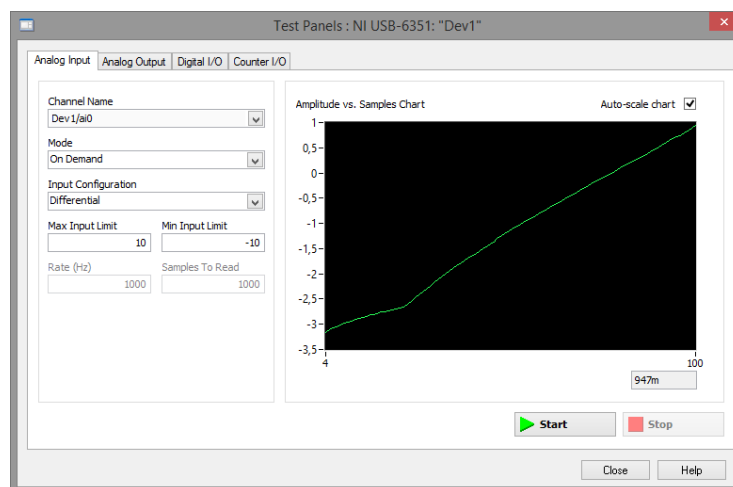


Рис. 12. Тестирование устройства сбора данных

Далее устанавливаем пакет *nidaqmx* из командной строки Windows в режиме администратора:

```
> python -m pip install nidaqmx
```

Эта библиотека постоянно модифицируется и ее функциональные возможности все время увеличиваются. Поэтому при использовании рассмотренных ниже примеров следует обязательно руководствоваться актуальным описанием для данной библиотеки. Далее, если не оговорено особо, мы использовали оболочку IDLE Python в режиме программирования (программа сохранялась в файл и запускалась на выполнение). Полное описание возможностей *nidaqmx* можно

найти в [7]. Итак, напишем простую проверочную программу на Python для тестирования подключения к плате сбора данных:

```
import nidaqmx

with nidaqmx.Task() as task:
    task.ai_channels.add_ai_voltage_chan("Dev1/ai0")
    data = task.read(number_of_samples_per_channel = 10)
print(data)
```

Работа с DAQ-устройствами происходит в парадигме задач. Таким образом, сначала устройство программируется на определенный вид деятельности, а затем мы начинаем получать данные с этого устройства. В нашем случае в строке 4 мы программируем аналоговый вход 0 на получения значений напряжения (это выглядит, на первый взгляд, тривиально, но в коллекции методов у нас есть возможность запрограммировать конкретный вход на измерение иных физических величин). Выполняем задачу в строке 5, измеряя 10 раз значение напряжения. Конструкция *with ... as* (так называемый менеджер контекста) делает более удобной работу с объектом *Task* библиотеки *nidaqmx* и корректно формирует завершение задачи. Все, что связано с DAQ-задачей, пишем внутри этой конструкции.

Результат выполнения программы будет выведен в виде списка:

```
0.07741842546889639,      0.07774029508782221,      0.07774029508782221,
0.07774029508782221,      0.07774029508782221,      0.07741842546889639,
0.07774029508782221,      0.07774029508782221,      0.07806216470670671,
0.07774029508782221]
```

Теперь подключим термопару (в нашем случае это была термопара *железо-константан*, т. е. типа *J*) ко входу *ai0* и модифицируем программу следующим образом:

```
1  import nidaqmx
2  import matplotlib.pyplot as plt
3
4  plt.ion()
5  i = 0
6
7  with nidaqmx.Task() as task:
8      task.ai_channels.add_ai_thrmcp1_chan("Dev1/ai0",
9          units = nidaqmx.constants.TemperatureUnits.DEG_C,
10         thermocouple_type = nidaqmx.constants.ThermocoupleType.J)
11     while i < 100:
12         temp = task.read(number_of_samples_per_channel = 1)
13         plt.scatter(i, temp, marker='*', s=4, c='g')
14         plt.pause(0.5) # задержка, в секундах
15         i = i + 1
16         print(temp)
```

Мы изменили вид данных, возвращаемых со входа *ai0*, разрешив использовать встроенную в библиотеку калибровку температуры для данного типа термопары (строка 10). В зависимости от вида термопары можно выбрать несколько уже готовых калибровок. В строке 12 указывается, что для каждого обращения к плате сбора данных мы будем брать только одно значение. Однако при этом создается цикл (со строки 11) для выборки 100 значений с задержкой между измерениями 0.5 с (строка 14). Хотелось бы, чтобы измеренное значение появлялось на графике в тот момент, когда выполнено очередное измерение. Для этого в строке 4 переводим окно визуализации графика в интерактивный режим. Теперь поле графика появится сразу после первого выполнения метода *plt.pause*. Для отображения точек был использован метод *scatter* из библиотеки *matplotlib* (строка 13), который выводит точку с указанными значениями координат (*i*, *temp*) на поле графика. Точки линиями не соединяются (рис. 13).

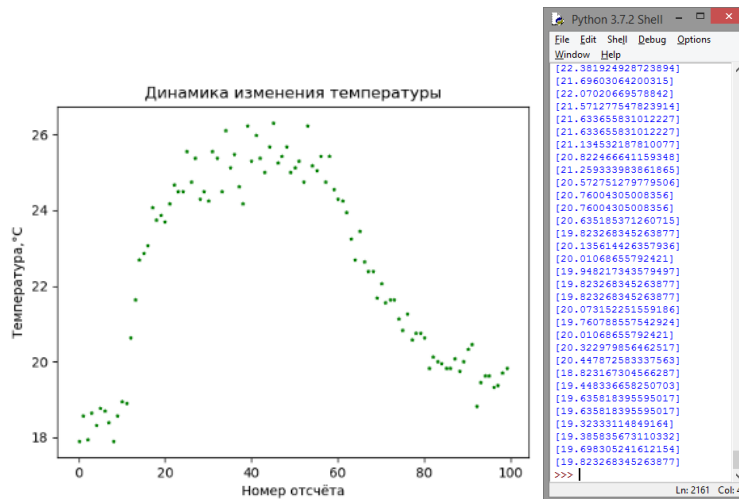


Рис. 13. Вывод значений температуры на график и в окно терминала

Интерактивный режим включается с помощью метода `ion()` пакета `matplotlib`, а выключается методом `ioff()`. При этом в интерактивном режиме метод `show()` не используется, вместо него можно использовать метод `draw()`, который отображает график и не задерживает выполнение программы. В нашем варианте функцию обновления графика с успехом выполняет метод `pause()`.

Если мы хотим привязаться не к номеру измерения, как в рассмотренном примере, а к реальным меткам времени, воспользуемся возможностями модуля `time` (импорт в строке 3):

```

1  import nidaqmx
2  import matplotlib.pyplot as plt
3  import time
4
5  plt.ion()
6  plt.ylabel('Температура, \N{DEGREE SIGN}C')
7  plt.xlabel('Время, с')
8  plt.title('динамика изменения температуры')
9  start = time.time() # установим начальную метку времени
10
11 with nidaqmx.Task() as task:
12     task.ai_channels.add_ai_thrmcp1_chan("Dev1/ai0",
13     units = nidaqmx.constants.TemperatureUnits.DEG_C,
14     thermocouple_type=nidaqmx.constants.ThermocoupleType.J)
15     try:
16         while True:
17             temp=task.read(number_of_samples_per_channel=1)
18             plt.scatter(time.time()-start, temp,
19                 marker='*', s=4, c='g')
20             # смещение во времени и будет координатой X
21             plt.pause(0.5)
22             print(temp)
23     except KeyboardInterrupt:
24         print('Выход по Ctrl-C')

```

Вычисление времени, прошедшего с начала измерения, осуществляется в строке 18. В этом варианте программы мы использовали бесконечный цикл для чтения информации с аналогового входа платы сбора данных, однако предусмотрели выход из программы по нажатию комбинации клавиш `Ctrl-C`. Для этого использовалась стандартная конструкция обработчика ошибок Python `try... except`.

Использование метода `scatter` позволяет достаточно удобно реализовать динамическое отображение измеряемых значений. В принципе, ничто не мешает заменить этот метод классическим `plot`, однако, следует учесть, что линии, соединяющие точки, при этом все равно не будут отображаться.

Программа, которая строит график в реальном времени с возможностью выбора типа соединительной линии и маркера экспериментальных точек, будет несколько сложнее (представлено Anthony Zhang в [8]). По указанной ссылке можно взять текст программы и адаптировать его

под свою задачу. При этом, как и в классическом виртуальном инструменте LabView *Chart*, получаем бесконечную «ленту самописца», что очень удобно для визуализации данных с неизвестным временем завершения процесса. Попробуйте реализовать этот вариант программы самостоятельно. Полученный нами результат представлен на рисунке 14.

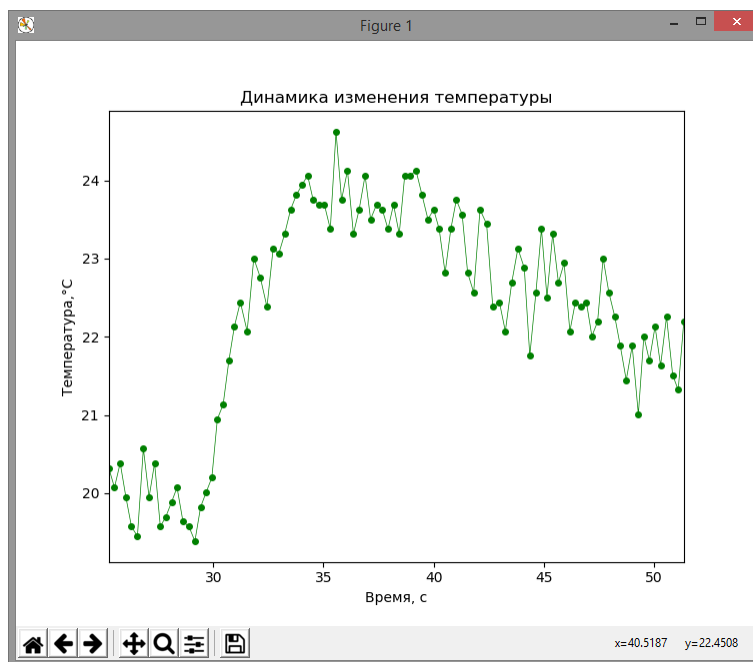


Рис. 14. Визуализация данных в стиле «ленты самописца»

Обратите внимание, что Python и библиотека *matplotlib* поддерживают отрисовку специальных символов в подписях осей координат и в любых других текстовых метках на поле графика (рис. 14, ось ординат, и строка 6 предыдущего листинга). Если потребуется отобразить, например, греческие буквы, можно использовать следующую конструкцию:

```
plt.title(r'\Sigma \alpha^2 \beta \omega_i')
```

Символ **r** (отключение экранировки) перед текстовой строкой обязателен. Можно, например, изобразить на графике символы $\sum_{j=0}^{\infty} \Delta_j$ с помощью следующей строки:

```
r'\sum_{j=0}^{\infty} \Delta_j'
```

2.2 Сохранение данных в файл

Теперь попробуем реализовать запись получаемых данных в файл. С точки зрения оптимального быстродействия для типичного эксперимента (если он не длится несколько суток и объём получаемых данных относительно невелик) логично сначала сохранять данные в список, который по окончанию (или прерыванию) процесса измерения может быть записан в файл на диске.

```
1 import nidagmx
2 import matplotlib.pyplot as plt
3 import time
4
5 # инициализация
6 plt.ion()
7 start = time.time()
```

```

8 data = []
9 i = 1
10 print('Для завершения измерительного цикла нажмите Ctrl-C')
11
12 with nidaqmx.Task() as task:
13     task.ai_channels.add_ai_thrmcp1_chan("Dev1/ai0",
14     units=nidaqmx.constants.TemperatureUnits.DEG_C,
15     thermocouple_type=nidaqmx.constants.ThermocoupleType.J)
16     try:
17         # получаем данные и отображаем их на графике
18         while True:
19             temp=task.read(number_of_samples_per_channel=1)
20             dtime = time.time()-start
21             res = temp[0]
22             plt.scatter(dtime, res, marker='*', s=4, c='g')
23             data.append([i, dtime, res])
24             plt.pause(0.5)
25             i = i + 1
26     except KeyboardInterrupt:
27         print('Измерения остановлены.')
28     # записываем данные в файл
29     name = input('Введите имя файла для записи данных: ')
30     if len(name)>0:
31         try:
32             f = open(name+'.txt','w')
33             for row in data:
34                 for element in row:
35                     f.write('%s\t' % element)
36                     f.write('\n')
37             print('Файл записан.')
38         except IOError:
39             print('Ошибка в процессе записи в файл!')
40     finally:
41         f.close()

```

В строке 8 создаем пустой список, в строке 23 добавляем в него запись, содержащую порядковый номер, метку времени и измеренное значение. При записи в текстовый файл предполагаем, что каждая запись будет начинаться с новой строки, а числа будут разделены табуляцией. Для записи в файл мы использовали функции *open... close*. Аргумент 'w' предполагает, что если файл с заданным именем уже существует на диске, он будет перезаписан. Если вас не устывает такая ситуация, следует проверить существование файла с указанным именем перед записью, изменив строку программы для ввода имени файла следующим образом:

```

while True:
    name = input('Введите имя файла для записи данных: ')
    if os.path.isfile(name+'.txt'):
        print('Файл с именем '+ name +
              '.txt уже существует!')
    else:
        break

```

В предыдущем примере использован модуль *os*, который предоставляет набор функций для работы с операционной системой. Вот еще несколько команд модуля *os*, которые могут быть полезными при разработке консольных приложений в среде Python:

```

import os
os.getcwd() # показать текущий рабочий каталог
os.chdir("C:/WORK/") # поменять рабочий каталог
os.system('cd c:\WORK') # выполнить консольную команду

```

Условием, типичным для данных небольшого объема, является их запись в текстовом формате, что удобно для загрузки полученных данных в какие-либо иные программы обработки. В принципе, при записи файла можно было отказаться от использования циклов *for* по списку,

а просто написать `f.write('%s' % data)`. При этом структура данных в файле имела бы следующий вид:

```
[[1, 0.08000421524047852, 18.385337261796522], [2, 0.7800629138946533, 19.385835673110332], [3, 1.4001131057739258, 19.135796318960296]]
```

Поскольку при записи списка в файл мы использовали собственное форматирование данных, результат получился таким:

```
1 0.06000399589538574 17.696964979313158
2 0.8430070877075195 18.51044937957971
3 1.5037767887115479 20.135614426357936
```

Обработка ошибок при записи файла (конструкция `try... except... finally`) необходима, если ваш код будет использоваться уже как готовый программный продукт, а при интерактивном взаимодействии с Python можно обойтись минимумом команд.

2.3 Загрузка данных из файла и их визуализация

Итак, мы собрали данные с термопары и записали их в файл `alpha.txt`. Чтение файла и визуализация данных не представляет проблемы, но способ обработки списков имеет свои особенности. Приведем первый вариант программы:

```
1 import matplotlib.pyplot as plt
2 data = []
3
4 name = input('Введите имя файла данных (без расширения): ')
5 with open(name+'.txt', 'r') as f:
6     rows = f.read().splitlines()
7     for row in rows:
8         elems = row.split('\t')
9         data.append([int(elems[0]),
10                    float(elems[1]), float(elems[2])])
11 x = [d[1] for d in data]
12 y = [d[2] for d in data]
13 plt.plot(x, y, 'o', markersize=3)
14 plt.show()
```

Результат работы программы в виде графика приведен на рисунке 15.

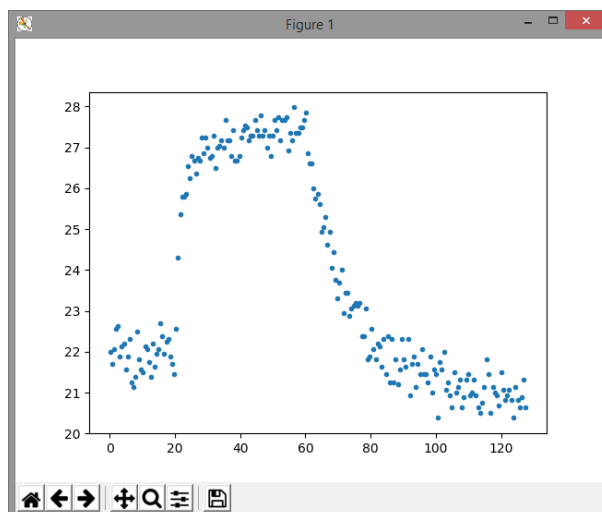


Рис. 15. Данные, загруженные из файла

Для упрощения работы с файлом здесь использована конструкция `with`, которая корректно обрабатывает открытие и закрытие файла. В этом примере мы не стали использовать библиотеку `numpy`. Читаем текстовый файл построчно, и эти строки записываются в список `rows`. Затем

каждая строка списка разделяется на текстовые элементы с помощью метода `split('\t')` (в качестве разделителя используется знак табуляции). И, наконец, текстовые строки требуется преобразовать в необходимый формат, поскольку Python требует строгой типизации. Это выполняется с помощью функций `int()` и `float()`. Список `data` после этого стал иметь ту же структуру, что и в исходной программе, которая формировала этот файл.

Однако, для того чтобы корректно отобразить данные на графике, нам нужно взять второй и третий столбцы списка. Не используя библиотеку `numpy`, вырезать столбец из `data` можно так, как показано в строках 11 и 12, для этого потребовалось создавать дополнительные списки `X` и `Y`.

А вот второй вариант программы чтения данных и отображения графика в случае, если мы будем использовать библиотеку `numpy`:

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 name = input('Введите имя файла (без расширения): ')
5 data = np.genfromtxt(name+'.txt', delimiter='\t',
6 dtype = [('i', '<i8'), ('time', '<f8'), ('temp', '<f8')])
7 plt.plot(data['time'], data['temp'], 'o', markersize=3)
8 plt.show()
```

Как видим, программа существенно упростилась, однако, следует обратить внимание на строки 5—6: импорт данных с помощью метода `genfromtxt` осуществляется в структурированный массив с именованными столбцами, при этом имена столбцов и тип данных указывается в списке значений параметра `dtype`. В случае, если тип не указан, он будет определяться по введенным значениям. Результат записывается в массив `data`, но структура этого массива отличается от списка предыдущей программы. В этой связи, для вывода графика в методе `plot` достаточно указать имя столбца в массиве `data` для выбора необходимых срезов матрицы (строка 7).

Третий вариант с библиотекой `pandas` (если вы предполагаете использовать специальные алгоритмы обработки больших данных [28, 29]) также может быть применен. В этом случае достаточно использовать для чтения файла метод `read_csv` так, как показано ниже. Вывод графика делается точно так же, как в предыдущем случае.

```
import pandas as pd
headers = ['i', 'time', 'temp', 'free']
data = pd.read_csv(name+'.txt', sep='\t', names=headers)
```

2.4 Работаем с осциллографом на Python

В настоящее время широко распространены цифровые запоминающие осциллографы, имеющие USB-интерфейс для связи с компьютером. Учитывая широкие возможности обработки и анализа электрических сигналов с помощью этого устройства, использование такого осциллографа может стать весьма полезным при проведении эксперимента. Обычно для таких осциллографов имеется собственная программа-драйвер, а также программа для тестирования возможностей удаленного управления. Представим, что нам требуется периодически получать данные с осциллографа в процессе проведения автоматизированного эксперимента. В таком случае от нас требуется реализовать прием и передачу команд и данных между управляющей программой на Python и осциллографом.

В качестве объекта интереса был взят двухканальный осциллограф Rigol DS1102E, для которого можно использовать программную прослойку от компании National Instruments, именуемую NI-VISA. Интерфейс VISA (Virtual Instrument Software Architecture) представляет собой хорошо зарекомендовавший себя стандарт для осуществления коммуникации с различными устройствами на различных платформах, в том числе подключаемых через USB-порт. В отличие от интерфейса DAQ, взаимодействие через интерфейс VISA очень напоминает работу через последовательный порт: передача команды от компьютера — прием ответа от

устройства. Поскольку выбранный нами осциллограф удовлетворяет протоколу USBTMC (Test and Measurement Class), он идентифицируется как «USB Test and measurement device (IVI)» и с ним можно работать через интерфейс VISA. Напомним, что вплоть до настоящего времени загрузка последних версий инсталляционного диска *NI-VISA*, равно как и пакета драйверов *NI-DAQ*, выполнялась с официального сайта компании National Instruments [5] свободно, требовалась лишь регистрация на сайте. При этом нет необходимости устанавливать LabView или иные платные программные продукты от National Instruments.

Для работы с интерфейсом NI-VISA из Python также нужно будет установить пакет *pyvisa* (команда `pip install pyvisa` в командной строке Windows в режиме администратора). Дополнительную информацию по пакету *pyvisa* можно посмотреть на сайте разработчиков [9].

После этого можно писать программу, взаимодействующую с осциллографом. В примере, показанном ниже, программа подстраивает осциллограф под характеристики измеряемого сигнала и получает содержимое экрана осциллографа.

```
1 import time, visa
2 import numpy as np
3 import matplotlib.pyplot as plt
4
5 rm = visa.ResourceManager()
6 dev = list(rm.list_resources())
7 scope = rm.open_resource(dev[0], timeout=20)
8 res = scope.query("*IDN?")
9 print('Установлена связь с устройством \n', res)
10 if res[:5]=='Rigol':
11     scope.write(":AUTO")
12     time.sleep(6)
13     dv = float(scope.query(":CHAN1:SCAL?"))
14     print('диапазон, В/дел. :', dv)
15     dt = float(scope.query(":TIM:SCAL?"))
16     print('Время, с/дел. :', dt)
17     data = scope.query_binary_values(
18         ":WAV:DATA? CHAN1", datatype='B')
19     n = len(data)
20     print('Взято данных: ', n)
21     x = np.array(range(n)) * dt/50.
22     data = (255 - np.array(data)) * dv/25.
23     plt.plot(x, data)
24     plt.show()
25 else:
26     print('некорректный ответ!')
```

В первую очередь нам необходимо выбрать нужное устройство с VISA-интерфейсом (строки 5—7), ибо таких устройств может быть подключено к компьютеру несколько. Структура *dev* содержит список всех подключенных VISA-совместимых устройств. В нашем случае это устройство было единственным, поэтому в строке 7 иницилируем работу с ним, выбирая первый элемент списка.

Основными методами *pyvisa* являются *read* (чтение информации из прибора), *write* (запись информации в прибор) и *query* (запись команды и сразу затем чтение возвращаемого результата). По умолчанию ввод-вывод осуществляется в символьном представлении. Первой командой, которую дадим осциллографу, является универсальный запрос **IDN?*, в ответ на который прибор должен «представиться». Для этого используем метод *query* (строка 8). Здесь мы проводим дополнительную проверку на то, корректно ли прибор отвечает.

В строке 11 мы пересылаем в осциллограф команду *:AUTO*, которая включает автоматическую настройку каналов осциллографа для наилучшего отображения регистрируемого сигнала. Выполнение команды занимает несколько секунд, поэтому в программу введена задержка по времени (строка 12). Команда не требует ответа прибора, поэтому был использован метод *write*. В строках 13—16 выводим установленный диапазон по значениям напряжения (для первого канала) и по горизонтальной развертке.

Строки 17—18 листинга наиболее интересны. Здесь выполняется команда *:WAV:DATA? CHAN1*, в ответ на которую осциллограф передает мгновенную копию сигнала на экране ос-

циллографа в виде последовательности байтов. Значение каждого байта есть инвертированное значение ординаты в диапазоне от 0 до 255, а порядковый номер байта — значение по абсциссам в диапазоне от 0 до 599. Для того чтобы перевести эти величины в реальные значения по напряжению и по времени, необходимо их умножить на соответствующий поправочный коэффициент (25 точек/дел. по Y — строка 22, и 50 точек/дел. по X — строка 21). Однако следует заметить, что в данном случае главной задачей была быстрая визуализация осциллограммы; если требуется получить от осциллографа *точное* значение величины сигнала в какой-либо точке, следует использовать другие команды.

Поскольку по умолчанию метод `query` работает с символьными данными, весьма ожидаемыми являются проблемы при передаче битового массива. Именно поэтому мы использовали метод `query_binary_values`. На самом деле можно было бы взять строку символов с помощью стандартного запроса `query`, а затем конвертировать их в битовый массив. Однако в программе, с учетом нюансов работы с символьными значениями, это бы выглядело более сложно:

```
scope._encoding = 'latin1'  
rawdata = scope.query(":WAV:DATA? CHAN1")[10:]  
data = bytearray(rawdata,'latin1')  
data = np.invert(np.array(data))
```

Первые десять символов, приходящих с осциллографа — идентификатор формата, поэтому в данном случае мы их пропускаем. Как можно заметить, в обоих случаях очень удобно использовать `numpy`-массивы. Массив по оси абсцисс достаточно просто формируется в строке 21: последовательность значений, соответствующих числу элементов в массиве `data`, каждый из которых умножен на поправочный коэффициент. В строке 22 массив `data` инвертируется (особенность осциллографа) и также нормируется на реальное значение по напряжению. Текстовый вывод программы приведен ниже, а результат представлен на рисунке 16.

```
>>>  
Установлена связь с устройством  
Rigol Technologies,DS1102E,DS1E...  
диапазон, В/дел. : 1.0  
Время, с/дел. : 0.0005  
Взято данных: 600
```

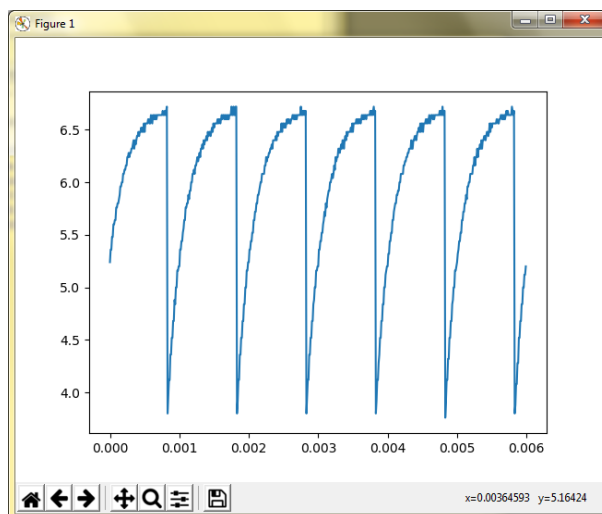


Рис. 16. Данные, полученные с осциллографа

2.5 Комментарии

В рассмотренных в этой главе примерах была показана возможность использования широко распространенных драйверов и библиотек от National Instruments. Однако более специфичное устройство может потребовать нестандартного подхода. В частности, нередки случаи, когда

устройство должно поддерживать широко распространенный протокол «COM via USB», но драйвер для него либо отсутствует, либо разработан только для конкретного приложения.

Об использовании модуля *pyserial* будет рассказано в главе 4. Однако при работе с USB-устройствами через Python без поддержки драйверов National Instruments может потребоваться пакет *pyusb*, устанавливаемый обычным образом. При запуске программы вы сразу же можете столкнуться с ошибкой «usb.core.NoBackendError: No backend available». Фактически Python требуется системная библиотека, с помощью которой и будет реализован доступ к USB-устройствам вашего компьютера. Для Microsoft Windows популярной библиотекой такого рода является *libusb-1.0.dll*, которую следует записать строго в каталог *Windows/System32*, даже если у вас 64-разрядная система [10].

Кроме того, устройство, к которому вы пытаетесь подключиться, не должно быть занято, т. е. желательно, чтобы у него вообще не было драйвера. Воспользуемся программой *ZADIG* [11], которая найдет на нашем компьютере USB-устройство, для которого не установлены драйверы, и предложит установить для него универсальный драйвер, совместимый с библиотекой *libusb*. После этого со стороны программы на Python можно пытаться взаимодействовать с внешним USB-устройством.

Глава 3. Обработка результатов эксперимента

3.1 Сглаживание экспериментальных данных

Попробуем сгладить набор экспериментальных данных, полученных в разделе 2.1. Совершенно очевидно, что любая процедура сглаживания будет подчинена какой-либо цели экспериментатора — например, показать общий тренд данных либо уменьшить шум. Воспользуемся стандартным методом библиотеки *numpy* с именем *convolve*. Вот полный текст программы:

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 data = np.genfromtxt('alpha.txt', delimiter='\t',
5 dtype=[('i', '<i8'), ('time', '<f8'), ('temp', '<f8')])
6 plt.plot(data['time'], data['temp'], 'o', markersize=3)
7
8 win = 6
9 filt = np.ones(win)/win
10 mov = win//2
11 res = np.convolve(data['temp'], filt, mode='same')
12 plt.plot(data['time'][mov:-mov], res[mov:-mov], 'r')
13
14 plt.show()
```

Строки 4—5 ничем не отличаются от рассмотренного в предыдущей главе формирования *numpy*-массива из форматированного текстового файла. В строке 6 исходный набор экспериментальных точек отображается в виде графика.

Метод *convolve* (строка 11) осуществляет линейную свертку двух векторов данных. Первый параметр — массив (вектор) исходных данных, второй параметр — сглаживающий (пробный) вектор, который формируется в строке 9. Его размер соответствует количеству точек, которые будут использованы для вычисления среднего для каждой точки из первого вектора (за исключением особых случаев в начале и конце массива данных). Третий параметр — режим свертки (из возможных ‘full’, ‘same’ и ‘valid’ описанную логику сглаживания дает только параметр ‘same’).

Параметр *win* задает размер второго вектора. Чем он больше — тем по большему числу экспериментальных точек идет усреднение, тем лучше сглаживание. Метод *ones* задает единичный вектор, но поскольку мы вычисляем среднее с равным весом по всем точкам, в строке 9 значение *filt* для *win* = 6 будет равно:

```
array([0.1667, 0.1667, 0.1667, 0.1667, 0.1667, 0.1667])
```

Пробный вектор можно представить движущимся вдоль массива исходных данных для расчета среднего для каждого элемента, но для значений в начале массива, индекс которых меньше *win/2*, и для значений в конце массива, индекс которых больше *win/2*, набора значений будет не хватать, и среднее будет рассчитываться неверно. Поэтому в строке 10 мы делим нацело значение *win* пополам, а в строке 12 строим график для среза нашего массива, в котором убраны начальные и конечные значения с соответствующими индексами.

Результат применения сглаживания приведен на рисунке 17.

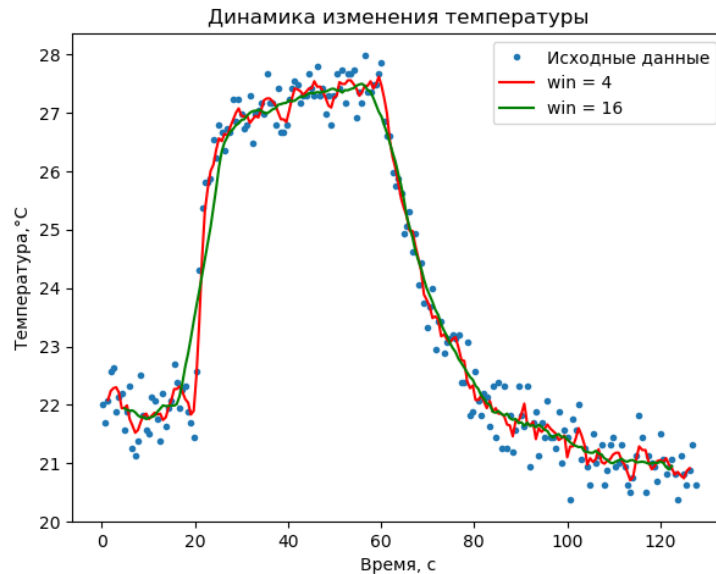


Рис. 17. Результат сглаживания экспериментальных данных

Как можно заметить, для изображения нескольких кривых с разными значениями параметра фильтра нам потребовалось изменить текст описанной выше программы. А именно, была написана функция, которая берет значение *win* и массив исходных данных, и выдает обрезанные массивы со значениями абсцисс и с результатом сглаживания:

```
def smooth(x_data, y_data, win=4):
    filt = np.ones(win)/win
    mov = win//2
    return x_data[mov:-mov], \
           np.convolve(y_data, filt, mode='same')[mov:-mov]

X, Y = smooth(data['time'], data['temp'])
plt.plot(X, Y, 'r', label = "win = 4")
X, Y = smooth(data['time'], data['temp'], 16)
plt.plot(X, Y, 'g', label = "win = 16")
```

3.2 Разбиение спектральных кривых на гауссианы

Нередко при обработке экспериментальных данных ставится задача аппроксимации исходного набора значений некоей ожидаемой функциональной зависимостью (чаще всего прямой, экспонентой или логарифмом). При анализе спектров излучения или поглощения света для аппроксимации отдельного пика на спектре можно выбрать функцию Гаусса (статистическую функцию плотности нормального распределения) вида $y = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\bar{x})^2}{2\sigma^2}}$, где \bar{x} — математическое ожидание, σ — дисперсия. Покажем на примере этой функции, как можно с помощью Python решить задачу аппроксимации.

Для начала разнообразим ситуацию с первичными данными: предположим, что результаты спектральных измерений записаны в электронную таблицу Microsoft Excel. В таком случае нам нужно найти метод для загрузки данных из *xls*-файла.

Для этого воспользуемся возможностями библиотеки *openpyxl*, которую можно установить стандартным образом (`pip install openpyxl`).

В представленном на рисунке 18 файле *spectra.xlsx*, открытом в Microsoft Excel, видны две вкладки. На вкладке *Лист1* имеется несколько экспериментальных наборов данных для разных материалов. Нам потребуются данные из столбцов **Ж** (10) и **К** (11).

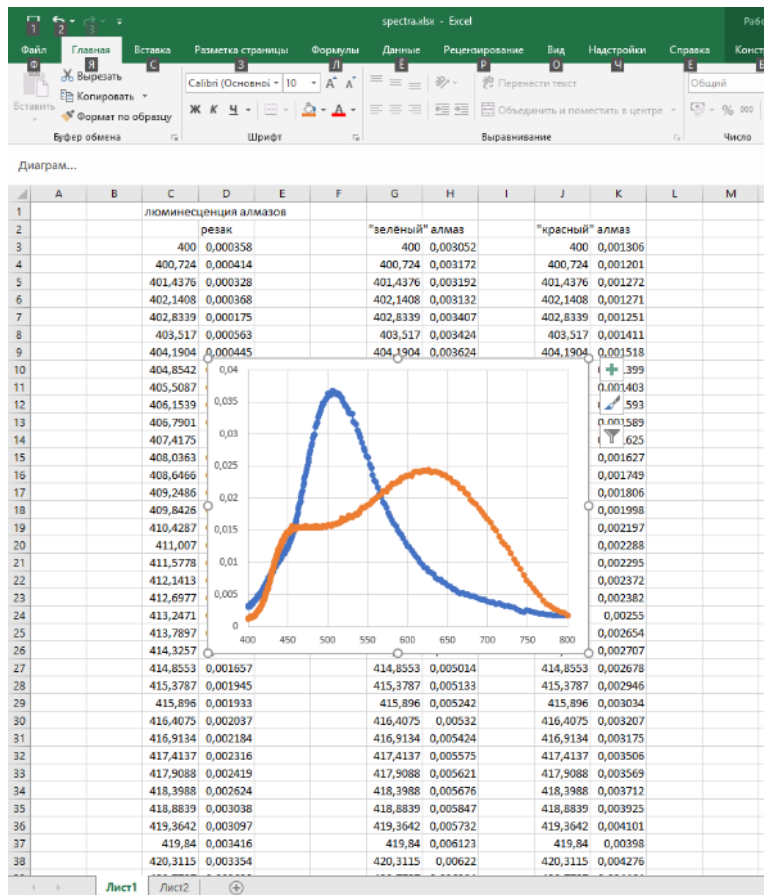


Рис. 18. Исходные данные, открытые в Microsoft Excel

Ниже представлена программа на Python, загружающая данные из Excel-файла. Результат работы программы показан на рисунке 19.

```

1 from openpyxl import load_workbook
2 import matplotlib.pyplot as plt
3
4 wb = load_workbook('spectra.xlsx')
5 print(wb.sheetnames)
6 sheet = wb["Лист1"]
7 print('выбрали рабочий лист: ', sheet.title)
8 # нам нужны данные в 10 и 11 столбцах (значения X и Y)
9 X = [1239.6/v[0].value for v in sheet.iter_rows(3, 270, 10, 10)]
10 Y = [v[0].value for v in sheet.iter_rows(3, 270, 11, 11)]
11 plt.plot(X, Y, 'r-')
12 plt.ylabel('интенсивность, отн. ед.')
13 plt.xlabel('Энергия, эВ')
14 plt.show()

```

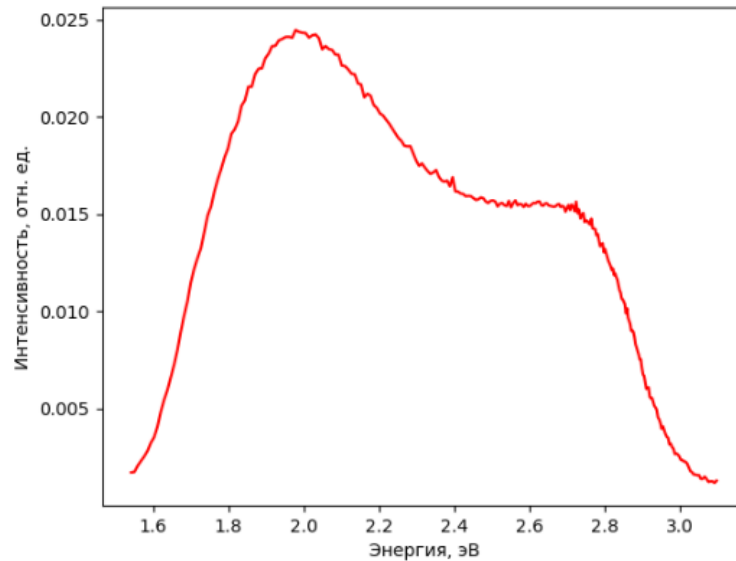


Рис. 19. Загруженные из файла и преобразованные экспериментальные данные

В строке 5 выводится на экран список всех листов книги Excel. Нужный лист выбираем по его имени в строке 6. Из десятого столбца таблицы считываем значения длин волн в нанометрах (строки таблицы с 3 по 270) и записываем в список X , предварительно переведя их в электронвольты (строка 9). Значения интенсивности считываем из столбца 11 и записываем в список Y (строка 10). В строках 11—14 строим график спектра.

Рассмотрим теперь способ аппроксимации полученных данных суммой функций Гаусса. В случае с представленным выше спектром люминесценции достаточно сложной формы можно предположить, что свечение обусловлено дефектами разной природы, каждый из которых дает пик, близкий по форме к статистическому распределению Гаусса. В данном случае как минимум два гауссиана составляют в сумме этот спектр. Напишем программу, которая решит поставленную задачу:

```

1  from openpyxl import load_workbook
2  import matplotlib.pyplot as plt
3  from scipy.optimize import curve_fit
4  from scipy import stats
5  import numpy as np
6
7  def Gauss(x, a, x0, s):
8      return a*np.exp(-((x-x0)/s)**2)
9
10 def DoubleGauss(x, a1, x01, s1, a2, x02, s2):
11     return Gauss(x, a1, x01, s1) + Gauss(x, a2, x02, s2)
12
13 # вектор начальных приближений
14 ip0= [.025, 1.8, 1., .015, 2.8, 1.]
15
16 wb = load_workbook('spectra.xlsx')
17 print(wb.sheetnames)
18 sheet = wb["лист1"]
19 print("Выбрали рабочий лист: ", sheet.title)
20 X = [1239.6/v[0].value for v in sheet.iter_rows(3, 270, 10, 10)]
21 Y = [v[0].value for v in sheet.iter_rows(3, 270, 11, 11)]
22
23 # расчёт
24 p, cov = curve_fit(DoubleGauss, X, Y, p0=ip0,
25                    bounds=(0, [1., 3., 10., 1., 3., 10.]))
26 print("Параметры гауссианов: ",p)
27 YY = DoubleGauss(X, *p)
28
29 # оценка погрешности аппроксимации и достоверности модели
30 print("Стандартное отклонение: ", np.std(Y-YY))

```



```

31 slope, ic, r_value, p_value, std_err = stats.linregress(Y, YY)
32 print("квадрат коэффициента корреляции: ", r_value**2)
33
34 plt.plot(X, Y, 'r-')
35 plt.ylabel('интенсивность, отн. ед.')
36 plt.xlabel('Энергия, эВ')
37 plt.plot(X, Gauss(X,p[0],p[1],p[2]),'b', linewidth=0.5)
38 plt.plot(X, Gauss(X,p[3],p[4],p[5]),'b', linewidth=0.5)
39 plt.plot(X, YY, 'g')
40 plt.savefig('result.png')
41 plt.show()

```

В строках 7—8 определим функцию, описывающую статистическое распределение Гаусса. Форма записи выражения в данном случае полностью совпадает с описанием функции Гаусса в приложении *Curve Fitting Toolbox* программного пакета MATLAB — для того, чтобы при необходимости проверить результаты аппроксимации, полученные в программе. Параметр x — значение по абсциссе, a — нормировка по высоте, x_0 — положение максимума, s — величина, пропорциональная полуширине.

Поскольку предполагается, что будут применяться групповые операции при работе с векторами (в стиле MATLAB), будем вычислять экспоненту в виде *np.exp*, и далее везде при работе с векторами будем также использовать библиотеку *numpy*. Функция *DoubleGauss* (строки 10—11) является суммой двух гауссианов и, соответственно, имеет шесть параметров, которые могут варьироваться для достижения наилучшей корреляции с экспериментальными данными.

Строки 16—21 полностью аналогичны предыдущему листингу и реализуют загрузку исходного набора данных. Собственно аппроксимация осуществляется в строке 24 с помощью метода *curve_fit* из библиотеки *scipy*. Эта библиотека в представленных примерах встретила в первый раз, но на самом деле она содержит весьма обширный набор методов анализа данных и носит название Scientific Computing Tools for Python [12]. Библиотека устанавливается в обычном порядке. Параметрами *curve_fit* являются аппроксимирующая функция, набор аппроксимируемых точек (списки X и Y), вектор начальных приближений p_0 , значение которого задано в строке 14, и кортеж ограничений по минимальным и максимальным значениям для всех варьируемых параметров. Следует заметить, что оба последних параметра очень важны и требуют понимания физической сущности описываемых явлений. В нашем случае, очевидно, значения всех варьируемых параметров не могут быть меньше нуля, поэтому первым элементом кортежа указан нуль. Второй элемент кортежа представляет собой список значений для каждого из варьируемых параметров.

Поскольку метод берет на себя все расчеты, в строке 26 мы уже готовы использовать полученные значения параметров гауссианов в списке p , которые дают наиболее хорошее согласие с экспериментом. Далее осталось только построить графики и оценить погрешность аппроксимации.

Результат выполнения программы представлен на рисунке 20.

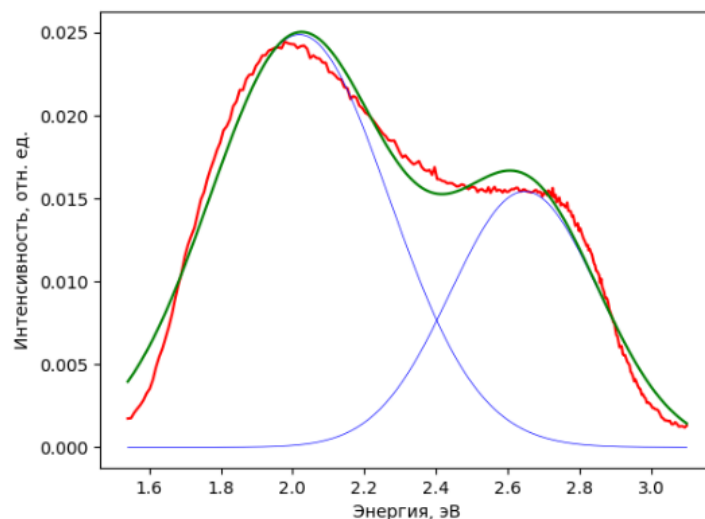


Рис. 20. Результат разбиения спектра на сумму двух гауссианов

Вывод в окно терминала выглядит следующим образом:

```
>>>
['лист1', 'лист2']
Выбрали рабочий лист: лист1
Параметры гауссианов: [0.02488419 2.01897886 0.35283369 0.01543551 2.64783453
0.2929665]
Стандартное отклонение: 0.0010226334705150182
квадрат коэффициента корреляции: 0.9779722449624711
```

Для оценки достоверности подгонки в строке 30 используется уже встречавшийся ранее метод *std* библиотеки *numpy*, но в научной литературе в подобных случаях обычно приводят коэффициент корреляции R , для вычисления которого можно использовать метод *linregress* из библиотеки *scipy* (строка 31, переменная *r_value*). Особенность *linregress* в том, что этот метод также считает другие статистические характеристики, и в том числе среднюю квадратичную ошибку (переменная *std_err* в строке 31). Таким образом, мы определяем вероятность соответствия значений Y , вычисленным с помощью функции *DoubleGauss*, значениям Y (эксперимент).

Несмотря на достаточно хорошую корреляцию модели с экспериментальными данными, даже визуально можно заметить систематические отклонения формы исходного спектра от полученной расчетной зависимости. Попробуем увеличить количество гауссовых пиков. Теоретически каждый новый пик с тремя новыми варьируемыми параметрами будет уменьшать погрешность модели, но в нашем случае будет физически обоснованно остановиться на сумме трех гауссианов. Однако предполагая, что для какого-либо иного спектра может потребоваться разбиение на более чем три гауссовых пика, следующую программу напишем для аппроксимирующей функции, представляющей собой сумму любого количества гауссианов. Для этого изменим вид функции *Gauss* и введем новую универсальную функцию *nGauss*, которая сформирует сумму столько гауссианов, на сколько хватит количества входных параметров (по три на каждый гауссиан). Функция *plotAllGauss* будет выводить на график каждую из гауссовых функций.

```
1 from openpyxl import load_workbook
2 import matplotlib.pyplot as plt
3 from scipy.optimize import curve_fit
4 from scipy import stats
5 import numpy as np
6
7 def Gauss(x, a, x0, s):
8     return a*np.exp(-((x-x0)/s)**2)
9
10 def nGauss(x, *p):
11     n = len(p)//3
12     res = 0
13     for i in range(n):
```

```

14         res += Gauss(x, p[i*3], p[i*3+1], p[i*3+2])
15     return res
16
17 def plotAllGauss(x, p):
18     n = len(p)//3
19     for i in range(n):
20         plt.plot(x, Gauss(x,p[i*3],p[i*3+1],p[i*3+2]),
21                 'b', linewidth=0.5)
22
23     # начальные приближения для трёх гауссианов
24     ip0 = [.025, 1.8, 1., .01, 2.4, 1., .015, 2.8, 1.]
25     # верхняя граница
26     top_limits = [.1, 2.5, 10., .1, 3., 10., .1, 3., 10.]
27
28     wb = load_workbook('spectra.xlsx')
29     print(wb.sheetnames)
30     sheet = wb["лист1"]
31     print("Выбрали рабочий лист: ", sheet.title)
32     X = [1239.6/v[0].value for v in sheet.iter_rows(3, 270, 10, 10)]
33     Y = [v[0].value for v in sheet.iter_rows(3, 270, 11, 11)]
34     plt.plot(X, Y, 'r+', markersize=3)
35
36     # интерполяция экспериментальных данных
37     p, cov = curve_fit(nGauss, X, Y, p0=ip0,
38                       bounds=(0, top_limits))
39     print("Параметры гауссианов: ")
40     for pl in p:
41         print(pl)
42
43     # погрешность вычислений
44     print("Станд. отклонение: ", np.std(Y-nGauss(X, *p)))
45     slope, ic, r_value, p_value, std_err = \
46     stats.linregress(Y, nGauss(X, *p))
47
48     # вывод графиков
49     plt.ylabel('Интенсивность, отн. ед.')
50     plt.xlabel('Энергия, эВ')
51     plt.text(2.8,0.021,'R2=%.4f' % r_value**2)
52     plotAllGauss(x, p)
53     plt.plot(x, nGauss(x, *p), 'g')
54     plt.savefig('result.png')
55     plt.show()

```

Введенные изменения никак не отразились на строках 28—38, для изменения числа гауссианов достаточно поменять число значений для списков в строках 24 и 26. Вывод графика отличается от предыдущего случая строкой 51 (показываем значение рассчитанного квадрата коэффициента корреляции) и строкой 54 (сохранение графика в *png*-файл). Результат запуска программы представлен на рисунке 21.

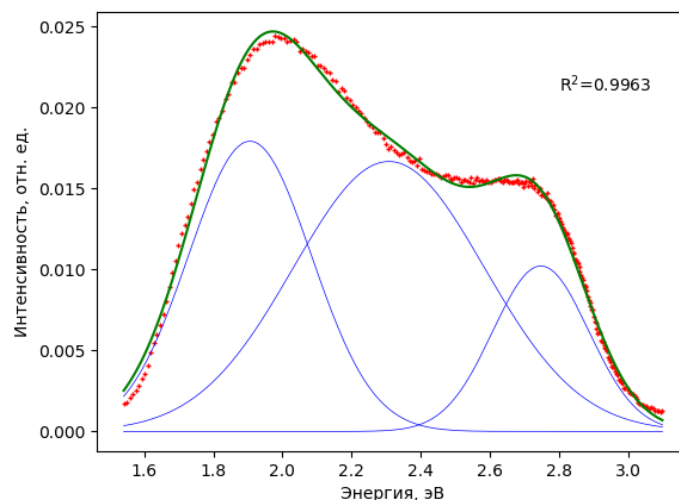


Рис. 21. Аппроксимация кривой суммой трех гауссианов. Красные — экспериментальные точки, зеленая — аппроксимирующая кривая, синие — отдельные гауссианы

Вывод в консоль Python выглядит следующим образом:

```
>>>
['лист1', 'лист2']
Выбрали рабочий лист: лист1
Параметры гауссианов:
0.01667446538287539
2.3079305189943593
0.39102048947240636
0.017927267046242815
1.906095575293327
0.25137630457432303
0.010218993081363254
2.7475262095865176
0.1965050875412267
Станд. отклонение: 0.0004111456843283068
```

Как можно убедиться, параметры всех трех пиков с физической точки зрения вполне реалистичны, корреляция модели лучше, а стандартная ошибка меньше, чем в первом случае.

3.3 Работа с файлами данных формата JDX

Рассмотрим возможность обрабатывать информацию, которая хранится в файлах формата JCAMP-DX. Это широко распространенный формат для хранения спектроскопических данных, поддерживаемый The joint Committee on Atomic and Molecular Physical data [13]. Вторая часть аббревиатуры расшифровывается как «Data Exchange format». Файлы подобного типа имеют расширение *.jdx*. Их обычно создает и обрабатывает программное обеспечение для ИК- или Рамановского спектрометра, но программы для многих других типов автоматизированных спектроскопических приборов также могут работать с этим форматом.

Файл имеет текстовый формат, однако его структура достаточно сложна. Достаточно сказать, что значения экспериментальных точек по оси абсцисс указаны с другим шагом, нежели значения этих же точек по оси ординат. Для того чтобы прочитать, например, ИК-спектр из файла, обычно приходится писать собственный алгоритм, пользуясь открытым описанием формата. Для языка Python ситуация проще, так как для работы с этим форматом уже написан соответствующий модуль *jcamp* [14], который имеет смысл научиться применять. Модуль устанавливается обычным образом — с использованием программы *pip* в режиме администратора:

```
>pip install jcamp
```

В нашем случае файл *test.JDX* содержит реальный спектр инфракрасного поглощения. Прочитаем данные и отобразим их на графике с помощью следующей программы, результат работы которой представлен на рисунке 22.

```
1 from jcamp import JCAMP_reader
2 import matplotlib.pyplot as plt
3 d = JCAMP_reader('test.JDX')
4 plt.plot(d['x'], d['y'], 'r', linewidth=0.5)
5 plt.gca().invert_xaxis()
6 plt.xlabel(d['xunits'])
7 plt.ylabel(d['yunits'])
8 plt.title('Спектр ИК-поглощения')
9 plt.show()
```

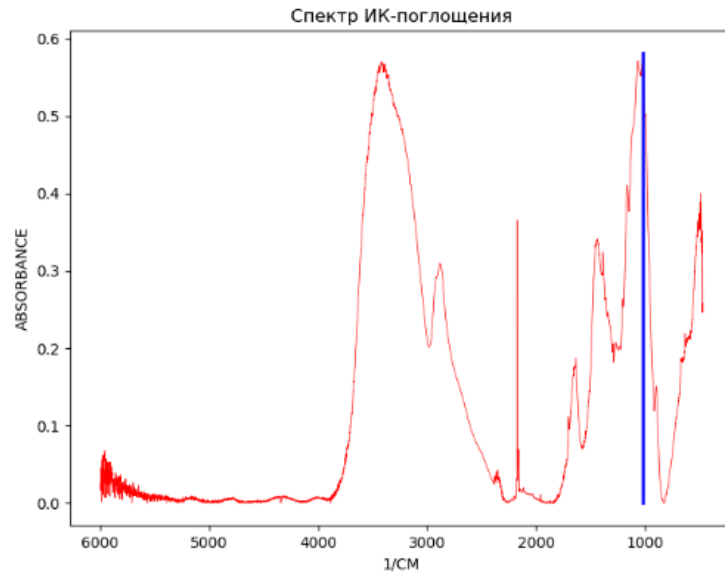


Рис. 22. Спектр ИК-поглощения, воспроизведенный из файла формата *jdx*

Фактически перемещение информации из файла в словарь Python произошло в строке 3. Структуру словаря *d* можно легко посмотреть с помощью *print(d)* или с использованием *iPython*, как это показано на рисунке 23.

Как следует из вида словаря, для отображения графика нам достаточно взять элементы 'x' и 'y', что и сделано в строке 4 листинга. Поскольку обозначения осей координат тоже взяты из файла и находятся в словаре, используем их в строках 6 и 7 для отображения на графике. И поскольку при публикации ИК-спектров ось абсцисс часто изображают в направлении от максимальных волновых чисел к минимальным, инвертируем ось с помощью метода *gca().invert_xaxis()* библиотеки *matplotlib* (строка 5). Если команды листинга выполнять интерактивно с помощью *iPython*, команда в строке 9 не нужна.

```

IPython: C:\Python\IPython
In [19]: d
Out[19]:
{'title': '23oct17_00035',
 'jcamp-dx': 4.24,
 'datatype': 'INFRARED SPECTRUM',
 '$note': '23.10.2017 ZnO + целлюлоза помол 5 ч (0.0041 г) в KBr 16:14:2 ^1 x60',
 '$par1': 1382,
 '$par2': 1692,
 '$sak': 60,
 '$apo': 1,
 '$ref': 0,
 '$datim': 2115681294,
 'date': '2017/10/23',
 'time': '16:14',
 '$background': 'c:\\WORK\\FTIR\\2017-10-23\\23oct17_00032.jdx',
 'xunits': '1/CM',
 'yunits': 'ABSORBANCE',
 'xfactor': 1,
 'yfactor': 2.70937e-10,
 'firstx': 469.7416004044,
 'lastx': 5999.5744445902,
 'firsty': 0.2586130069,
 'maxx': 5999.5744445902,
 'minx': 469.7416004044,
 'maxy': 0.581834,
 'miny': 0,
 'npoints': 5734,
 'xydata': '(X++(Y..Y))',
 'end': '',
 'x': array([ 469.742      ,  470.84428571,  471.94657143, ..., 5997.64466675,
            5998.60955567, 5999.57444459]),
 'y': array([0.25861254, 0.24622451, 0.27289315, ..., 0.01588574, 0.01770571,
            0.02530142]),
 'filename': 'test.JDX'}
In [20]:

```

Рис. 23. Структура словаря, содержащего данные из прочитанного файла

На рисунке 22 синяя линия показывает позицию абсолютного максимума на кривой. Попробуем, используя *iPython*, осуществить поиск и индикацию этого максимума следующим образом:

```
In [1]: Y = d['y']
In [2]: x = d['x']

In [3]: amax(Y) # максимальное значение элемента массива
Out[3]: 0.581832776867239

In [4]: argmax(Y) # индекс элемента с макс. значением
Out[4]: 572

In [5]: xmax = x[argmax(Y)] # абсцисса для макс. значения
In [6]: plt.plot((xmax, xmax), (0, amax(Y)), 'b',
linewidth=2) # изобразили линию, индицирующую максимум
```

Конечно, подобным образом мы можем найти только самый большой пик или, например, выброс в экспериментальных значениях. Но если нам необходимо узнать позиции всех пиков, то требуется либо написать собственный алгоритм, основываясь на априорной информации о том, что следует считать достоверным пиком поглощения, либо попытаться адаптировать уже готовые алгоритмы, которые также можно найти в библиотеке *scipy*. Например, метод *find_peaks_cwt* осуществляет поиск пиков в одномерном массиве *Y*, указанном в качестве первого параметра. Вторым параметром — это список возможных ширинок пиков (под шириной пика здесь понимается количество подряд идущих значений в массиве *Y*, которые вместе могут составлять некий пик). В нашем случае ширина «пика» выбрана больше 10 точек, но меньше 100. Результат, показанный на рисунке 24, уже может удовлетворить экспериментатора, при этом следует иметь в виду, что возможна настройка и других параметров указанного метода.

```
In [7]: from scipy.signal import find_peaks_cwt
In [8]: import numpy as np
In [9]: peaks = find_peaks_cwt(Y, np.arange(10,100))
In [10]: plt.plot(x[peaks],Y[peaks],'bo')
```

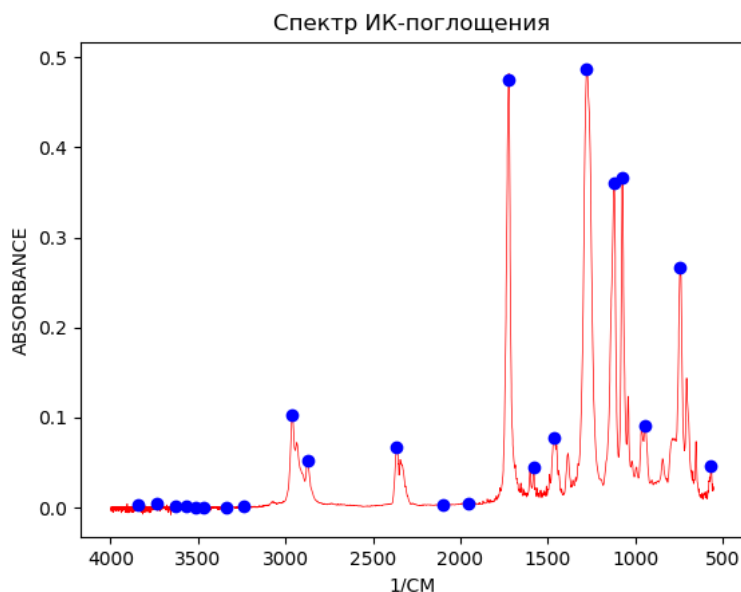


Рис. 24. Результат поиска пиков на графике спектра

Глава 4. Автоматизация спектроскопического эксперимента

4.1 Проектирование управляющей программы

Напомним, что Python — интерпретируемый язык, и создавать на нем критичные ко времени выполнения функции совершенно не рационально. Ведь при решении реальных задач для работы с периферийным оборудованием необходимо получать высокую производительность и функциональность, часто основанную на обработке прерываний. В этом плане одним из вариантов является соединение программы на Python с предварительно разработанными и оптимизированными библиотеками, написанными, например, на C / C++.

Более логичным вариантом разработки автоматизированной системы является вынесение наиболее критичных по быстродействию алгоритмов в периферийные микропроцессорные устройства либо микрокомпьютеры (SOC — System on a Chip), которые будут непосредственно взаимодействовать с датчиками и управляющими устройствами в режиме реального времени. В таком случае главная программа будет выполнять функции распорядителя задач и сбора данных. Такие функции очень хорошо подходят для реализации на языке Python.

Кроме того, далеко не всегда программа управления комплексом экспериментального оборудования, создаваемая в стенах научной лаборатории, должна быть выведена на уровень коммерческого программного продукта. Скорее наоборот, при тестировании различных измерительных методик приходится постоянно совершенствовать алгоритмы управления оборудованием, подключать новые устройства, переписывать ранее разработанные программы. Наличие простых способов отладки программ, удобное представление результатов и быстрота разработки и модификации программ — эти типичные для Python преимущества дают основание считать этот язык весьма привлекательным для решения задач автоматизации эксперимента.

Программа управления люминесцентным комплексом в минимальном варианте реализует функции управления положением дифракционной решетки монохроматора МДР-23 и получением данных с фотоэлектронного умножителя. Таким образом, задача программы состоит в измерении спектра свечения какого-либо источника излучения. Для этого необходимо последовательно выводить монохроматор на заданную длину волны и для каждой длины волны измерять соответствующую интенсивность излучения, прошедшего через оптическую схему монохроматора. В рамках данного учебного пособия мы намеренно не рассматриваем программирование GDI (Graphical Device Interface), поэтому покажем возможность построения полноценной управляющей программы, работающей в Windows-терминале (интерфейс командной строки). Поскольку программа достаточно большая, рассмотрим далее лишь ключевые моменты кода.

Пользовательский интерфейс представляет собой меню выбора функций. Представленный ниже фрагмент (файл *arti.py*) определяет каркас программы, которая может работать как при непосредственном взаимодействии с экспериментальной установкой, так и на любом другом компьютере, если в этом будет необходимость.

```
1 from cursesmenu import SelectionMenu
2 # ... фрагмент программы пропущен
3 menu = ['Инициализация установки',
4         'Ввод параметров измерения',
5         'Просмотр параметров измерения',
6         'Старт измерения',
7         'Просмотр результатов измерения и запись в файл',
8         'Просмотр файлов данных',
9         'Изменить путь к файлам данных',
10        'Консоль команд управления',
11        'Выход из программы']
12 meas_flag = False
13 while True:
14     item = SelectionMenu.get_selection(menu,
```

```

15     "программа управления спектральным комплексом",
16     exit_option = False)
17     if item == 0: # инициализация установки
18         pass # реализацию см. далее
19     if item == 1: # ввод параметров измерения
20         input('для выхода в основное меню нажмите <Enter>')
21 # ... фрагмент программы пропущен
22     if item == 7: # управление программой Arduino
23         console()
24     if item == 8: # выход из программы
25         break
26     print('программа работу завершила!')

```

В первую очередь следует отметить, что данная программа *не будет* работать в оболочке IDLE Python в связи с использованием библиотеки *cursesmenu* (`pip install curses-menu`), с помощью которой легко создавать интерактивные меню прямо в консоли командной строки (рис. 25). Для работы с данной библиотекой под Windows дополнительно потребуется установить пакет *windows-curses* (`pip install windows-curses`). Поэтому для редактирования текста программы в данном случае удобно использовать универсальные редакторы типа *Atom* либо *Microsoft Code*, а запуск программы следует осуществлять в режиме командной строки (`> python arti.py`). Рабочим каталогом по умолчанию предполагается каталог, где находится файл *arti.py*, относительно которого строятся все пути к файлам, читаемым или сохраняемым программой.

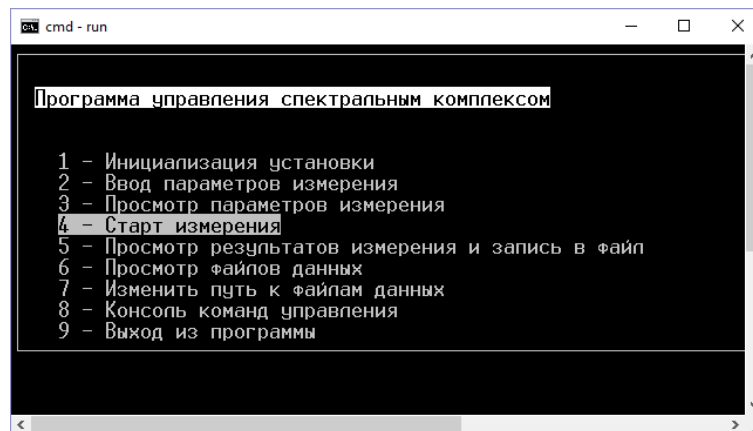


Рис. 25. Интерфейс управляющей программы

4.2 Взаимодействие с контроллером Arduino

Управление монохроматором МДР-23 осуществляет контроллер Arduino Uno, для которого написана программа, работающая с шаговым двигателем, который поворачивает дифракционную решетку. Программа выполняет команды, которые передаются ей через виртуальный «последовательный порт», связанный с платой Arduino (фактически Arduino подключена как USB-устройство и имеет собственный драйвер, реализующий интерфейс последовательного порта). Программирование снижавшего в настоящее время популярность контроллера Arduino не рассматривается в данном учебном пособии, мы можем лишь обратить внимание читателя на некоторые известные книги из обширной литературы по этой теме [15—17]. Ограничимся тем, что программа, разработанная для Arduino и загруженная в микроконтроллер (рис. 26), умеет распознавать и выполнять среди прочих следующие команды:

init L — инициализация установки, *L* — текущая длина волны монохроматора, которую следует указать (вещественное число, в нанометрах);

repl N — установленная оператором дифракционная решетка, *N* — наименование решетки (целое число);

wave L — сместить дифракционную решетку на относительную величину *L* (вещественное число, в нанометрах);

goto L — установить дифракционную решетку на указанную длину волны L (вещественное число, в нанометрах);

show — без параметров, выдает в порт текущее значение длины волны (в нанометрах) и наименование установленной дифракционной решетки.

Например, команда *wave 20* приведет к запуску шагового двигателя, который сместит (повернет) дифракционную решетку на 20 нм в сторону увеличения длин волн. Команда *wave -20* сделает то же, но в сторону уменьшения длин волн. Команда *goto 555.5* установит решетку в положение, соответствующее длине волны 555.5 нм. После выполнения каждой команды в последовательный порт возвращается символ «.» (точка). Если команда завершилась с ошибкой или была прервана, возвращаются символы «X.».

Таким образом, задачей управляющей программы на Python является только передача управляющих команд через виртуальный «последовательный порт» и прием результатов их выполнения. Рассмотрим вариант взаимодействия программы на Python и платы Arduino, который, очевидно, можно использовать не только для выбранной нами экспериментальной установки, но и для многих других вариантов автоматизации.

Инициализация обмена командами через последовательный порт реализована в пункте меню 1 «Инициализация установки».

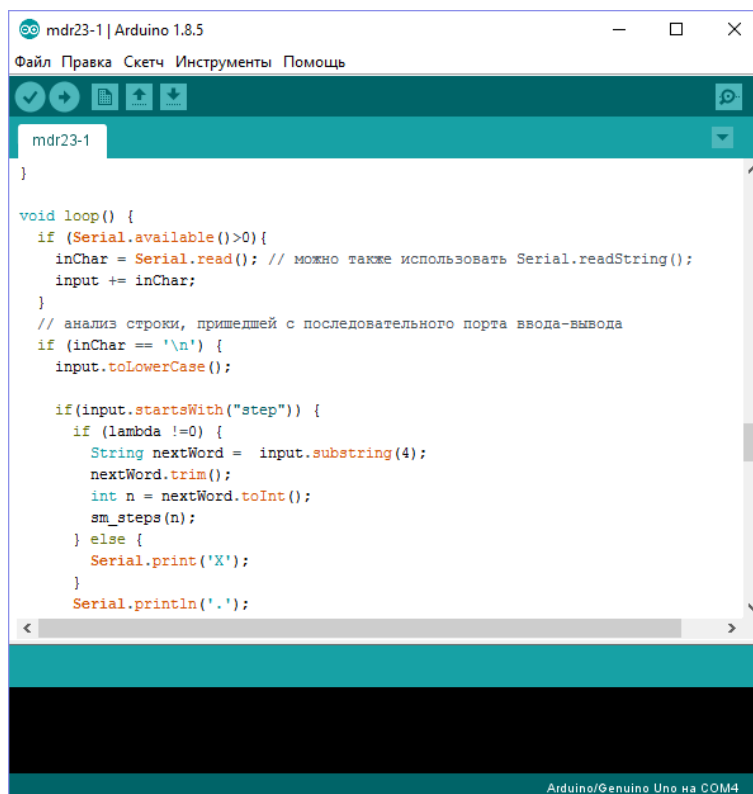


Рис. 26. Программирование платы Arduino

```
1 import serial, time
2 rate = 19200 # скорость последовательного порта
3 # фрагмент программы пропущен,
4 # далее вместо строки 18 предыдущего листинга
5 if meas_flag:
6     input('Установка уже инициализирована. Нажмите <Enter>')
7     continue
8 ser = serial.Serial("COM3", rate, timeout = 0)
9 time.sleep(2)
10 s = ser.readline().decode()
11 if s[:8] != 'MDR-2019':
12     print('Ответ: ', s)
13     print('Неверная версия программы контроллера Arduino!')
14 else:
15     replicas = ['1200', '600', '300']
```

```

16     item = SelectionMenu.get_selection(replics,
17         "Выберите установленную решётку, шт./мм:",
18         exit_option = False)
19     s = 'repl '+replics[item]+'\\n'
20     ser.write(s.encode())
21     time.sleep(0.1)
22     s = ser.readline().decode()
23     if s[0] != '.':
24         input('Ошибка команды repl!')
25         continue
26     s = input('Введите показания барабана монохроматора: ')
27     s = 'init '+ s +'\\n'
28     ser.write(s.encode())
29     time.sleep(0.1)
30     s = ser.readline().decode()
31     if s[0] != '.':
32         input('Ошибка команды init!')
33         continue
34     meas_flag = True
35     input('Инициализация успешно завершена! Нажмите <Enter>')

```

Работа с последовательным портом осуществляется с использованием модуля *pyserial* (строка 1). Устанавливайте пакет так: `pip install pyserial`. Параметры последовательного порта устанавливаются в строке 8. Поскольку программа в Arduino запрограммирована на скорость обмена 19200 бод, та же скорость выбрана и здесь. Задержка при инициализации порта в 2 секунды (строка 9) выбрана опытным путем при запуске программы на различных компьютерах. Не рекомендуется делать ее меньше 1 с. При инициализации программа Arduino выводит в порт начальное сообщение, которое читается в строке 10. Информация через последовательный порт передается в виде байтов, поэтому для ее преобразования в строку символов используем метод *decode()*.

Далее программа запрашивает ввод двух параметров, необходимых для инициализации системы, и пишет в последовательный порт команды *repl* и *init* (строки 20 и 28). В этом случае используется обратное преобразование из строки в байты и устанавливается необходимая временная задержка после каждой команды. Заметим, что мы не стали программировать последовательный порт на автоматическое добавление символа «возврата каретки» и «перевода строки» — вместо этого к каждой выводимой строке добавляем символ «\\n» разрыва строки. После ввода каждой команды переключаемся в режим чтения порта и ожидаем окончания выполнения команды (прихода со стороны Arduino символа «.»). Если все команды выполнены без ошибок, переменная *meas_flag* принимает значение *True*, и становится доступным режим проведения измерений (пункт меню 4 на рис. 25).

Для отладки и настройки работы программы нижнего уровня (записанной в контроллер Arduino) реализован пункт меню «8. Консоль команд управления». Из этого меню вызывается функция, основное назначение которой — служить посредником при передаче команд от пользователя (режим «командной строки») к микроконтроллеру через интерфейс последовательного порта. Текст функции приведен ниже — он намного проще предыдущего и не нуждается в дополнительных комментариях. Выход из цикла — команда *quit*.

```

1     def console():
2         while True:
3             comm = input("Введите команду (quit для выхода): ")
4             if comm[:4] == "quit":
5                 break
6             comm = comm + "\\n"
7             # unicode переводим в битовый массив
8             ser.write(comm.encode())
9             time.sleep(0.1)
10            # битовый массив переводим обратно в строку
11            line = ser.readline().decode()
12            # убираем из выдачи последние символы \\r\\n
13            line = line[:-2]
14            print(line)

```

Конечно, созданный интерфейс для виртуального последовательного порта можно было заменить любой из множества существующих программ-терминалов, работающих с последовательным портом. Но зачем, если реализация такого терминала занимает не более 11 строк кода на Python?

4.3 Работа с платой сбора данных *National Instruments*

Фотоэлектронный умножитель (ФЭУ) является устройством, которое преобразует падающий на него поток излучения в электрический сигнал. Именно такой вариант регистрации сигнала используется в нашей установке. При этом чем выше интенсивность света — тем выше напряжение, снимаемое с выхода ФЭУ. Однако диапазон изменения соответствующего сигнала весьма широк, а нам обычно требуется обеспечить высокую чувствительность и высокое отношение «сигнал/шум» как при очень слабом, так и при очень сильном сигнале, поступающем с ФЭУ. Как вариант, для регистрации такого сигнала необходимо использовать аналого-цифровой преобразователь, перед которым ставить усилители с переключаемым диапазоном усиления сигнала, что само по себе является довольно сложной схемотехнической задачей. Другое решение — использовать на выходе ФЭУ усилитель-дискриминатор, который выдает последовательность импульсов со скоростью, пропорциональной току ФЭУ. Таким образом, чем сильнее освещается входное окно ФЭУ, тем чаще идут импульсы. При этом темновой ток ФЭУ оставим за порогом дискриминации. Возьмем за основу этот вариант. Тогда для регистрации соответствующего сигнала нам потребуется плата сбора данных, в которой реализован счетчик импульсов. В нашем случае использовалась плата NI PCI 6250, обладающая необходимым нам функционалом. Она содержит два 32-разрядных счётчика/таймера, разрешения каждого из которых будет вполне достаточно для решения поставленной задачи.

Так же, как и в примере с термопарой (глава 2), для работы с NI PCI 6250 будем использовать поддержку на уровне операционной системы в виде пакета драйверов от компании National Instruments, а для Python — пакет *nidaqmx*.

Строки инициализации платы сбора данных можно поместить в описанную ранее функцию инициализации экспериментальной установки. Инициализируем один счетчик импульсов, а усилитель-дискриминатор ФЭУ подключаем ко входной линии PFI 8 платы:

```
1 task = nd.Task()
2 task.ci_channels.add_ci_count_edges_chan('Dev1/ctr0')
```

Для корректного завершения работы программы не забудем предварительно закрыть интерфейсы для задействованных внешних устройств:

```
1 if meas_flag:
2     task.close()
3     ser.close()
```

4.4 Режим измерения

Пункты меню «2. Ввод параметров измерения» и «3. Просмотр параметров измерения» достаточно просты для реализации и здесь не рассматриваются. Лишь упомянем, что все параметры, редактируемые пользователем, записываются в словарь *params*:

```
params = {'l_start': 400.0, 'l_stop': 700.0, 'l_step': 10.0, 't_meas': 1.0,
'data_path': './DATA/'}
```

Это позволяет очень легко организовать сохранение параметров при выходе из программы и их загрузку при старте, что экономит время при серийных измерениях с одинаковыми условиями эксперимента. Кроме того, добавление в словарь какого-нибудь нового параметра никак не скажется на файловых операциях:

```

1 import pickle
2
3 # начало работы программы
4 try:
5     with open('arti.conf', 'rb') as f:
6         params = pickle.load(f) # найден файл конфигурации
7 except (FileNotFoundError, IOError):
8     # если программа запущена впервые
9     with open('arti.conf', 'wb') as f:
10        pickle.dump(params, f)
11 # завершение работы программы
12 with open('arti.conf', 'wb') as f:
13    pickle.dump(params, f)

```

В данном случае был использован стандартный модуль *pickle*, предназначенный для записи в файл сериализованных объектов. Заметим, что при этом создается бинарный файл, который невозможно просмотреть в текстовом редакторе.

Пункт меню «4. Старт измерения» предполагает проведение следующих операций: а) выход на начальную длину волны (l_{start}), б) измерительный цикл с шагом l_{step} до конечной длины волны l_{stop} .

```

1 import matplotlib.pyplot as plt
2
3 # следующий текст располагается после if item == 3:
4 if meas_flag == False:
5     input('Не инициализирована установка! Нажмите <Enter>')
6     continue
7 print('Идем на начальную длину волны... "X" для отмены')
8 s = 'goto ' + str(params['l_start']) + '\n'
9 ser.write(s.encode())
10 time.sleep(0.1)
11 key = 0
12 run_flag = True
13 if wait_mdr(): # функция ожидания событий
14     print('Начинаем измерительный цикл...')
15     data = []
16     plt.ion()
17     plt.ylabel('Интенсивность, имп.')
18     plt.xlabel('Длина волны, нм')
19     plt.title('Идет измерение')
20     l_work = params['l_start']
21     plt.xlim(params['l_start'], params['l_stop'])
22     meas_point(l_work) # измерительная функция
23     while (l_work < params['l_stop']) & (run_flag):
24         l_work += l_step
25         s = 'wave ' + str(l_step) + '\n'
26         ser.write(s.encode())
27         time.sleep(0.1)
28         run_flag = wait_mdr()
29         meas_point(l_work)
30     plt.ioff()
31     with open("curr_meas.tmp", "wb") as f:
32         pickle.dump(data, f)
33     print('Измерения завершены.')
34     plt.close()
35 else:
36     print('Измерения прерваны!')
37     input('Для выхода в основное меню нажмите <Enter>')

```

Поскольку измерения могут продолжаться достаточно длительное время, было бы удобным сразу отображать текущий результат в виде графика. Именно поэтому в строке 16 мы инициализируем графику в интерактивном режиме. Поскольку границы диапазона по длинам волн определены параметрами измерения, используем метод *plt.xlim* для установки масштаба по оси абсцисс. Осуществлять измерение и выводить точки на график будет функция *meas_point*, о которой скажем чуть позже. Строки 23—29 составляют основной измерительный цикл программы, после которого мы выключаем интерактивный режим графика (чтобы графическое

окно не оказалось заблокированным). Экспериментальные данные записываются в список *data* неопределенной длины, который инициализируется в строке 15. Сразу же после измерения данные записываются во временный файл как сериализованный объект. Это избавляет экспериментатора от возможных случайных ошибок на финишной стадии эксперимента. Впоследствии полученные данные можно сохранить в обычный текстовый файл и снабдить необходимыми комментариями.

В программе предусмотрен аварийный останов позиционирования монохроматора на любой стадии процесса измерения. После отсылки команды *goto* (строка 8) на уровень Arduino программа переходит к ожиданию выполнения действия (функция *wait_mdr*, строка 13). Однако это не простое ожидание:

```
1 import msvcrt
2
3 def wait_mdr():
4     key = 0
5     flag = True
6     while True:
7         res = ser.readline().decode()
8         time.sleep(0.1)
9         if msvcrt.kbhit(): # не прерываем цикл
10            key = msvcrt.getch() # берём код символа
11            if (key == b'x') | (key == b'X'):
12                ser.write(b'x\n')
13                time.sleep(0.1)
14            if res.find('.') >= 0:
15                if res.find('X') >= 0:
16                    flag = False
17            return flag
```

Во время ожидания (бесконечный цикл *while True*) программа анализирует буфер клавиатуры, используя модуль *msvcrt*. Обратите внимание, что этот модуль корректно работает только в окне терминала Windows, для программ под Linux есть иные решения. Метод *kbhit* проверяет наличие кода нажатой клавиши в буфере, не останавливая цикл. Таким образом, мы не мешаем программе отслеживать состояние последовательного порта. В случае, если мы нажали на клавиатуре клавишу «X», символ «x» отправится в контроллер Arduino (строка 12). Программа контроллера, занимаясь вращением шагового двигателя, попутно анализирует последовательный порт. Если там появляется «x», происходит прерывание работы программы позиционирования и в порт возвращается «X». Программа на Python принимает этот символ (строка 15) и останавливает измерительную процедуру.

Как было сказано выше, измерение, вывод точки на график и запись нового значения в массив данных осуществляются функцией *meas_point*. В качестве параметра ей передается текущее значение длины волны.

```
1 def meas_point(wave):
2     print('длина волны: ', wave)
3     task.start()
4     time.sleep(params['t_meas'])
5     res = task.read()
6     task.stop()
7     print('измеренное значение: ', res)
8     global data
9     data.append([wave, res])
10    PX, PY = zip(*data)
11    plt.plot(PX, PY, linestyle = '--', color = 'g', lw = 1.0,
12            marker='+', markeredgcolor = 'r', markersize = 4)
13    plt.pause(0.1)
```

Эта функция не очень эстетична с точки зрения структурного подхода к программированию, поскольку здесь осуществляется взаимодействие с несколькими разнородными устройствами на чтение и запись, а также одновременное использование глобальных переменных и параметров функции. Однако, для проведения тестирования установки это оказалось удобно. В ходе

измерительного цикла в первую очередь осуществляется запуск счетчика импульсов и остановка через промежуток времени, установленный параметром *t_meas* (строки 3—6). Полученное значение *res* выводится на консоль (в виде числа). В список данных добавляется новое значение с помощью метода *append* (строка 9).

И, наконец, в строках 10—13 данные выводятся на график. Следует отметить, что, в отличие от примеров главы 2, использовать метод *plt.scatter* оказалось не очень удобно — в нашем эксперименте точек обычно не бывает более 500, и очень часто их менее 100. При этом маркеры, не связанные линией, не дают экспериментатору хорошего восприятия данных. Однако построить в интерактивном режиме график с маркерами точек и линиями, соединяющими точки, тоже очень просто. Достаточно использовать стандартный метод *plt.plot* с необходимыми значениями параметров. Требуется лишь задавать в качестве одномерных массивов *PX* и *PY* весь набор экспериментальных значений, полученных на данном шаге. Поскольку список заполняется построчно, а требуются столбцы абсцисс и ординат, хорошим решением является использование функции *zip*, которая из исходного списка, состоящего из вложенных списков, вновь формирует списки, состоящие из последовательности элементов с одинаковыми индексами. Символ «*» при аргументе убирает внешний список, позволяя распаковать исходный набор данных в две переменных. Использование метода *plt.pause* является обязательным для корректного отображения графика, однако, следует помнить, что время паузы «мертвое», никаких действий программа при этом не совершает, и этот промежуток времени желательно сделать как можно более коротким.

4.5 Просмотр и сохранение результатов измерений

Оставшиеся для реализации пункты меню «5. Просмотр результатов измерения и запись в файл» и «6. Просмотр файлов данных» не содержат каких-либо важных нововведений и предполагают, в первую очередь, правильную обработку различного вида исключений. Формат итогового файла для записи результата предполагается текстовым с построчным размещением данных. Разграничитель в строке между числами — символ табуляции (*\t*). Код функции для записи файла данных выглядит следующим образом:

```
1 import os, datetime
2
3 # следующий текст может быть размещён после if item == 4:
4 s = 'y'
5 try:
6     with open("curr_meas.tmp", "rb") as f:
7         data = pickle.load(f)
8 except Exception as e:
9     print('Нечего показывать!')
10    s = 'n'
11 if s == 'y': # просмотр автоматически сохранённых данных
12     print()
13     for row in data:
14         print(row)
15     print()
16     plt.ylabel('Интенсивность, имп.')
17     plt.xlabel('длина волны, нм')
18     plt.title('Просмотр предыдущего измерения')
19     plt.plot([d[0] for d in data], [d[1] for d in data],
20             linestyle = '-', color = 'b', lw = 0.7,
21             marker='o', markerfacecolor = 'r', markersize = 4)
22     print('Закройте окно графика для продолжения.')
23     plt.show()
24     # запись файла на диск
25     fname = input('Введите имя файла: ')
26     if len(fname) > 0:
27         fname = params['datapath'] + fname + '.txt'
28         if os.path.isfile(fname):
29             s = input('Файл '+ fname + ' уже существует.' + \
30                     ' Переписать (Y/N)? ')
31             s = s.lower()
```

```

32         if s == 'y':
33             comment = input('Введите комментарий: ')
34             with open(fname, 'w') as f:
35                 f.write('ARTI data file. '+ \
36                     datetime.datetime.today().strftime("%d-%m-%Y-%H:%M:%S") + \
37                     '\n')
38                 f.write(comment + '\n')
39                 f.write('wavelength, nm \tintensity, imp.\n')
40                 for row in data:
41                     for element in row:
42                         f.write('%s\t' % element)
43                         f.write('\n')
44                 print('Файл записан.')
45         else:
46             print('данные НЕ сохранены!')
47     else:
48         print('данные НЕ сохранены!')
49     input('Для выхода в основное меню нажмите <Enter>')

```

Дадим некоторые пояснения к представленному коду. Сначала читаются данные из файла *curr_meas.tmp*, автоматически сохраненные в процессе измерения (строки 5—9). Значение переменной *s* (строка 4) определяет логику выполнения фрагментов кода. Если файл прочитан корректно, то данные будут загружены в список *data* и будут отображены в текстовом виде в терминальном окне (строки 13—14). График по этим же экспериментальным значениям строится в строках 16—23. Поскольку здесь мы не использовали *numpy*-массивы, для выборки необходимых столбцов из списка *data* можно использовать *for*-генераторы (строка 19). Пока окно графика не будет закрыто, программа будет ожидать на строке 23.

Со строки 25 пользователю будет предложено записать показанные данные в файл. Здесь также переменная *s* используется для управления логикой работы с файлами. Путь к каталогу с экспериментальными данными задаётся в параметре *params['datapath']*. В программе можно предусмотреть код для изменения значения этого параметра. Модуль *os* здесь используется для того, чтобы выяснить, нет ли уже на диске файла с заданным вами именем (строка 28). Модуль *datetime* потребовался для записи даты и времени создания файла как часть комментария к данным (строка 35).

Структура данного текстового файла является авторским решением и отличается некоторой избыточностью сохраняемой информации. Собственно цикл записи набора экспериментальных данных в файл в текстовом формате находится в строках 40—43. Для записи был использован метод *write* для записи каждого отдельного значения, отделенного табуляцией, строка 43 потребовалась для установки символа перехода на новую строку. Строки с 34 по 39 формируют заголовок файла, содержащий необходимую информацию о сохраняемых данных.

Функция для чтения и отображения файла данных написана сходным образом, и здесь ее код не приводится, приведем только результат ее работы — рисунок 27.

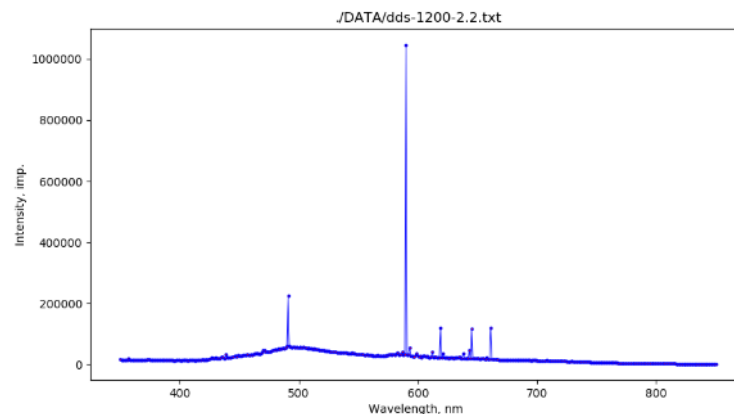


Рис. 27. Пример вывода на график измеренного спектра

4.6 Организация работы с базой данных

В случае достаточно интенсивно используемого экспериментального оборудования для параллельной работы над различными научными задачами всегда возникает проблема в виде огромных плохо структурированных каталогов, каждый из которых содержит несколько сотен файлов экспериментальных данных со сходными именами. Поиск результатов, измеренных, например, несколько лет назад, обычно превращается в длительный квест. Решение данной проблемы очевидно: программа должна иметь возможность работать с базой данных, т. е. напрямую записывать, а также проводить поиск и загружать из базы данных необходимые результаты. Поскольку реализация этой задачи в целом достаточно трудоемка, мы рассмотрим базовые моменты взаимодействия сетевой реляционной СУБД *PostgreSQL* с программой на Python.

Инсталлятор СУБД *PostgreSQL* и подробная инструкция по установке находятся в свободном доступе на сайте разработчиков [18]. Для взаимодействия с СУБД *PostgreSQL* будем использовать возможности консоли *iPython*, поскольку создавать структуру базы нам придется с использованием команд языка SQL. Для этого дополнительно потребуется установить, как минимум, три пакета:

```
> pip install ipython-sql
> pip install psycopg2
> pip install pgcli
```

Если по каким-либо причинам у вас не установился пакет *pgcli*, вы, безусловно, можете использовать любую из программ-оболочек для *PostgreSQL*, в том числе устанавливаемую вместе с этой СУБД клиентскую программу *psql*. Если же все установилось, запустите *iPython* и введите команду:

```
In[1]: %load_ext pgcli.magic
```

Попробуем соединиться с СУБД, которая у нас находилась по локальному IP-адресу (если ваша СУБД работает на том же компьютере, что и Python, вместо конкретного адреса введите *localhost:5432*) под аккаунтом администратора (имя *postgres*, база данных *postgres*):

```
In[2]: pgcli postgres://postgres:myPassword@172.20.180.147:5432/postgres
```

Если соединение было успешным, вы попадаете в консоль управления СУБД:

```
postgres@172:postgres>
```

Создадим базу данных для хранения наших экспериментальных результатов и определим пользователя, от имени которого мы будем работать с данными. Этому пользователю необходимо разрешить чтение, добавление, изменение и удаление данных для всех таблиц из базы данных. Для этого требуется выполнить несколько стандартных SQL-инструкций администрирования. В настоящем пособии основы языка SQL не рассматриваются, поэтому мы можем лишь порекомендовать читателям обратиться к соответствующим учебным ресурсам по данному вопросу и прежде всего к учебникам по СУБД *PostgreSQL* [19, 20].

В первую очередь следует соединиться с базой данных *postgres* с аккаунтом администратора и выполнить следующие команды:

```
...> create database experiments;
...> create user labo with encrypted password 'ВашПароль';
...> grant connect on database experiments to labo;
```

Теперь выйдем из базы postgres...

```
...> quit;
```


... и соединимся с созданной базой данных *experiments* все с тем же аккаунтом администратора. Далее выполним команды:

```
...> create schema exp;
...> grant usage on schema exp to labo;
...> grant all privileges on all sequences in schema exp to labo;
...> grant select, insert, update, delete on all tables in schema exp to labo;
...> alter user labo set search_path to exp, public;
```

Далее создадим связанные внешними ключами реляционные таблицы, в которые будем записывать информацию о проведенных экспериментах и сохранять сами экспериментальные данные.

...> create table exp.sets(id serial primary key, name varchar(150), description varchar(400));	1. описание изм. установок идентификатор записи название установки описание установки
...> create table exp.meatypes(id serial primary key, description varchar(400));	2. описание видов измерений идентификатор записи название изм. методики
...> create table exp.results(id bigserial primary key, name varchar(150) not null, measdate timestamp, setid int references sets(id), typeid int references meatypes(id), comment varchar(600), data text);	3. хранение результатов эксперимента идентификатор записи название измерения дата и время измерения ссылка на тип измерит. установки ссылка на вид измерения описание измерения полученные данные

Мы реализовали простейшую структуру базы данных, состоящую из трех таблиц, имеющих ключевые поля *id* в виде счетчиков, значение которых автоматически увеличивается с каждой новой записью. Таблицы *sets* и *meatypes* являются дополнительными и служат для уточнения вида хранимых данных. Эти таблицы связаны с таблицей *results* с помощью внешних ключей (поля *setid* и *typeid*). В таблицу *results* будут заноситься результаты экспериментов с помощью программы, которую напишем на Python.

Выйдем теперь из режима суперпользователя и попробуем соединиться с базой данных с помощью логина и пароля для пользователя *labo*.

```
...> quit;
In[3]: pgcli postgres://labo:МойПароль@172.20.180.147:5432/experiments
labo@172:experiments>
```

Теперь мы работаем от имени пользователя, которому разрешено читать и записывать в созданные таблицы. Именно этот аккаунт будет использовать программа на Python. Наполним таблицы *Sets* и *MeasTypes* некоторым содержанием:

```
...> insert into sets(name, description) values  
( 'MDR-2019', 'Спектральный комплекс на базе монохроматора МДР-23'),  
( 'VA-vacuum', 'Измерение вольт-амперных характеристик на базе Keithley 6485');  
  
...> insert into meatypes(description) values  
( 'Измерение спектров люминесценции'),  
( 'Измерение кинетик люминесценции'),  
( 'Измерение вольт-амперных характеристик');
```

Посмотрим содержимое таблиц *Sets* и *MeasTypes* (рис. 28).

Далее напишем функцию на языке Python, которая записывает информацию в базу данных. Предположим, у нас есть файл со спектроскопическими данными, который мы получили в предыдущей задаче, и созданные нами функции можно будет включить в общую программу

управления спектральной установкой. В представленной ниже функции параметр *d_data* — это словарь, который включает в себя и параметры измерения, и обозначение осей координат, и набор результатов измерений. Для того чтобы корректно записать и, самое главное, воспроизвести этот набор данных, мы сохраняем их в поле *data* таблицы *results* в формате JSON.

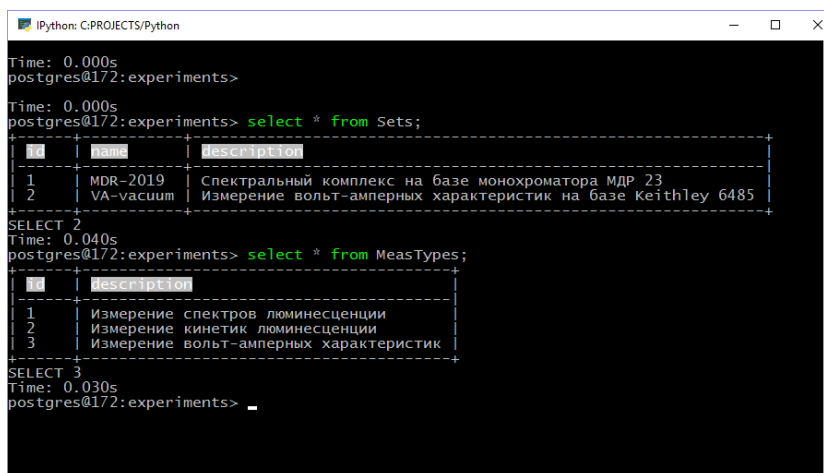


Рис. 28. Просмотр заполненных таблиц в базе данных

```

1 def dbSaveData(d_data):
2     d_name = input('Введите наименование измерения: ')
3     d_comment = input('Введите комментарий: ')
4     try:
5         conn = psycopg2.connect(host='172.20.180.147',
6                                 port='5432', database='experiments',
7                                 user='labo', password='мойпароль')
8         cur = conn.cursor()
9         cur.execute(''INSERT INTO results
10                    (name, measdate, setid, typeid, comment, data)
11                    VALUES ( %s, now(), 1, 5, %s, %s )'' ,
12                    (d_name, d_comment, json.dumps(d_data)))
13         conn.commit()
14         print('данные записаны в базу.')
15     except (Exception, psycopg2.Error) as error:
16         print('Ошибка записи данных в базу! ', error)
17     finally:
18         if conn:
19             cur.close()
20             conn.close()

```

Формат JSON (JavaScript Object Notation), хоть и имеет отношение к языку *JavaScript*, но в настоящее время обрел популярность как универсальный текстовый формат обмена данными [21]. Этот формат достаточно компактно описывает структуру данных, при этом сама информация остается человеко-читаемой, а значит, при необходимости можно вручную вносить правку в JSON-строку. В нашем случае информация, которая будет заноситься в поле *data* таблицы *results*, будет иметь вид, подобный этому:

```

{"parameters": {"l_start": 400.0, "l_stop": 500.0,
"l_step": 1.0, "t_meas": 1.0}, "coordinates":
["wavelength, nm ", "Intensity, imp."], "data": [[350.0,
16012.0], [351.0, 15365.0], [352.0, 15685.0], [353.0,
15432.0], [354.0, 15271.0], [355.0, 15652.0], [356.0,
15232.0], [357.0, 18715.0], [358.0, 15262.0], [359.0,
14974.0], [360.0, 15159.0], [361.0, 15149.0], [362.0,
15252.0], [363.0, 15075.0], [364.0, 15342.0], [365.0,
15260.0], [366.0, 15097.0], [367.0, 15016.0], [368.0,
15228.0], [369.0, 15135.0], [370.0, 15479.0]]}]

```

Из представленного JSON-кода довольно легко выделить структуру данных в стиле словарей и списков Python. Очевидно, что для другой экспериментальной методики структура данных

наверняка будет какой-либо иной. Важно, что в любом случае достаточно одного метода `json.dumps()`, чтобы реализовать предложенную схему хранения (строка 12). Не забудьте добавить `import json` в начало программы.

Как же реализована запись в базу данных? Как обычно, Python предлагает весьма лаконичный код для реализации такого действия, поскольку основной функционал реализован в модуле `psycopg2`. Метод `psycopg2.connect()` (строки 9—12) создает соединение с базой данных, которое по окончании обмена данными должно быть закрыто с помощью метода `close()`. Все операции по взаимодействию с базой данных мы заключили в конструкцию `try... except... finally` для корректной обработки возможных ошибок без прерывания работы программы. Для выполнения запроса нам потребуется метод `cursor` объекта `conn` (строка 8). Курсор для баз данных представляет собой широко используемую идеологию последовательного доступа к данным, при этом сам набор данных, который мы можем сканировать с помощью курсора, формируется с помощью SQL-запроса. Однако в данном случае мы не читаем, а пишем в базу данных, поэтому ограничимся только выполнением одной SQL-инструкции. Строки 9—12 содержат параметризованный запрос **INSERT INTO** на добавление одной строки данных. Сам SQL-запрос оформлен в тройные кавычки для того, чтобы его можно было разместить на нескольких строках. На самом деле язык SQL строко-независимый, и это сделано только для удобства восприятия кода. Операторы форматирования `%s` внутри строки представляют типичную для Python реализацию форматирования текстовых строк с возможностью введения нескольких параметров. Они определяют место в строке SQL-инструкции, куда будут введены значения переменных `d_name`, `d_comment` и `d_data`, которые необходимо занести в базу данных. Важно, что набор переменных должен быть оформлен в виде кортежа — это второй аргумент метода `execute`. Даже если переменная одна — она должна быть оформлена как элемент кортежа.

Поскольку мы не ждем возврата результата от базы данных, необходимо завершить транзакцию (тем самым запустив выполнение запроса) с помощью `conn.commit()` и закрыть соединение с базой данных. Используя командную строку `iPython + pgcli`, можно посмотреть, изменилось ли содержимое таблицы `results` (`SELECT * FROM results`). Если ваша запись содержит русские символы, возможны проблемы с их отображением в терминале `iPython + pgcli`, однако при чтении соответствующей информации с помощью Python-программы проблем не должно быть. Рассмотрим следующую тестовую функцию, которая читает информацию из базы данных:

```
1 def dbLoadData(id):
2     block = 0
3     try:
4         conn = psycopg2.connect(host='172.20.180.147',
5                                 port='5432', database='experiments',
6                                 user='labo', password='МойПароль')
7         cur = conn.cursor()
8         cur.execute('''SELECT r.name, r.measdate,
9                        m.description, s.name, r.comment, r.data
10                       FROM results r JOIN meastypes m
11                       ON r.typeid = m.id JOIN sets s
12                       ON r.setid = s.id WHERE r.id = %s''', (id,))
13         rows = cur.fetchall()
14         for row in rows:
15             print(row[0], '\n', row[1], '\n', row[2], '\n',
16                   row[3], '\n', row[4])
17             block = json.loads(row[5])
18             print('Прочитано данных: ', len(block['data']))
19     except (Exception, psycopg2.Error) as error:
20         print('Ошибка чтения из базы данных! ', error)
21     finally:
22         if conn:
23             cur.close()
24             conn.close()
25     return block
```

Функция выводит только одну запись из всех связанных таблиц, значение поля `id` которой в таблице `results` соответствует параметру функции `id`. Поэтому курсор принимает значение

SELECT-запроса с условием WHERE, который выдаст шесть «столбцов» данных выбранной нами записи. Собственно экспериментальные результаты находятся в JSON-массиве в последнем столбце, который мы конвертируем обратно в словарь Python с помощью метода *json.loads* (строка 17).

Конечно, для этой функции необходимо указать параметр *id*, который как-то надо узнать или получить из таблицы *results*, например, в результате поиска записи по заданным условиям. Но поскольку мы тестируем работу с базой данных, вызов функции *dbLoadData* может быть оформлен так:

```
1 id_rec = input('Введите id записи: ')
2 res = dbLoadData(id_rec)
3 if res !=0:
4     s = input('показать график? (Y/N): ')
5     if s.lower() == 'y':
6         title='Загружено из БД'
7         showData(title, res['coordinates'],
8                 res['data'])
9     else:
10    print('ничего показывать!')
```

Полагаем, что реализацию функции *showData*, которая изображает график спектра (строки 7—8), вы сможете без особых проблем написать самостоятельно.

4.7 Заключительные комментарии

Мы предполагаем, что роль нашего пособия — это не учебник по азам языка Python и не книга по постановке лабораторных экспериментов. Это своего рода коллекция «лайфхаков», или сборник рецептов, который полезно иметь под рукой любому разработчику. Однако многие важные моменты описания языка заведомо остались вне нашего поля зрения — например, конструирование объектно-ориентированных программ. Но мы уверены, что следующие по одному с нами пути обязательно обратятся за необходимой информацией к очень хорошим учебникам по Python, которые охватывают различные аспекты применения языка [22—29].

Конечно, потенциал Python таков, что, как можно было убедиться на представленных примерах, даже приложение, работающее в командной строке, может полноценно управлять автоматизированной экспериментальной установкой и заниматься визуализацией полученных данных на уровне, достойном лучших научных изданий. Но совершенно очевидно, что останавливаться на таких программах противоестественно. В частности, продолжение работы с базой данных потребует реализации полноценного поиска по всей информации, доступной в описании экспериментальных данных. Но реализовать этот поиск удобнее всего (и проще всего) в приложении с графическим интерфейсом, управляемым событиями, где возможна корректная визуализация списков и таблиц.

В заключение хотелось бы поблагодарить сообщество Python-программистов, которые делятся своими разработками, советами, комментариями на различных интернет-площадках мирового уровня, и прежде всего на GitHub [30]. Их роль весьма важна и значима для развития и популяризации языка и привлечения интереса к нему. Примеры, рассмотренные в данном пособии, показывают, что наличие мощных библиотек очень упрощает решение многих практических задач. Но удобство и лаконичность языка Python при этом неоспоримы.

Конечно, представленные проекты не бесспорны и не академичны. Они, как и наши знания и умения, постоянно совершенствуются и видоизменяются. Поэтому про самое нужное и интересное... пока еще не написано.

Список литературы

1. *Kass, S.* The 2018 Top Programming Languages [Электронный ресурс] / S. Kass. — Электрон. ст. — [IEEE Spectrum], 2018. — URL: <https://spectrum.ieee.org/>, свободный. — (22.06.2019).
2. [tiobe.com/tiobe-index](https://www.tiobe.com/tiobe-index) [Электронный ресурс] : сайт TIOBE software BV: TIOBE Index for December 2018. — URL: <https://www.tiobe.com/tiobe-index>. — (22.06.2019).
3. *Cox, T.* Raspberry Pi Cookbook for Python Programmers / Т. Кох. — Birmingham : Packt Publishing Ltd., 2014. — 391 p.
4. [python.org/downloads](https://www.python.org/downloads) [Электронный ресурс] : сайт Python Software Foundation: Download Python. — URL: <https://www.python.org/downloads>. — (22.06.2019).
5. *Тэйлор, Дж.* Введение в теорию ошибок / Дж. Тэйлор. — Москва : Мир, 1985. — 272 с.
6. ni.com/downloads [Электронный ресурс] : сайт компании National Instruments: Загрузка ПО и драйверов. — URL: <http://ni.com/downloads>. — (22.06.2019).
7. nidaqmx-python.readthedocs.io/en/latest/index.html [Электронный ресурс] : сайт NI-DAQmx Python API: NI-DAQmx Python Documentation. — URL: <https://nidaqmx-python.readthedocs.io/en/latest/index.html>. — (22.06.2019).
8. gist.github.com/Uberi/283a13b8a71a46fb4dc8 [Электронный ресурс] : сайт GitHub Gist: Anthony Zhang Uberi/realtime-plot.py. — URL: <https://gist.github.com/Uberi/283a13b8a71a46fb4dc8>. — (22.06.2019).
9. pyvisa.readthedocs.io/en/master [Электронный ресурс] : сайт PyVISA master: Control your instruments with Python. — URL: <https://pyvisa.readthedocs.io/en/master>. — (22.06.2019).
10. libusb.info [Электронный ресурс] : сайт проекта libusb: A cross-platform user library to access USB devices. — URL: <https://libusb.info/>. — (22.06.2019).
11. zadig.akeo.ie [Электронный ресурс] : сайт проекта Zadig: USB driver installation made easy. — URL: <https://zadig.akeo.ie/>. — (22.06.2019).
12. [scipy.org/about.html](https://www.scipy.org/about.html) [Электронный ресурс] : сайт SciPy.org: Scientific Computing Tools for Python. — URL: <https://www.scipy.org/about.html>. — (22.06.2019).
13. <http://www.jcamp-dx.org> [Электронный ресурс] : сайт IUPAC Committee on Printed and Electronic Publications: Subcommittee on Electronic Data Standards (JCAMP-DX), Homepage of the IUPAC CPEP Subcommittee on Electronic Data Standards. — URL: <http://www.jcamp-dx.org/>. — (22.06.2019).
14. pypi.org/project/jcamp [Электронный ресурс] : сайт проекта Find, install and publish Python packages with the Python Package Index: jcamp 1.2.1. — URL: <https://pypi.org/project/jcamp/>. — (22.06.2019).
15. *Петин, В. А.* Arduino и Raspberry Pi в проектах Internet of Things / В. А. Петин. — Санкт-Петербург : БХВ-Петербург, 2016. — 320 с.
16. *Иго, Т.* Arduino, датчики и сети для связи устройств / Т. Иго. — Санкт-Петербург : БХВ-Петербург, 2015. — 544 с.
17. *Монк, С.* Программируем Arduino. Профессиональная работа со скетчами / С. Монк. — Санкт-Петербург : Питер, 2017. — 252 с.
18. [postgresql.org](https://www.postgresql.org/) [Электронный ресурс] : сайт PostgreSQL: The World's Most Advanced Open Source Relational Database. — URL: <https://www.postgresql.org/>. — (22.06.2019).
19. postgrespro.ru/docs [Электронный ресурс] : сайт PostgresPro: Документация PostgreSQL и Postgres Pro. — URL: <https://postgrespro.ru/docs>. — (22.06.2019).
20. *Моргунов, Е. П.* PostgreSQL. Основы языка SQL : учебное пособие / Е. П. Моргунов. — Санкт-Петербург : БХВ-Петербург, 2018. — 336 с.
21. [json.org](https://www.json.org/) [Электронный ресурс] : сайт ECMA-404 The JSON Data Interchange Standard: Introducing JSON. — URL: <https://www.json.org/>. — (22.06.2019).
22. *Лутц, М.* Изучаем Python / М. Лутц. — 4-е изд. — Санкт-Петербург : Символ-Плюс, 2011. — 1280 с.
23. *Лутц, М.* Программирование на Python / М. Лутц. — 4-е изд. — Санкт-Петербург : Символ-Плюс, 2011. — Т. 1. — 990 с.; — Т. 2. — 992 с.

24. *Доусон, М.* Программируем на Python / М. Доусон. — 3-е изд. — Санкт-Петербург : Питер, 2014. — 416 с.
25. *Прохоренок, Н. А.* Python 3 и PyQt 5. Разработка приложений / Н. А. Прохоренок, В. А. Дронов. — Санкт-Петербург : БХВ-Петербург, 2016. — 832 с.
26. *Любанович, Б.* Простой Python. Современный стиль программирования / Б. Любанович. — Санкт-Петербург : Питер, 2016. — 480 с.
27. *Stewart, J. M.* Python for Scientists. Second Edition / J. M. Stewart. — New York : Cambridge University Press, 2017. — 257 p.
28. *VanderPlas, J.* Python Data Science Handbook / J. VanderPlas. — Sebastopol : O'Reilly Media, Inc., 2017. — 530 p.; то же. — URL: <https://jakevdp.github.io/PythonDataScienceHandbook/>, свободный. — (22.06.2019).
29. *Phuong, Vo. T. H.* Python: Data Analytics and Visualization / Vo. T. H. Phuong, M. Czygan, A. Kumar. — Birmingham : Packt Publishing, 2017. — 866 p.
30. github.com/python [Электронный ресурс] : сайт GitHub, Python: Repositories related to the Python Programming language. — URL: <https://github.com/python>. — (22.06.2019).

Учебное электронное издание

Пикулев Виталий Борисович
Логинова Светлана Владимировна

Язык Python в приложении к экспериментальным исследованиям

Учебное электронное пособие

Редактор *Л. М. Дейнега*

Оригинал-макет, электронная версия и оформление обложки *Н. Н. Осипов*

Подписано к изготовлению 03.12.2019.
1CD-R. 2.7 Мб. Тираж 100 экз. Изд. № 149



Федеральное государственное бюджетное образовательное
учреждение высшего образования
ПЕТРОЗАВОДСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
185910, г. Петрозаводск, пр. Ленина, 33

<https://petsu.ru>
Тел.: (8142) 71-10-01

Изготовлено в Издательстве ПетрГУ
185910, г. Петрозаводск, пр. Ленина, 33
URL: press.petsu.ru/UNIPRESS/UNIPRESS.html
Тел./факс: (8142) 78-15-40
nvrahomova@yandex.ru