Аллен Б. Дауни

Think DSP

Цифровая обработка сигналов на Python

Think DSP

Digital Signal Processing in Python

Allen B. Downey

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo O'REILLY®

Think DSP

Цифровая обработка сигналов на Python

Аллен Б. Дауни

Москва, 2017

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО

УДК 534:004.438Python
ББК 22.32с
Д21

Д21 Аллен Б. Дауни
Think DSP. Цифровая обработка сигналов на Python / пер. с англ. Бряндинский А. Э. – М.: ДМК Пресс, 2017. – 160 с.: ил.

ISBN 978-5-97060-454-0

Изучить обработку сигналов легко – достаточно знания основ математики и программирования на Python. Обычно изучение этой сложной темы начинают с теории, а в основу данной книги положены сугубо практические примеры. Уже в первой главе звук будет разложен на гармоники, которые модифицируются и создают новые звуки. Кроме того, в книге рассмотрены: периодические сигналы и их спектры; гармоническая структура простого сигнала; чирпы и иные звуки с изменяющимся во времени спектром; шумовые сигналы и естественные источники шума; дискретное косинусное преобразование (ДКП) для сжатия информации; дискретное и быстрое преобразования Фурье для спектрального анализа, а также многое другое.

Издание будет полезно всем, кто интересуется цифровой обработкой сигналов.

УДК 534:004.438Python
ББК 22.32с

ISBN 978-1-49193-845-4 (англ.)
ISBN 978-5-97060-454-0 (рус.)

Оглавление

Введение

Обработка сигналов – одна из любимых тем автора. Она полезна во многих областях науки и техники – и, если постичь основы, то будет легче разобраться во многих вещах, видимых в мире, и тем более в слышимых.

Но с обработкой сигналов практически никто, кроме инженеров, не знаком. Проблема в том, что большинство книг (и использующие их курсы) представляют материал «снизу вверх», начиная с математических абстракций, например с фазоров. И эти книги обычно теоретические, с немногочисленными и малополезными примерами.

Цель этой книги – научить тех, кто умеет программировать, использовать свое умение для познания нового, да еще и с удовольствием.

С таким подходом, основанным на программировании, можно сразу понять самые важные идеи. Прочитав первую главу, вы научитесь анализировать звукозаписи или иные сигналы, а также генерировать новые звуки. В каждой главе вводятся новый прием и приложение, работающее с реальными сигналами. На каждом этапе сначала изучается применение приема, а затем – его работа.

Такой подход более практичен, и, вероятно, занимателен.

Предисловие к русскому изданию

Эта книга чрезвычайно необычна. Она аканонична во всем. «Оголтелый» прагматизм автора, вероятно, может вызвать легкий обморок у специалистов с академическим образованием.

И именно этим книга хороша! Автор использует практически все методы и алгоритмы, известные на сегодняшний день, причем в ясной и прозрачной манере. Читатель избавлен от скучной теории, от мрачных тенет аналитической геометрии, линейной алгебры и высшего анализа.

Автор сразу вооружает читателя готовыми инструментами, которые пригодятся в самых разных областях науки, техники и предпринимательства, где нужна и полезна цифровая обработка сигналов.

У книги есть один недостаток – не введено понятие аналитического сигнала. Но большинство читателей от этого только выигрывает! Ведь, изучив материал книги, им будет заметно проще перейти от обработки записи звука скрипки к обработке нестационарных случайных сигналов – от радиолокаторов или гидроакустических станций до ультразвуковых диагностических приборов или масс-спектрометров и газоанализаторов.

Для кого эта книга?

Примеры и соответствующий код для этой книги написаны на Python. Надо знать основы Python и основы объектно-ориентированного программирования – хотя бы уметь применять объекты, пусть и не свои собственные.

Если язык Python вам еще не знаком, можно начать с книги Аллена Б. Дауни (Allen B. Downey) *Think Python* (это введение в Python для тех, кто никогда ранее не программировал) или с книги Марка Луца (Mark Lutz) *Learning Python* – она удобнее для людей с опытом программирования.

В книге широко использованы NumPy и SciPy, и все используемые в них функции и структуры данных подробно объясняются. А если они уже известны, то это хорошо.

Предполагается, что читатель знаком с основами математики, в том числе с комплексными числами. Весь матанализ помнить не нужно; достаточно понятий об интегрировании и дифференцировании. По мере использования линейной алгебры все будет подробно объяснено.

Работа с кодом

Код и образцы звуков, используемые в этой книге, доступны в репозитории GitHub: `https://github.com/AllenDowney/ThinkDSP`. Для тех, кто не знаком с Git и GitHub, поясним: Git – это система управления версиями, в которой можно работать с входящими в проект файлами. Коллекция файлов под контролем Git называется «репозиторий». GitHub – это хостинг для хранения репозиториев Git, и у него удобный веб-интерфейс.

На домашней странице репозитория автора в GitHub с кодом можно работать несколькими способами:

- создать копию репозитория на GitHub, нажав кнопку «вилка» (fork). Если аккаунта на GitHub еще нет, его надо завести. По-

сле разветвления (копии) появится новый, личный репозиторий GitHub; в нем удобнее отслеживать код при работе с этой книгой. Затем можно клонировать свой репозиторий, скопировав файлы на компьютер;
- клонировать репозиторий автора. Для этого аккаунт GitHub не нужен, но записать изменения обратно в GitHub уже нельзя;
- скачать файлы в архиве ZIP, нажав кнопку в правом нижнем углу страницы GitHub (если использовать Git нежелательно).

Весь код в книге работает без трансляции и в Python 2, и в Python 3.

Эта книга разработана с помощью Anaconda компании Continuum Analytics (Continuum Analytics, компания) – свободного дистрибутива Python, включающего все нужные для запуска кода пакеты (и многое другое). Устанавливается Anaconda легко. По умолчанию эта платформа ставится на уровне пользователя, а не на системном, так что права администратора не нужны. Она поддерживает и Python 2, и Python 3. Anaconda можно скачать из `http://continuum.io/downloads`.

Если использовать Anaconda нежелательно, то понадобятся следующие пакеты:

- NumPy для работы с числами (`http://www.numpy.org`);
- SciPy для научных вычислений (`http://www.scipy.org`);
- Matplotlib для визуализации (`http://matplotlib.org`).

Эти пакеты часто используются, но они не входят ни в один дистрибутив Python, и в некоторых средах их бывает трудно установить. Если будут проблемы с их установкой, рекомендуется прибегнуть к Anaconda или одному из других дистрибутивов Python, включающих эти пакеты.

Большинство упражнений – это Python-скрипты, но в некоторых также используется «блокнот Jupyter». Ознакомиться с Jupyter и изучить его можно на `http://jupyter.org`.

Есть три способа работы с блокнотами Jupyter:

1. Запустить Jupyter на своем компьютере.

   Если Anaconda установлена, то и Jupyter, вероятно, установлен по умолчанию. Проверьте это, запустив сервер из командной строки:

   ```
   $ jupyter notebook
   ```

   Если он не установлен, его можно установить в Anaconda:

   ```
   $ conda install jupyter
   ```

При запуске сервера он запустит веб-браузер по умолчанию или же создаст новую вкладку в уже открытом окне браузера.

2. Запустить Jupyter в Binder.

   Binder – это служба, запускающая Jupyter на виртуальной машине. По ссылке `http://mybinder.org/repo/AllenDowney/ThinkDSP` открывается домашняя страница Jupyter с блокнотами для этой книги, а также дополнительными данными и скриптами.

   Скрипты можно запускать и изменять их для запуска собственного кода, но виртуальная машина будет временной. Любые изменения пропадут, если пауза в работе с ней продлится более часа.

3. Просмотреть блокноты на nbviewer.

   В книге также приводятся ссылки на nbviewer, с его помощью можно просматривать код и результаты. Это удобно для чтения блокнотов и прослушивания примеров, но ни изменить код, ни запустить его, ни использовать интерактивные виджеты нельзя.

Удачи в работе!

Условные обозначения в этой книге

В этой книге используются следующие типографские соглашения:

Курсив
: Выделение новых терминов и особо значимых фрагментов текста.

Жирный шрифт
: Используется для обозначения сочетаний клавиш, элементов программного интерфейса.

`Моноширинный шрифт`
: Используется для обозначения URL-адресов и адресов электронной почты, листингов программ, имен файлов, расширений файлов, а также элементов программ, таких как имена переменных и функций, типов данных, инструкций и ключевых слов.

`Моноширинный полужирный шрифт`
: Отображает набираемые пользователем команды или иной текст.

Список корреспондентов

Предложения или замечания по поводу книги направляйте на downey@allendowney.com. Если на основе ваших сообщений будут сделаны изменения, то автора сообщения добавят в список корреспондентов (при отсутствии специальных условий).

Желательно указывать хотя бы часть предложения, в котором обнаружена ошибка, – так ее легче найти. Можно сослаться и на номер страницы или название раздела, хотя это осложняет поиск. Спасибо за понимание!

- Прежде чем автор приступил к написанию этой книги, идеи, которые легли в ее основу, развивались в общении с Булосом Хэрбом (Boulos Harb) из Google и с Аурелио Рамосом (Aurelio Ramos), ранее работавшим в Harmonix Music Systems.
- Во время осеннего семестра 2013 года Натан Линц (Nathan Lintz) и Ян Дэниэр (Ian Daniher) работали со автором над независимым учебным проектом и помогли с первым изданием этой книги.
- На DSP-форуме Reddit пользователь под ником Ramjetsoundwave помог решить проблему с реализацией броуновского шума. Другой участник форума, andodli, нашел опечатку.
- Весной 2015 года этот материал преподавался совместно с проф. Оскаром Мур-Мирандой (Oscar Mur-Miranda) и проф. Сиддхартаном Говиндасами (Siddhartan Govindasamy). Оба внесли много предложений и исправлений.
- Сайлас Гигер (Silas Gyger) исправил арифметическую ошибку.
- Джузеппе Мазетти (Giuseppe Masetti) прислал ряд полезных замечаний.

Особая благодарность техническим рецензентам – Эрику Петерсу (Eric Peters), Брюсу Ливенсу (Bruce Levens) и Джону Винсенту (John Vincent) – за многие полезные замечания, разъяснения и исправления.

Также благодарность Freesound, источнику многих звуков, используемых в книге, и пользователям Freesound, предоставившим их. Некоторые из них включены в репозиторий GitHub этой книги, причем использованы оригинальные имена файлов, так что источники легко найти.

К сожалению, большинство пользователей Freesound не указывает своих имен, поэтому здесь можно привести только их ники. Образцы

звуков, используемые в книге, были предоставлены пользователями Freesound – iluppai, wcfl10, thirsk, docquesting, kleeb, landup, zippi1, themusicalnomad, bcjordan, rockwehrmann, marcgascon7, jcveliz.

Спасибо им всем!

Глава 1. Звуки и сигналы

Сигнал представляет собой изменяющуюся во времени величину. Это определение довольно абстрактно, так что начнем с конкретного примера – звука. *Звук* – это изменение давления воздуха. *Звуковой сигнал* – изменения давления воздуха во времени.

Микрофон – это устройство, воспринимающее такие изменения и генерирующее соответствующий звуку электрический сигнал. Динамик – это устройство, принимающее электрический сигнал и производящее звук. Микрофоны и динамики называются *преобразователями*, так как они преобразуют сигналы из одного вида в другой.

Эта книга посвящена обработке сигналов, то есть процессам их синтеза, преобразования и анализа. Для изучения наиболее удобны звуковые сигналы, но все рассматриваемые методы применимы и к электронным сигналам, и к механическим вибрациям, и к сигналам вообще.

Они также применимы к сигналам, изменяющимся не во времени, а в пространстве, как, скажем, изменение высот профиля местности. Методы применимы и к многомерным сигналам, таким как изображения, то есть сигналам, изменяющимся в двумерном пространстве. Так, фильм – это сигнал, меняющийся в двумерном пространстве и во времени.

Начнем с простого одномерного звука.

Код для главы 1 находится в репозитории этой книги, в блокноте `chap01.ipynb` (см. раздел «Работа с кодом» на стр. 9). Его также можно просмотреть на `http://tinyurl.com/thinkdsp01`.

Периодические сигналы

Начнем с *периодических сигналов*, то есть с сигналов, повторяющихся через некоторый период времени. Например, колокол после удара вибрирует, издавая звук. Если записать этот звук и построить график преобразованного сигнала, получится кривая, показанная на рис. 1.1.

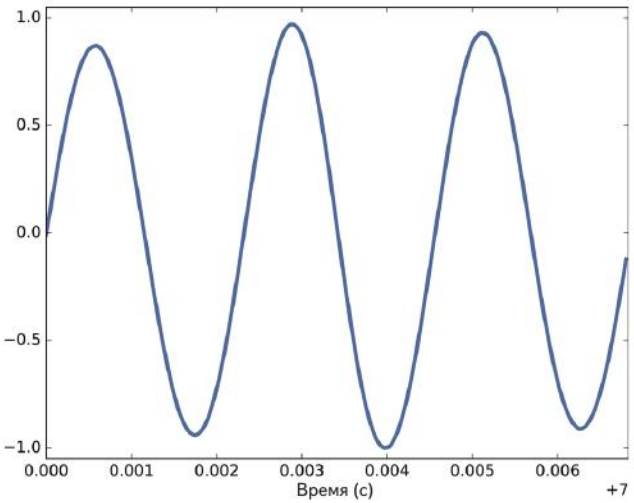

Рис. 1.1. Сегмент звукозаписи колокола

Этот сигнал напоминает *синусоиду*, то есть он похож на тригонометрическую функцию синус.

Видно, что это периодический сигнал. Интервал выбран так, чтобы показать три полных повтора, или *цикла*. Длительность каждого цикла, называемая *периодом*, составляет около 2,3 мс.

Частота сигнала – число циклов в секунду, она обратна периоду. За единицу частоты приняты циклы в секунду, или *герц*, сокращенно «Гц». (Строго говоря, число циклов – безразмерная величина, поэтому герц – это «циклов в секунду».)

Частота этого сигнала около 439 Гц – чуть ниже 440 Гц, стандартной высоты тона для оркестровой музыки. Музыкальное имя этой ноты – A (ля), или, точнее, A4. В американской системе нотации числовой суффикс указывает, из какой октавы нота. A4 – это A выше среднего C. A5 – на октаву выше. Подробнее см. `https://ru.wikipedia.org/wiki/Американская_система_нотации`.

Колебания зубцов *камертона* – это форма простого гармонического движения, поэтому сигнал похож на синусоиду. Большинство музыкальных инструментов производят периодические сигналы, но форма этих сигналов не синусоидальная. Например, на рис. 1.2 показан сегмент записи звука скрипки, на которой играют 3-ю часть струнного квинтета № 5 ми-мажор Боккерини.

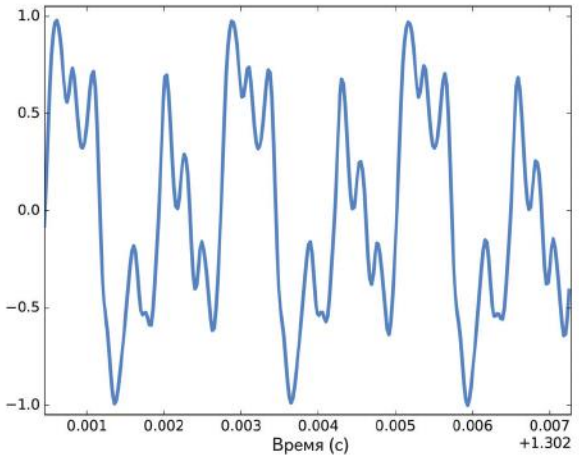

Рис. 1.2. Сегмент записи звука скрипки

Видно, что сигнал периодический, но форма сигнала заметно сложнее. В англоязычной литературе форму периодического сигнала называют *waveform*. В русском научно-техническом лексиконе такого понятия нет. Ведь сигнал – он и есть сигнал. И он может иметь самую разную форму.

Сигналы большинства музыкальных инструментов сложнее синусоиды. От формы сигнала зависит музыкальный *тембр*, определяющий восприятие качества звука. Люди обычно воспринимают сложные сигналы как богатые, теплые, более интересные, чем синусоидальные.

Разложение в спектр

Самая важная тема этой книги – *разложение в спектр*. Суть ее в том, что любой сигнал можно представить суммой синусоид с разными частотами.

Самая важная математическая идея в этой книге – *дискретное преобразование Фурье* (ДПФ). Оно берет сигнал и выдает его спектр. *Спектр* – это набор синусоид, составляющих сигнал.

А самый важный алгоритм в этой книге – *быстрое преобразование Фурье* (БПФ) – эффективнейший способ вычисления ДПФ.

Например, на рис. 1.3 показан спектр записи звука скрипки по рис. 1.2. Ось x – область частот, составляющих сигнал. Ось y показывает размахи, или *амплитуды*, составляющих спектра.

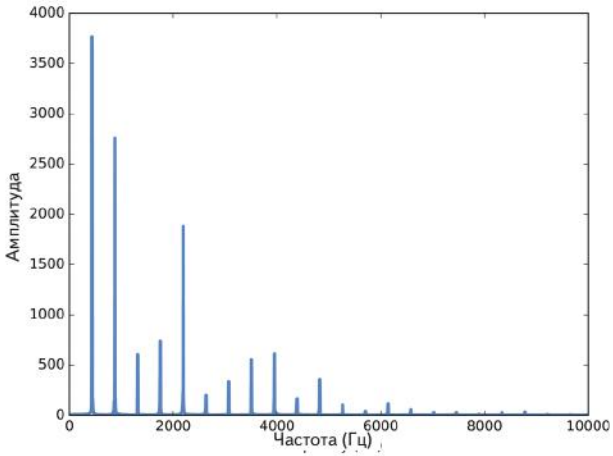

Рис. 1.3. Спектр сегмента записи звука скрипки

Компонента с самой низкой частотой называется *основной частотой*. Основная частота этого сигнала – около 440 Гц (на самом деле чуть ниже, или «площе»).

В данном сигнале у основной частоты самая большая амплитуда, так что это еще и *доминирующая частота*. Обычно воспринимаемая высота звука определяется основной частотой, даже если она и не доминирующая.

Другие пики в спектре находятся на частотах 880, 1320, 1760 и 2200, и это целые кратные основной. Эти компоненты называются *гармониками*, потому что они музыкально гармоничны с основной:

- 880 – частота A5, на одну октаву выше основной. *Октава* – это удвоение частоты;
- 1320 – приблизительно E6, которая выше A5 ровно на квинту. Про музыкальные интервалы и про чистую квинту можно прочесть здесь: `https://ru.wikipedia.org/wiki/Интервал_(музыка)`;
- 1760 – это A6, на две октавы выше основной;
- 2200 – примерно C#7, на большую терцию выше A6.

Эти гармоники – ноты аккорда A-мажор, хоть они и не все в одной октаве. Некоторые из них не совсем точны, поскольку ноты, составляющие «западную» музыку, были нормализованы для *равномерной темперации* (см. `https://ru.wikipedia.org/wiki/Равномерно_темперированный_строй`).

Зная гармоники и их амплитуды, сигнал можно восстановить сложением синусоид. Далее будет показано, как.

Сигналы

Автор написал Python-модуль, `thinkdsp.py`, содержащий классы и функции для работы с сигналами и спектрами. Он помещен в репозиторий этой книги (см. раздел «Работа с кодом» на стр. 9).

Для представления сигналов в `thinkdsp` есть класс, называемый `signal`. Это родительский класс для нескольких типов сигналов, включая `Sinusoid`, представляющий сигналы синус и косинус.

`thinkdsp` предоставляет функции для создания сигналов синус и косинус:

```
cos_sig = thinkdsp.CosSignal(freq=440, amp=1.0, offset=0)
sin_sig = thinkdsp.SinSignal(freq=880, amp=0.5, offset=0)
```

`freq` – частота в герцах, `amp` – амплитуда в относительных единицах, причем 1,0 определяется как наибольшая возможная записываемая или воспроизводимая амплитуда.

`offset` – это *фазовый сдвиг*, или просто *фаза*, в радианах. Фазовый сдвиг определяет, в каком месте периода начинается сигнал. Например, сигнал синус с параметром `offset = 0` начинается в sin 0, то есть в «0». При `offset = Pi/2` начало в sin $\pi/2$, то есть в «1».

У сигналов есть метод `__add__`, так что можно использовать оператор + при их сложении:

```
mix = sin_sig + cos_sig
```

Результатом будет `SumSignal`, представляющий собой сумму двух или более сигналов.

Обычно `signal` – это Python-представление математической функции. Большинство сигналов определены для всех значений t от минус бесконечности до бесконечности.

Более ничего с `signal` не сделать, если не начать его обработку. В этом контексте «обработка» означает последовательность моментов времени, `ts`, и получение соответствующего им значения сигнала, `ys`. Представлены `ts` и `ys` в виде массивов NumPy, инкапсулированных в объект, называемый `wave`.

`wave` – это сигнал, обрабатываемый в последовательности моментов времени. Каждый момент времени называется *кадром* (frame) – термин заимствован из кино и видео. Результат измерения называет-

ся *выборкой*, или *отсчетом*, хотя понятия «кадр» и «выборка» иногда взаимозаменяемы.

`signal` дает `make_wave`, возвращающий новый объект `wave`:

```
wave = mix.make_wave(duration=0.5, start=0, framerate=11025)
```

`duration` – длина `wave` в секундах.

`start` – время старта в секундах.

`framerate` – число (целое) кадров в секунду, а также число выборок в секунду.

11 025 кадров в секунду – одна из многих частот выборки, обычно используемых в звуковых файлах разных форматов, включая Waveform Audio File (WAV) и MP3.

В этом примере обрабатывается сигнал от $t = 0$ до $t = 0{,}5$ в 5513 кадрах с равным интервалом (поскольку 5513 – это половина от 11 025). Время между кадрами, или *шаг*, составляет 1/11 025 секунд (около 91 мкс).

`wave` поддерживает метод `plot`, использующий `pyplot`. Сигнал можно вывести на печать вот так:

```
wave.plot()
pyplot.show()
```

`pyplot` – это часть `matplotlib`; он включен во многие дистрибутивы Python, и его всегда можно доустановить.

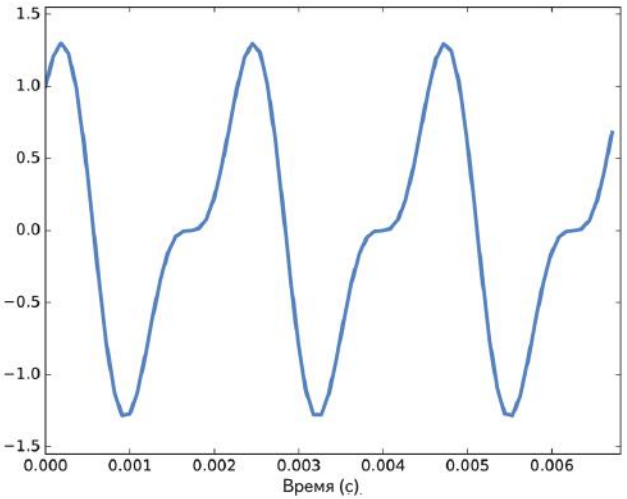

Рис. 1.4. Сегмент сигнала из смеси двух синусоид

При `freq` = 440 за 0,5 секунды пройдет 220 периодов, так что при печати этот участок будет представлен сплошной заливкой. Меньшее число периодов можно вывести, используя `segment` – он копирует некий сегмент `wave` и возвращает новый вариант:

```
period = mix.period
segment = wave.segment(start=0, duration=period*3)
```

`period` – это свойство `signal`; он возвращает период в секундах.

`start` и `duration` выражены в секундах. В этом примере из `mix` копируются первые три периода. Результат – объект `wave`.

На графике `segment` выглядит как на рис. 1.4. В этом сигнале две частотных компоненты; он сложнее сигнала камертона, но проще скрипичного.

Чтение и запись сигналов

`thinkdsp` предоставляет `read_wave`, читающий WAV-файл и возвращающий `wave`:

```
violin_wave = thinkdsp.read_wave('input.wav')
```

А `wave` поддерживает `write`, записывающий WAV-файл:

```
wave.write(filename='output.wav')
```

Прослушать `wave` можно на любом мультимедиа-проигрывателе, воспроизводящем WAV-файлы. На Unix-системах это aplay, простой и надежный, включенный во многие дистрибутивы Linux.

В `thinkdsp` также есть `play_wave`, запускающий мультимедийный проигрыватель как подпроцесс:

```
thinkdsp.play_wave(filename='output.wav', player='aplay')
```

По умолчанию он использует aplay, но можно указать и другой проигрыватель.

Спектры

`wave` поддерживает `make_spectrum`, возвращающий спектр – `spectrum`:

```
spectrum = wave.make_spectrum()
```

А `Spectrum` поддерживает `plot`:

```
spectrum.plot()
thinkplot.show()
```

`thinkplot` – написанный автором модуль, и это обертка для некоторых функций в `pyplot`. Он есть в репозитории этой книги (см. раздел «Работа с кодом» на стр. 9).

`Spectrum` предоставляет три метода, изменяющих спектр:

- `low_pass` применяет фильтр нижних частот (ФНЧ), то есть компоненты выше частоты среза ослабляются на некую величину.
- `high_pass` применяет фильтр верхних частот (ФВЧ), то есть ослабляются компоненты ниже частоты среза.
- `band_stop` применяет полосно-заграждающий фильтр (ФПЗ); он ослабляет компоненты в полосе частот между двумя частотами среза.

В этом примере все частоты выше 600 ослабляются на 99%:

```
spectrum.low_pass(cutoff=600, factor=0.01)
```

Фильтр НЧ удаляет «яркие» высокочастотные звуки, и результат звучит глуше, «темнее». Услышать это можно, если преобразовать `Spectrum` обратно в `wave` и воспроизвести:

```
wave = spectrum.make_wave()
wave.play('temp.wav')
```

Метод `play` записывает `wave` в файл, а затем воспроизводит его. В блокноте Jupyter можно использовать `make_audio`, создающий аудиовиджет для проигрывания звука.

Объекты Wave

В `thinkdsp.py` нет ничего сложного. Большинство функций в нем – лишь тонкие обертки на функциях из NumPy и SciPy.

Основные классы в `thinkdsp` – это `signal`, `wave` и `Spectrum`. Имея `signal`, можно сделать `wave`. Имея `wave`, можно сделать `Spectrum`, и наоборот. Эти соотношения показаны на рис. 1.5.

Рис. 1.5. Соотношения классов в `thinkdsp`

У объекта `wave` есть три атрибута: `ys` – NumPy-массив, содержащий значения сигнала; `ts` – массив моментов выборки или преобразования сигнала; `framerate` – число выборок в единицу времени. За единицу времени обычно принимаются секунды, хотя допустимы и другие варианты. В одном из примеров это будут дни.

`wave` также имеет три свойства, доступные только для чтения: `start`, `end` и `duration`. Если изменить `ts`, эти свойства изменятся соответственно.

Изменить сигнал можно, явно изменив `ts` и `ys`. Например:

```
wave.ys *= 2
wave.ts += 1
```

Первая строка вдвое увеличивает размах, делая сигнал громче. Вторая строка сдвигает сигнал во времени, замедляя начало на 1 секунду.

Но `wave` дает методы, выполняющие множество общих операций. Например, можно записать эти же два преобразования так:

```
wave.scale(2) wave.shift(1)
```

Эти и другие методы описаны на веб-странице `http://greenteapress.com/thinkdsp.html`.

Объекты Signal

`signal` – это родительский класс, предоставляющий функции, общие для всех типов сигналов, таких как `make_wave`. Дочерние классы наследуют эти методы и дают `evaluate`, оценивающий сигнал в заданные моменты времени.

Например `Sinusoid` – это дочерний класс `signal` с таким определением:

```
class Sinusoid(signal):

    def __init__(self, freq=440, amp=1.0, offset=0, func=np.sin):
        signal. __init__ (self)
        Self.freq = freq
        self.amp = amp
        self.offset = offset
        self.func = func
```

Параметрами `__init__` будут:

`freq`
: Частота в циклах в секунду, или герцах.

`amp`
: Амплитуда. Единицы амплитуды произвольны: обычно их выбирают так, что 1,0 соответствует максимуму входного уровня с микрофона или максимальному уровню на выходе.

`offset`
: Указывает, в какой части своего периода сигнал начинается; `offset` указывается в радианах.

`func`
: Функция Python, используемая для оценки сигнала в определенный момент времени. Обычно это `np.sin` или `np.cos`, то есть синусоидальный или косинусоидальный сигнал.

Как и многие init-методы, этот просто «откладывает» параметры «на потом».

`signal` дает `make_wave` следующим образом:

```
def make_wave(self, duration=1, start=0, framerate=11025):
    n = round(duration * framerate)
    ts = start + np.arange(n) / framerate
    ys = self.evaluate(ts)
    return wave(ys, ts, framerate=framerate)
```

`Start` и `duration` – время начала и длительность в секундах.

`framerate` – число кадров (выборок) в секунду.

`n` – число выборок, а `ts` – массив времен выборки NumPy.

Для расчета `ys` сигнал `make_wave` вызывает функцию `evaluate`, получаемую из `Sinusoid`:

```
def evaluate(self, ts):
     phases = PI2 * Self.freq * ts + self.offset
     ys = self.amp * self.func(phases)
     return ys
```

Раскрутим ее до одного шага по времени:

1. `Self.freq` – частота в циклах в секунду, а каждый элемент `ts` дает время в секундах, поэтому их произведение будет числом циклов от начала.
2. `PI2` – константа, хранящая 2π. Умножение на `PI2` преобразует циклы в фазу. Можно рассматривать фазу как «число радианов от начала». Каждый цикл равен 2π радиан.
3. `Self.offset` – это фаза в момент $t = 0$. На графике это выглядит как сдвиг сигнала по оси времени влево или вправо.
4. Если `self.func` есть `np.si` или `np.cos`, то результатом будет значение в интервале между –1 и + 1.

5. Умножение на `self.amp` дает сигнал, меняющийся от `-self.amp` до `+self.amp`.

Математически `evaluate` записывается следующим образом:

$$y = A \cos(2\pi ft + \phi_0)$$

Здесь A – амплитуда, f – частота, t – время и ϕ_0 – начальная фаза. Может показаться, что кода слишком много для вычисления одного простого выражения, но далее будет видно, что на этом коде основана обработка всех видов сигналов, а не только синусоиды.

Упражнения

Для нижеследующих упражнений полезно скачать код для этой книги, следуя инструкциям в разделе «Работа с кодом» на стр. 9. Решения этих упражнений находятся в блокноте `chap01soln.ipynb`.

Упражнение 1.1

Для Jupyter надо загрузить `chap01.ipynb`, прочитать пояснения и запустить примеры. Этот блокнот можно также просмотреть на вебстранице `http://tinyurl.com/thinkdsp01`.

Упражнение 1.2

Скачайте с сайта `http://freesound.org` образец звука, включающий музыку, речь или иные звуки, имеющие четко выраженную высоту. Выделите примерно полусекундный сегмент, в котором высота постоянна. Вычислите и распечатайте спектр выбранного сегмента. Как связаны тембр звука и гармоническая структура, видимая в спектре?

Используйте `high_pass`, `low_pass` и `band_stop` для фильтрации тех или иных гармоник. Затем преобразуйте спектры обратно в сигнал и прослушайте его. Как звук соотносится с изменениями, сделанными в спектре?

Упражнение 1.3

Создайте сложный сигнал из объектов `SinSignal` и `CosSignal`, суммируя их. Обработайте сигнал для получения `wave` и прослушайте его. Вычислите `Spectrum` и распечатайте. Что произойдет при добавлении частотных компонент, не кратных основным?

Упражнение 1.4

Напишите функцию `stretch`, берущую `wave` и коэффициент изменения. Она должна ускорять или замедлять сигнал изменением `ts` и `framerate`. Подсказка: должно получиться всего две строки кода.

Глава 2. Гармоники

В этой главе представлено несколько новых сигналов; рассмотрены их спектры и гармоническая структура – набор синусоид, из которых они состоят.

Рассмотрено также одно из наиболее важных явлений в цифровой обработке сигналов – *биения* (aliasing, *алиасинг*). Кроме того, подробнее объяснена работа класса `Spectrum`.

Код для этой главы находится в репозитории книги, в блокноте `chap02.ipynb` (см. раздел «Работа с кодом» на стр. 9). Код можно просмотреть и на веб-странице `http://tinyurl.com/thinkdsp02`.

Треугольный сигнал

У синусоиды только одна частотная компонента, поэтому в ее спектре лишь один пик. Более сложные сигналы, вроде записи звука скрипки, показанной на рис. 1.2, дают ДПФ со множеством пиков. В этом разделе исследуется связь между формой сигналов и их спектрами.

Начнем с треугольного сигнала, похожего на спрямленную версию синусоиды. На рис. 2.1 показан треугольный сигнал частотой 200 Гц.

Генерацию треугольного сигнала начнем с `thinkdsp.TriangleSignal`:

```
class TriangleSignal(Sinusoid):

    def evaluate(self, ts):
        cycles = Self.freq * ts + self.offset / PI2
        frac, _ = np.modf(cycles)
        ys = np.abs(frac - 0.5)
        ys = normalize(unbias(ys), self.amp)
        return ys
```

`TriangleSignal` наследует `__init__` от `Sinusoid`, поэтому он принимает те же аргументы: `freq`, `amp` и `offset`.

Разница лишь в `evaluate`. Как и ранее, `ts` обозначает последовательность моментов времени, в которых оценивается сигнал.

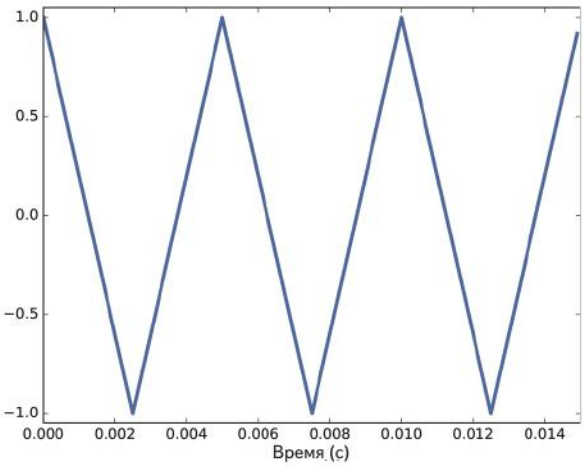

Рис. 2.1. Сегмент треугольного сигнала частотой 200 Гц

Есть много способов создания треугольного сигнала. Детали не важны, но вот как работает `evaluate`:

1. `cycles` – число циклов со времени начала. `np.modf` разделяет число циклов на дробную часть `frac` и целую часть, которая игнорируется[1].
2. `frac` – это последовательность, растущая в пределах от 0 до 1 с заданной частотой. Вычитание 0,5 меняет пределы на –0,5 и 0,5. Абсолютное значение дает сигнал, меняющийся от 0,5 до 0.
3. `unbias` смещает сигнал так, что он центрируется по 0; затем `normalize` масштабирует его до заданной амплитуды `amp`.

Вот код, генерирующий сигнал, показанный на рис. 2.1:

```
signal = thinkdsp.TriangleSignal(200)
signal.plot()
```

Теперь возьмем `signal`, сделаем `wave` и рассчитаем его `Spectrum`:

```
wave = signal.make_wave(duration=0.5, framerate=10000)
spectrum = wave.make_spectrum()
spectrum.plot()
```

На рис. 2.2 показаны два вида результата; вид справа отмасштабирован – четче видны *гармоники*. Как и ожидалось, самый высокий пик – на основной частоте, 200 Гц, но есть дополнительные пики гармоник на частотах, целых кратных 200.

[1] Знак подчеркивания в имени переменной – это соглашение: «Не намерен использовать это значение».

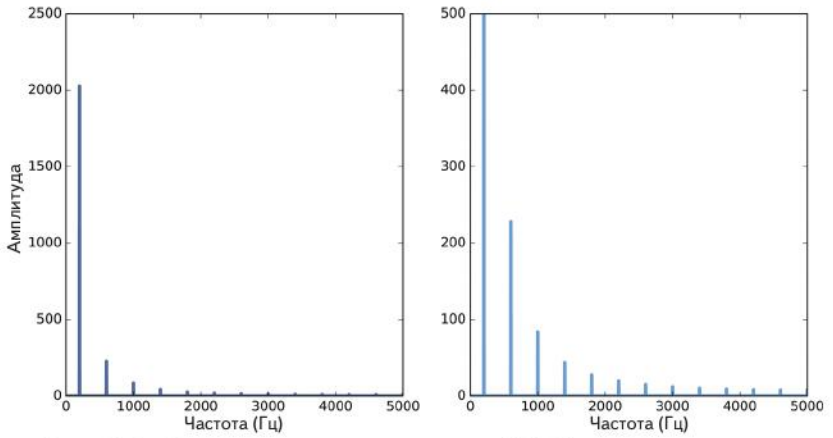

Рис. 2.2. Спектр треугольного сигнала 200 Гц, с двумя шкалами по вертикали. У правого обрезана основная, чтобы четче показать гармоники

Но сюрприз в том, что нет четных пиков: 400, 800 и т. д. У треугольного сигнала есть только нечетные гармоники, кратные основной частоте (в этом примере – 600, 1000, 1400 и т. д.).

Еще одна особенность этого спектра – взаимосвязь между амплитудой и частотой гармоник. Их амплитуда спадает пропорционально квадрату частоты. Например, отношение частот первых двух гармоник (200 и 600 Гц) – 3, а соотношение амплитуд будет примерно 9. Соотношение частот следующих двух гармоник (600 и 1000 Гц) – 1,7, а соотношение амплитуд будет примерно $1{,}72^2 = 2{,}9$. Эта взаимосвязь называется *гармонической структурой*.

Прямоугольный сигнал

`thinkdsp` также предоставляет `SquareSignal`, то есть прямоугольный сигнал. Вот определение класса:

```
class SquareSignal(Sinusoid):

    def evaluate(self, ts):
        cycles = Self.freq * ts + self.offset / PI2
        frac, _ = np.modf(cycles)
        ys = self.amp * np.sign(unbias(frac))
        return ys
```

Как и `TriangleSignal`, `SquareSignal` наследует `__init__` от `Sinusoid` и принимает те же параметры.

И метод `evaluate` похож. Вновь `cycles` – число циклов со времени старта, `frac` – дробная часть, растущая от 0 до 1 за каждый период.

`unbias` сдвигает `frac` так, что он растет от –0,5 до 0,5, причем `np.sign` сопоставляет отрицательные величины с – 1, а положительные – с +1. Умножение на `amp` дает прямоугольный сигнал, меняющийся от `–amp` до `+amp`.

На рис. 2.3 показаны три периода прямоугольного сигнала частотой 100 Гц, а на рис. 2.4 представлен его спектр.

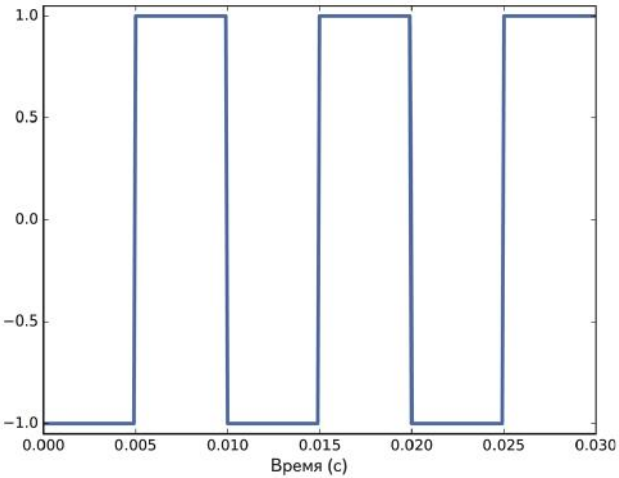

Рис. 2.3. Сегмент прямоугольного сигнала 100 Гц

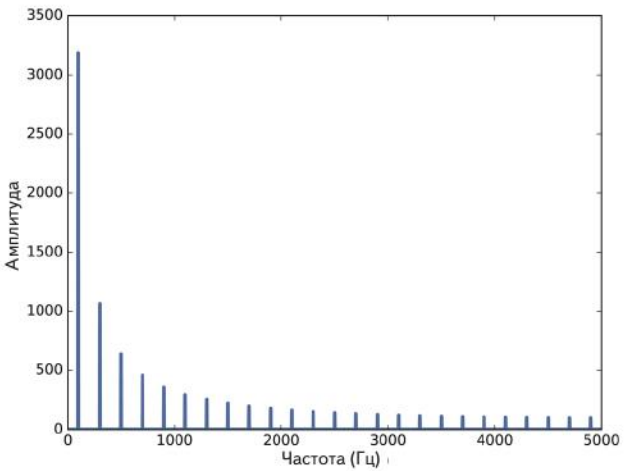

Рис. 2.4. Спектр прямоугольного сигнала 100 Гц

Как и треугольный сигнал, прямоугольный содержит только нечетные гармоники, поэтому пики приходятся на 300, 500 и 700 Гц и т. д. Но амплитуда гармоник спадает медленнее – точнее, пропорционально частоте (а не квадрату частоты).

В упражнениях в конце этой главы можно исследовать другие сигналы и гармонические структуры.

Биения (алиасинг)

Примеры для предыдущего раздела выбраны так, чтобы все было понятно. А вот этот раздел вас может и озадачить.

Для начала рассмотрим рис. 2.5.

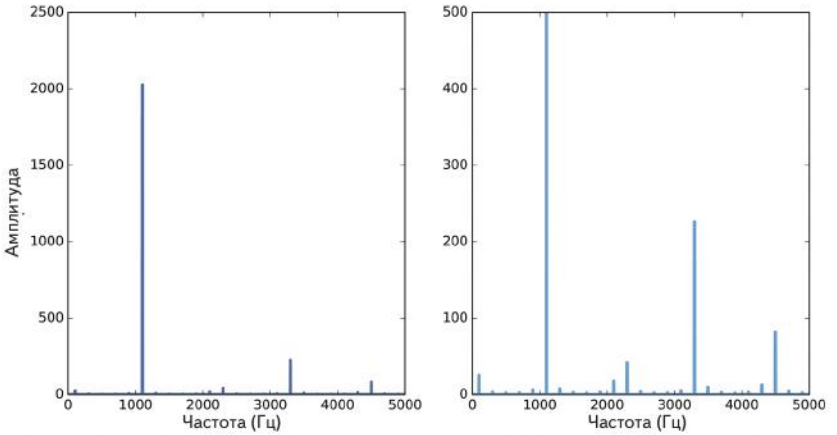

Рис. 2.5. Спектр треугольного сигнала 1100 Гц при 10 000 выборок в секунду. Вид справа масштабирован для выявления гармоник

Гармоники сигнала должны быть 3300, 5500, 7700 и 9900 Гц. На рисунке, как и ожидалось, есть пики на 1100 и 3300 Гц, но третий пик – на 4500, а не на 5500 Гц. Четвертый пик находится на 2300, а не на 7700 Гц. И если посмотреть внимательно, то пик, который должен быть на 9900, на самом деле приходится на 100 Гц. Что произошло?

Проблема в том, что при взятии выборок из сигнала в дискретные моменты времени теряется информация о том, что было между выборками. Для низкочастотных компонент это не проблема, потому что выборок за период достаточно много.

Но если сигнал 5000 Гц при 10 000 кадров в секунду, то у нас будет только две выборки за период. Этого достаточно, а вот если частота сигнала еще выше, то возникнут проблемы.

Чтобы понять почему, сгенерируем косинусоидальные сигналы 4500 и 5500 Гц, а скорость выборки возьмем 10 000 кадров в секунду:

```
framerate = 10000

signal = thinkdsp.CosSignal(4500)
duration = signal.period*5
segment = signal.make_wave(duration, framerate=framerate)
segment.plot()

signal = thinkdsp.CosSignal(5500)
segment = signal.make_wave(duration, framerate=framerate)
segment.plot()
```

Результат показан на рис. 2.6. Оба `signal` выведены тонкими серыми линиями, а выборки – вертикальными линиями, чтобы упростить сравнение двух `wave`. Тут-то и обнаруживается: `signal` разные, а `wave` – одинаковые!

Выборки с частотой 10 000 кадров в секунду из сигнала 5500 Гц неотличимы от выборок из сигнала 4500 Гц. По этой же причине сигнал 7700 Гц неотличим от сигнала 2300 Гц, а сигнал 9900 Гц – от сигнала 100 Гц.

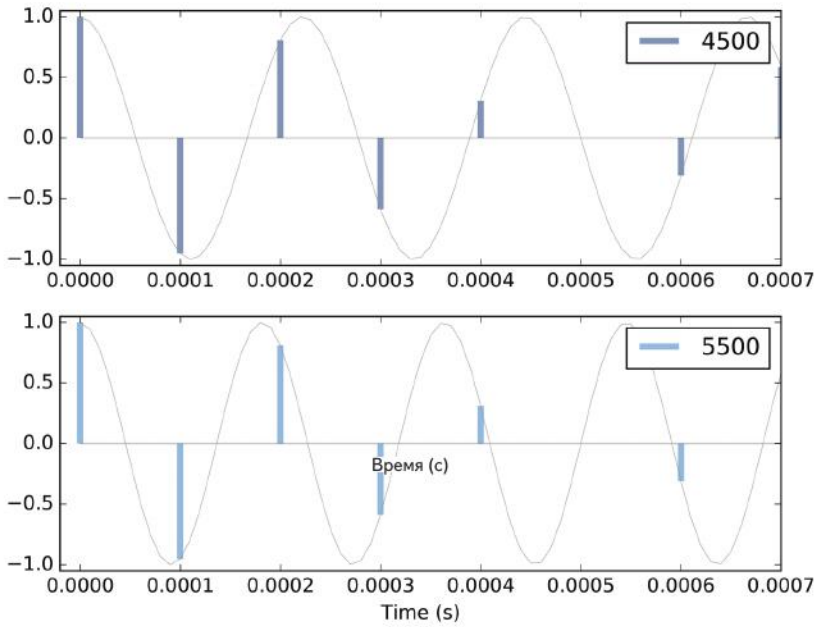

Рис. 2.6. Косинус-сигналы 4500 и 5500 Гц, выборка 10 000 кадров в секунду. Выборки одинаковы при разных сигналах

Этот эффект называется *биениями*, или *алиасингом* (aliasing), потому что выборки из сигнала с высокой частотой кажутся выборками из сигнала с низкой.

В этом примере самая высокая частота, которую еще можно обработать, составит 5000 Гц (половина частоты дискретизации). Частоты выше 5000 Гц «заворачиваются» вокруг 5000 Гц, так что этот порог иногда называют *частотой заворота*. Встречается также термин *частота Найквиста*; см. `http://ru.wikipedia.org/wiki/Частота_Найквиста`.

Заворот повторяется, когда частота биений оказывается меньше нуля. Например, пятая[1] гармоника треугольного сигнала 1100 Гц составит 12 100 Гц. Заворот на 5000 Гц даст –2100 Гц, но на 0 Гц будет новый заворот, и он даст 2100 Гц. В самом деле, на рис. 2.4 заметен небольшой пик на 2100 Гц, а также следующий, на 4300 Гц.

Вычисление спектра

Метод `make_spectrum` у объекта `wave` уже встречался. Применим его еще раз (некоторые детали рассмотрены ниже):

```
from np.fft import rfft, rfftfreq

# class wave:
    def make_spectrum(self):
        n = len(self.ys)
        d = 1 / self.framerate

        hs = rfft(self.ys)
        fs = rfftfreq(n, d)

        return Spectrum(hs, fs, self.framerate)
```

Параметр `self` – это объект `wave`; `n` – количество выборок в сигнале, а `d` – обратная частоте кадров величина, то есть время между выборками.

`np.fft` – это модуль NumPy, дающий функции для быстрого преобразования Фурье (БПФ) – эффективный алгоритм вычисления дискретного преобразования Фурье (ДПФ).

`make_spectrum` использует `rfft`, то есть «действительное БПФ», поскольку `wave` содержит действительные числа, а не комплексные. Ниже рассмотрено полное БПФ, обрабатывающее комплексные сигналы (см. раздел «ДПФ реальных сигналов» на стр. 96). Результат `rfft`, обозначаемый `hs`, – это NumPy-массив комплексных чисел,

[1] По счету, то есть одиннадцатая кратная основной. – *Прим. ред.*

представляющих амплитуду и фазу каждой частотной компоненты в сигнале.

Результат `rfftfreq`, обозначаемый `fs`, – это массив, содержащий частоты, соответствующие `hs`.

Разберемся с содержимым `hs` – рассмотрим два способа представления комплексных чисел:

- комплексное число – сумма действительной и мнимой частей, часто в виде $x + iy$, где i – мнимая единица, $\sqrt{-1}$. x и y следует понимать как координаты в декартовой системе;
- комплексное число – это также произведение модуля и комплексной экспоненты – $Ae^{i\phi}$, где A – *модуль*, а ϕ – *угол* в радианах, называемый также *аргументом*. A и ϕ следует понимать как полярные координаты.

Каждое значение в `hs` соответствует частотной компоненте: размах пропорционален амплитуде соответствующей компоненты; а угол – это фаза.

Класс `Spectrum` содержит два свойства только для чтения, `amps` и `angles`, возвращающие NumPy-массивы, представляющие амплитуды и углы `hs`. На графиках объект `Spectrum` обычно представлен как `amps` относительно `fs`. Но иногда полезно отобразить и `angles` относительно `fs`.

Можно получить действительные и мнимые части `hs`, но обычно они не нужны. Проще рассматривать ДПФ как вектор амплитуд и фаз, записанный в виде комплексных чисел.

Для изменения `Spectrum` можно обращаться прямо к `hs`. Например:

```
spectrum.hs *= 2
spectrum.hs[spectrum.fs > cutoff] = 0
```

Первая строка умножает элементы `hs` на 2, удваивая амплитуды всех компонент. Вторая строка обнуляет те элементы `hs`, частота которых превышает некий порог.

Но `Spectrum` также дает методы для таких вот операций:

```
spectrum.scale(2) spectrum.low_pass(cutoff)
```

Документацию по этим и другим методам можно найти на вебстранице `http://greenteapress.com/thinkdsp.html`.

На этом этапе уже должно быть понятно, как работают классы `signal`, `wave`, `Spectrum`. А вот разговор о том, как работает быстрое преобразование Фурье, растянется на несколько следующих глав.

Упражнения

Решения этих упражнений находятся в блокноте `chap02soln.ipynb`.

Упражнение 2.1

Для Jupyter надо загрузить `chap02.ipynb` и примеры. Этот блокнот можно также просмотреть на веб-странице `http://tinyurl.com/thinkdsp02`.

Упражнение 2.2

Пилообразный сигнал линейно нарастает от – 1 до 1, а затем резко падает до –1 и повторяется. См. `http://en.wikipedia.org/wiki/Sawtooth_wave` (на англ. яз.).

Напишите класс, называемый `SawtoothSignal`, расширяющий `signal` и предоставляющий `evaluate` для оценки пилообразного сигнала.

Вычислите спектр пилообразного сигнала. Как соотносится его гармоническая структура с треугольным и прямоугольным сигналами?

Упражнение 2.3

Создайте прямоугольный сигнал 1100 Гц и вычислите `wave` с выборками 10 000 кадров в секунду. Постройте спектр и убедитесь, что большинство гармоник «завернуты» из-за биений. Слышны ли последствия этого при проигрывании?

Упражнение 2.4

Возьмите объект `Spectrum` и распечатайте несколько первых значений `spectrum.fs`. Убедитесь, что они начинаются с нуля, то есть `Spectrum.hs[0]` – амплитуда компоненты с частотой 0. Но что это значит?

Проведите такой эксперимент:

1. Создайте треугольный сигнал с частотой 440 Гц и `wave` длительностью 0,01 секунд. Распечатайте сигнал.
2. Создайте объект `Spectrum` и распечатайте `Spectrum.hs[0]`. Каковы амплитуда и фаза этого компонента?
3. Установите `Spectrum.hs[0] = 100`. Как эта операция повлияет на сигнал? Подсказка: `Spectrum` дает метод, называемый `make_wave`, вычисляющий `wave`, соответствующий `Spectrum`.

Упражнение 2.5

Напишите функцию, принимающую `Spectrum` как параметр и изменяющую его делением каждого элемента `hs` на соответствующую

частоту из `fs`. Подсказка: поскольку деление на ноль не определено, надо задать `Spectrum.hs[0] = 0`.

Проверьте эту функцию, используя прямоугольный, треугольный или пилообразный сигналы:

1. Вычислите `Spectrum` и распечатайте его.
2. Измените `Spectrum`, вновь используя свою функцию, и распечатайте его.
3. Используйте `Spectrum.make_wave`, чтобы сделать `wave` из измененного `Spectrum`, и прослушайте его. Как эта операция повлияла на сигнал?

Упражнение 2.6

У треугольных и прямоугольных сигналов есть только нечетные гармоники; в пилообразном сигнале есть и четные, и нечетные гармоники. Гармоники прямоугольных и пилообразных сигналов уменьшаются пропорционально $1/f$; гармоники треугольных сигналов – пропорционально $1/f^2$. Можно ли найти сигнал, состоящий из четных и нечетных гармоник, спадающих пропорционально $1/f^2$?

Подсказка: для этого есть два способа. Можно собрать желаемый сигнал из синусоид, а можно взять сигнал со спектром, похожим на необходимый, и изменять его параметры.

Глава 3. Апериодические сигналы

До сих пор были рассмотрены сигналы периодические, непрерывно повторяющиеся. И их частотные компоненты не изменяются во времени. В этой главе рассмотрены сигналы *апериодические*, частотные компоненты которых изменяются во времени, – другими словами, практически все звуковые сигналы.

В этой главе также представлены спектрограммы – распространенный способ визуализации апериодических сигналов.

Код для главы 3 находится в репозитории этой книги, в блокноте `chap03.ipynb` (см. раздел «Работа с кодом» на стр. 9). Также код можно просмотреть на веб-странице `http://tinyurl.com/thinkdsp03`.

Линейный чирп

Рассмотрим *чирп*, сигнал с переменной частотой. `thinkdsp` дает `signal`, называемый `Chirp`, который создает синусоиду с частотой, линейно изменяющейся в некотором диапазоне.

Вот пример с перестройкой от 220 до 880 Гц, то есть на две октавы – от A3 до A5:

```
signal = thinkdsp.Chirp(start=220, end=880)
wave = signal.make_wave()
```

На рис. 3.1 показаны сегменты периода этого сигнала – в начале, в середине и в конце. Очевидно, что частота увеличивается.

Вначале рассмотрим, как устроен Chirp. Вот определение класса:

```
class Chirp(signal):

    def __init__ (self, start=440, end=880, amp=1.0):
        self.start = start
        self.end = end
        self.amp = amp
```

`start` и `end` – частоты в герцах, в начале и в конце чирпа; `amp` – амплитуда.

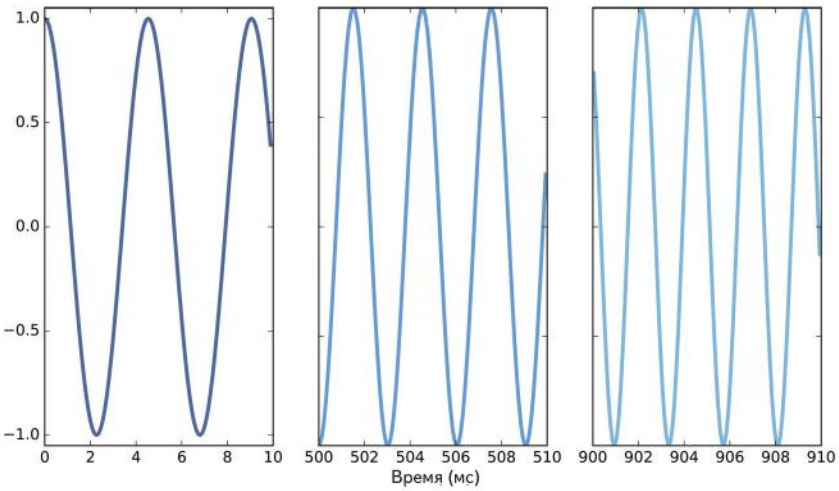

Рис. 3.1. Чирп-сигнал в начале, в середине и в конце

Вот функция, обрабатывающая сигнал:

```
def evaluate(self, ts):
    freqs = np.linspace(self.start, self.end, len(ts)-1)
    return self._evaluate(ts, freqs)
```

`ts` – это последовательность моментов времени обработки сигнала; для упрощения предполагается, что они равномерны.

Если длина `ts` есть `n`, его можно представить как последовательность из $n-1$ интервалов времени. Для вычисления частоты на каждом интервале используется `np.linspace`, возвращающая NumPy-массив из $n-1$ значений между `start` и `end`.

`_evaluate` – частный метод, производящий остальные расчеты[1]:

```
def _evaluate(self, ts, freqs):
    dts = np.diff(ts)
    dphis = PI2 * freqs * dts
    phases = np.cumsum(dphis)
    phases = np.insert(phases, 0, 0)
    ys = self.amp * np.cos(phases)
    return ys
```

`np.diff` вычисляет разницу между соседними элементами `ts`, возвращая длину каждого интервала в секундах. Если элементы `ts` равноотстоящие, все значения `dts` одинаковы.

[1] Если имя метода начинается с символа подчеркивания, это означает, что он «частный», то есть не является частью API, который должен использоваться вне определения класса.

Далее надо выяснить, насколько меняется фаза за каждый интервал. В разделе «Объекты Signal» на стр. 22 показано, что при постоянной частоте фаза ϕ линейно растет с течением времени:

$$\phi = 2\pi ft$$

Если частота – функция времени, изменение фазы за дискрет времени Δt составит:

$$\Delta\phi = 2\pi f(t)\Delta t$$

В Python, поскольку `freqs` содержит $f(t)$, а `dts` – интервалы времени, можно записать:

```
dphis = PI2 * freqs * dts
```

Теперь, поскольку `dphis` содержит изменения фазы, можно получить полную фазу на каждом отрезке времени, суммируя изменения:

```
phases = np.cumsum(dphis)
phases = np.insert(phases, 0, 0)
```

`np.cumsum` вычисляет нарастающую сумму, но она начинается не с 0. Поэтому `np.insert` добавляет этот 0 в начало.

Результат – массив NumPy, где `i`-ый элемент содержит сумму из первых `i` значений `dphis`, то есть полную фазу в конце `i`-го интервала. Наконец, `np.cos` вычисляет амплитуду сигнала в зависимости от фазы (фаза выражается в радианах).

Вспомнив матанализ, можно заметить, что предел при устремлении Δt к нулю будет:

$$d\phi = 2\pi f(t)\, dt$$

Деление на dt дает:

$$d\phi/dt = 2\pi f(t)$$

Другими словами, частота есть производная от фазы. И наоборот, фаза – это интеграл от частоты. То есть `cumsum`, примененный для перехода от частоты к фазе, аппроксимирует интегрирование.

Экспоненциальный чирп

Слушая этот чирп, можно заметить, что высота звука сначала резко нарастает, а затем рост замедляется. Чирп занимает две октавы, но первую он пролетает всего за 2/3 s, а вторую – в два раза медленнее.

Причина в том, что восприятие высоты звука зависит от логарифма частоты. В результате слышимый между двумя нотами интервал зависит от соотношения их частот, а не от их разницы. *Интервал* –

музыкальный термин для воспринимаемой разницы между двумя тонами.

Например, *октава* – интервал, где соотношение частот двух тонов равно 2. Так, интервал от 220 до 440 Гц – одна октава, и интервал от 440 до 880 Гц – также одна октава. Разница по частоте больше, а соотношение одинаковое.

В результате, если частота увеличивается линейно, как у линейного чирпа, воспринимаемая высота звука растет логарифмически.

Если нужно, чтобы воспринимаемая высота звука росла линейно, частота должна расти по экспоненте. Сигнал такого типа называется *экспоненциальным чирпом*.

Вот определение класса `ExpoChirp`:

```
class ExpoChirp(Chirp):

    def evaluate(self, ts):
        start, end = np.log10(self.start), np.log10(self.end)
        freqs = np.logspace(start, end, len(ts)-1)
        return self._evaluate(ts, freqs)
```

В отличие от `np.linspace`, эта версия `evaluate` использует `np.logspace`, создающий набор равноотстоящих логарифмов частот, то есть частота растет экспоненциально.

Все остальное повторяет `Chirp`. Вот код для его получения:

```
signal = thinkdsp.ExpoChirp(start=220, end=880)
wave = signal.make_wave(duration=1)
```

Примеры можно прослушать в `chap03.ipynb` и сравнить линейные чирпы с экспоненциальными.

Спектр чирпа

Выясним, что получится при вычислении спектра чирпа. Вот пример, создающий односекундный однооктавный чирп и его спектр:

```
signal = thinkdsp.Chirp(start=220, end=440)
wave = signal.make_wave(duration=1)
spectrum = wave.make_spectrum()
```

На рис. 3.2 показан результат. В спектре есть компоненты на каждой частоте от 220 до 440 Гц с вариациями размаха, похожими на око Саурона (см. `http://en.wikipedia.org/wiki/Sauron`).

Спектр – практически плоский от 220 до 440 Гц, а это указывает на то, что в данном диапазоне равное изменение частоты сигнала

занимает равное время. Исходя из рисунка можно догадаться, как выглядит спектр экспоненциального чирпа.

В этом спектре видны особенности структуры сигнала, но соотношение между частотой и временем скрыто. Например, глядя на спектр, нельзя сказать, меняется ли частота вверх, вниз или в обоих направлениях.

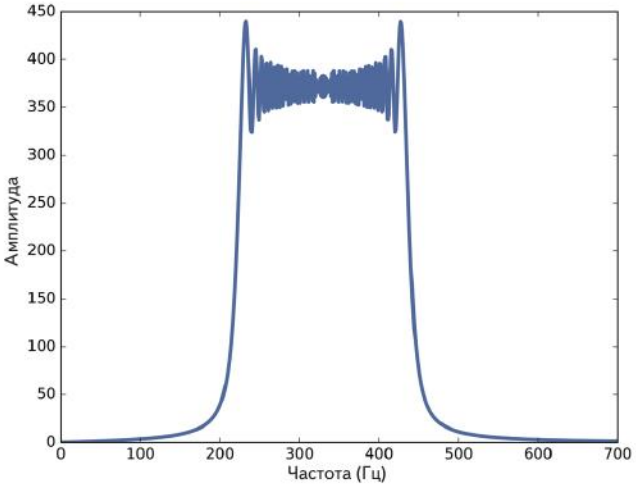

Рис. 3.2. Спектр односекундного однооктавного чирпа

Спектрограмма

Чтобы проявить связь между частотой и временем, можно разбить чирп на сегменты и построить спектр каждого сегмента. Результат называется *кратковременным преобразованием Фурье* (КВПФ).

Есть несколько способов визуализации КВПФ, но наиболее распространены *спектрограммы*, у которых на оси x время, а на оси y – частоты. Каждый столбец на спектрограмме показывает спектр короткого сегмента, а для представления амплитуд используются цвет или оттенки серого.

В качестве примера вычислим спектрограмму такого чирпа:

```
signal = thinkdsp.Chirp(start=220, end=440)
wave = signal.make_wave(duration=1, framerate=11025)
```

`wave` дает `make_spectrogram`, возвращающий объект `Spectrogram`:

```
spectrogram = wave.make_spectrogram(seg_length=512)
spectrogram.plot(high=700)
```

`seg_length` – количество выборок в каждом сегменте. Выберем 512, поскольку БПФ наиболее эффективен при числе выборок, кратном степени 2.

На рис. 3.3 показан результат. По оси x – время от 0 до 1 секунды. По оси y – частота от 0 до 700 Гц. Верхняя часть спектрограммы ограниченна, так как полный диапазон достигает 5512,5 Гц – половины частоты выборки.

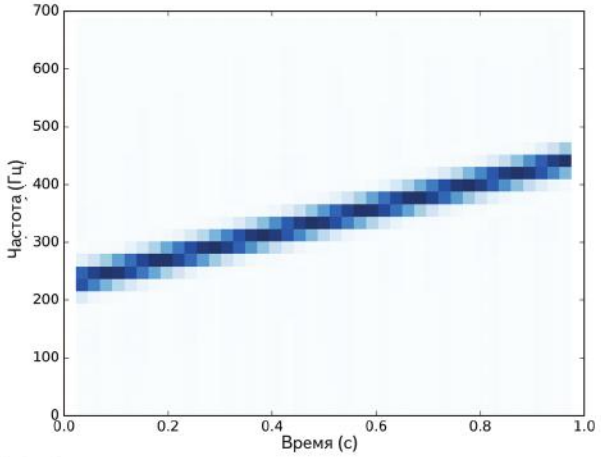

Рис. 3.3. Спектрограмма односекундного одноктавного чирпа

По спектрограмме четко видно, что с течением времени частота увеличивается линейно. Однако обратите внимание, что пик в каждом столбце размыт на две-три клетки. Это размытие отражает ограниченное разрешение спектрограммы.

Предел Габора

Разрешение по времени спектрограммы зависит от длительности сегментов, соответствующих ширине ячеек спектрограммы. Каждый сегмент состоит из 512 кадров, а всего их – 11 025 за секунду, длительность каждого сегмента составляет около 0,046 секунд.

Разрешение по частоте – это частотный интервал между элементами спектра с одинаковой высотой ячеек. При 512 кадрах получим 256 частотных компонент в диапазоне от 0 до 5512,5 Гц, а интервал между компонентами будет 21,6 Гц.

В общем представлении, если n – длина сегмента, спектр содержит $n/2$ компонент. Если частота кадров – r, максимальная частота спек-

тра будет $r/2$. То есть разрешение по времени будет n/r, а разрешение по частоте составит:

$$r/2 \ / \ n/2$$

что дает r/n.

В идеале при быстрых изменениях частоты нужно хорошее разрешение по времени. А при малых изменениях частоты нужно хорошее разрешение по частоте. Но и то и другое вместе получить нельзя. Заметим, что разрешение по времени n/r обратно разрешению по частоте r/n, так что если одно меньше, то другое больше.

Например, если удвоить длину сегмента, то вдвое улучшится разрешение по частоте (что хорошо), но во столько же раз ухудшится разрешение по времени (что плохо). Повышение частоты кадров также не поможет: будет больше выборок, но увеличится и диапазон частот.

Такой компромисс называют *пределом Габора*, и это фундаментальное ограничение данного типа частотно-временного анализа.

Утечка

Ниже рассматривается работа `make_spectrogram`; работа его основана на окнах; но прежде чем их изучать, разберемся с утечкой – именно из-за нее нужны окна.

Дискретное преобразование Фурье (ДПФ), используемое для вычисления `Spectrum`, обрабатывает сигналы, предполагая их периодичность; то есть конечный сегмент считается полным периодом бесконечного, непрерывно повторяющегося сигнала. На практике это предположение часто ложно, и отсюда возникают проблемы.

Одна из распространенных проблем – разрыв между началом и концом сегмента. При ДПФ предполагается, что сигнал периодический, поэтому начало и конец сегмента неявно соединяются в петлю. Если конец не совпадет с началом, то из-за разрыва в сегменте появятся дополнительные частотные компоненты, которых нет в сигнале.

Рассмотрим в качестве примера синусоидальный сигнал, содержащий только одну компоненту с частотой 440 Гц:

```
signal = thinkdsp.SinSignal(freq=440)
```

Если в сегменте будет целое число периодов, конец сегмента плавно соединится с началом, и ДПФ отработает хорошо:

```
duration = signal.period * 30
wave = signal.make_wave(duration)
spectrum = wave.make_spectrum()
```

На рис. 3.4 (слева) показан результат. Как и ожидалось, есть один пик на 440 Гц.

Но если длительность не кратна периоду, возникают осложнения. Если `duration = signal.period * 30,25`, то сигнал начинается с 0, а заканчивается на 1.

На рис. 3.4 (в центре) показан спектр этого сегмента. Опять же виден пик на 440 Гц, но теперь появились некие компоненты, которые «размазаны» в диапазоне от 240 до 640 Гц. Эту «размазню» и называют *утечкой спектра*, поскольку некоторая часть энергии основной частоты утекает в другие частоты.

На рис. 3.4 (справа) показан спектр сигнала, прошедшего окно. С ним утечки сократились заметно, но не полностью.

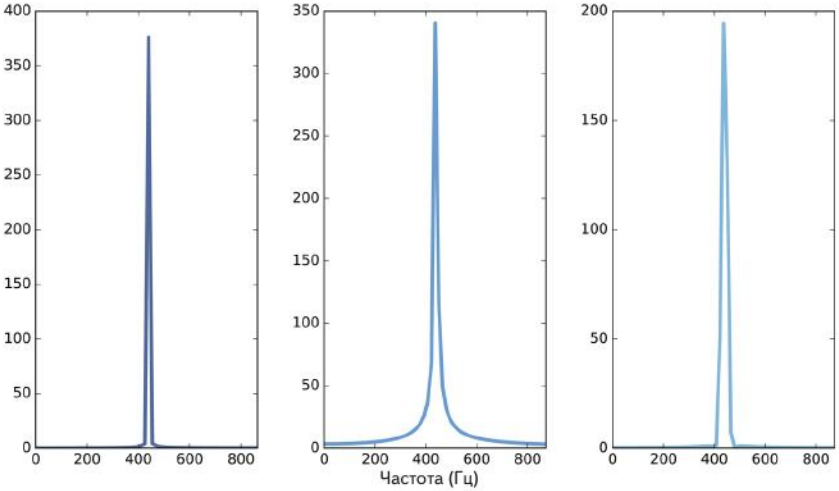

Рис. 3.4. Спектры периодического сегмента синусоиды (слева), апериодического сегмента (в центре) и апериодического сегмента с окном (справа)

В нашем примере утечка происходит от того, что ДПФ взято как бы от периодического сегмента, а он с разрывом.

Окна

Утечки можно уменьшить, сглаживая разрыв между началом и концом сегмента, и один из способов добиться этого – использование *окон*.

Окно – это функция, преобразующая апериодический сегмент в нечто похожее на периодическое. На рис. 3.5 (вверху) показан сегмент, где начало и конец сигнала соединяются с разрывом.

На рис. 3.5 (в центре) показано *окно Хэмминга* – одна из наиболее распространенных оконных функций. Совершенных оконных функций нет, но в том или ином случае некоторые из них оптимальны, а окно Хэмминга – это окно-«вездеход».

На рис. 3.5 (внизу) показан результат умножения окна на оригинальный сигнал. Там, где окно близко к 1, сигнал не изменяется. Там, где окно близко к 0, сигнал ослабляется. Поскольку окно мало на обоих краях, конец сегмента гладко стыкуется с началом.

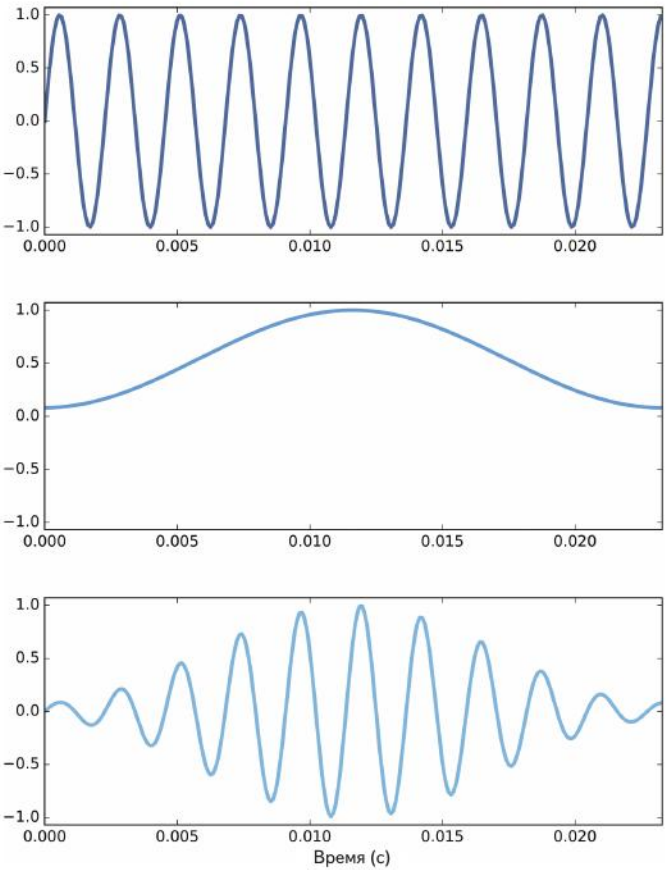

Рис. 3.5. Сегмент синусоиды (вверху), окно Хэмминга (средний) и произведение сегмента и окна (внизу)

Вот как выглядит код. `wave` дает `window`, включающее окно Хэмминга:

```
#class wave:
    def window(self, window):
        self.ys *= window
```

А в NumPy есть `hamming`, вычисляющая окно Хэмминга заданной длины:

```
window = np.hamming(len(wave))
wave.window(window)
```

NumPy дает функции для вычисления других оконных функций, включая `bartlett`, `blackman`, `hanning` и `kaiser`. В одном из упражнений в конце главы можно поэкспериментировать с этими окнами.

Реализация спектрограмм

Теперь, разобравшись с окнами, перейдем к `make_spectrogram`. Вот метод `wave`, вычисляющий спектрограммы:

```
#class wave:
    def make_spectrogram(self, seg_length):
        window = np.hamming(seg_length)
        i, j = 0, seg_length
        step = seg_length/2

        spec_map = {}

        while j < len(self.ys):
            segment = self.slice(i,j)
            segment.window(window)

            t = (segment.start + segment.end)/2
            spec_map[t] = segment.make_spectrum()

            i += step
            j += step
        return Spectrogram(spec_map, seg_length)
```

Это самая длинная функция в книге, так что если в ней разобраться, все остальное покажется простым.

Параметр `self` – это объект `wave`; `seg_length` – количество выборок в каждом сегменте.

`window` – окно Хэмминга с той же длиной, что и у сегментов.

`i` и `j` – индексы фрагментов, выбираемых из сигнала; `step` – смещение между сегментами. Поскольку `step` – это половина `seg_length`,

сегменты наполовину перекрываются. На рис. 3.6 показано, как выглядят перекрывающиеся окна.

`spec_map` – это указатель, сопоставляющий метки времени со `Spectrum`.

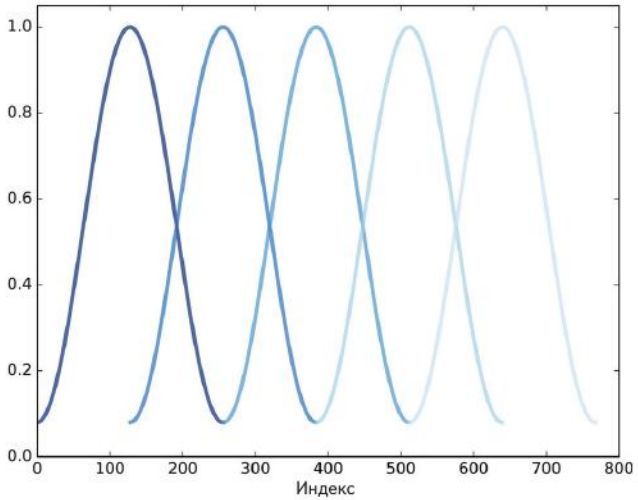

Рис. 3.6. Перекрывающиеся окна Хэмминга

Внутри петли `while` выбирается «срез» сигнала и накладывается окно; затем строится объект `Spectrum` и добавляется в `spec_map`. Номинальное время каждого сегмента `t`, – это середина сегмента.

Затем увеличиваются `i` и `j` – до тех пор пока `j` не выйдет за пределы `wave`.

Наконец, метод создает и возвращает объект `Spectrogram`. Вот определение класса:

```
class Spectrogram(object):
    def __init__ (self, spec_map, seg_length):
        self.spec_map = spec_map
        self.seg_length = seg_length
```

Как и многие методы `__init__`, этот просто сохраняет параметры как атрибуты.

`Spectrogram` дает `plot`, генерирующий псевдоцветной рисунок со временем по оси x и частотой по оси y.

Так реализован `Spectrogram`.

Упражнения

Решения этих упражнений находятся в блокноте `chap03soln.ipynb`.

Упражнение 3.1

Запустите и прослушайте примеры из блокнота `chap03.ipynb`. Вы найдете его в репозитории этой книги, а также на веб-странице `http://tinyurl.com/thinkdsp03`.

В примере с утечкой замените окно Хэмминга одним из окон, предоставляемых NumPy, и посмотрите, как они влияют на утечку. См. `http://docs.scipy.org/doc/numpy/reference/routines.window.html`.

Упражнение 3.2

Напишите класс, называемый `SawtoothChirp`, расширяющий `Chirp` и переопределяющий `evaluate` для генерации пилообразного сигнала с линейно увеличивающейся (или уменьшающейся) частотой.

Подсказка: надо совместить функции `evaluate` из `Chirp` и `SawtoothSignal`.

Нарисуйте эскиз спектрограммы этого сигнала, а затем распечатайте ее. Эффект биений должен быть очевиден, а если сигнал внимательно прослушать, то биения можно и услышать.

Упражнение 3.3

Создайте пилообразный чирп, меняющийся от 2500 до 3000 Гц, и на его основе сгенерируйте сигнал длительностью 1 с и частотой кадров 20 кГц. Нарисуйте, каким примерно будет `Spectrum`. Затем распечатайте `Spectrum` и посмотрите, правы ли вы.

Упражнение 3.4

В музыкальной терминологии *глиссандо* – это нота, меняющаяся от одной высоты до другой, то есть своеобразный чирп.

Найдите или запишите звук глиссандо и распечатайте спектрограмму первых нескольких секунд. Для справки: «Rhapsody in Blue» Джорджа Гершвина начинается с известного глиссандо на кларнете, и ее можно скачать с `http://archive.org/details/rhapblue11924`.

Упражнение 3.5

Тромбонист играет глиссандо, непрерывно дуя в мундштук и двигая кулису тромбона. При этом общая длина трубы меняется, а играемая нота обратно пропорциональна этой длине.

Если предположить, что музыкант двигает кулису с постоянной скоростью, как будет меняться во времени частота?

Напишите класс, называемый `TromboneGliss`, расширяющий `Chirp` и предоставляющий `evaluate`. Создайте сигнал, имитирующий глиссандо на тромбоне от C3 до F3, и обратно до C3. C3 – 262 Гц; F3 – 349 Гц.

Напечатайте спектрограмму полученного сигнала. На что похоже глиссандо на тромбоне – на линейный или же экспоненциальный чирп?

Упражнение 3.6

Сделайте или найдите запись серии гласных звуков и посмотрите на спектрограмму. Сможете ли вы различить разные гласные?

Глава 4. Шум

В английском языке шум означает нежелательные или неприятные звуки. (А в русском – еще и повод к выпивке!!! – *Прим. ред.*) В контексте же обработки сигналов это слово используется в двух значениях:

1. Как и в бытовом употреблении, слово «шум» может означать нежелательный сигнал любого рода. Если два сигнала мешают друг другу, то один из них относительно другого рассматривается как шум.
2. Шум – это также сигнал, содержащий компоненты с самыми разными частотами, но не имеющий гармонической структуры периодических сигналов, рассмотренных в предыдущих главах.

В настоящей главе будет рассматриваться именно второй вариант.

Код для этой главы находится в репозитории книги, в блокноте `chap04.ipynb` (см. раздел «Работа с кодом» на стр. 9). Также можно просмотреть код на веб-странице `http://tinyurl.com/thinkdsp04`.

Некоррелированный шум

Разберемся с шумом самым простым способом – сгенерируем его. Для начала попробуем *некоррелированный равномерный шум* (Uncorrelated Uniform Noise, *UU-шум*). «Равномерный» означает, что сигнал содержит случайные значения из равномерного распределения, то есть в заданных пределах каждое значение равновероятно. «Некоррелированный» означает, что значения независимые, то есть из одного значения нельзя получить сведения о других.

Вот класс, дающий UU-шум:

```
class UncorrelatedUniformNoise(_Noise):

    def evaluate(self, ts):
        ys = np.random.uniform(-self.amp, self.amp, len(ts))
        return ys
```

`UncorrelatedUniformNoise` наследует `_Noise`, наследующий `signal`.

Как обычно, для оценки сигнала функция `evaluate` берет время из `ts`. Она использует `np.random.uniform`, создающий значения из равномерного распределения. В этом примере значения находятся в диапазоне между `-amp` и `amp`.

Следующий пример генерирует UU-шум с длительностью 0,5 секунды на скорости 11 025 выборок в секунду:

```
signal = thinkdsp.UncorrelatedUniformNoise()
wave = signal.make_wave(duration=0.5, framerate=11025)
```

При прослушивании этот сигнал звучит как шум, слышимый при перестройке радио между станциями. На рис. 4.1 показано, как выглядит сигнал. Он действительно кажется случайным.

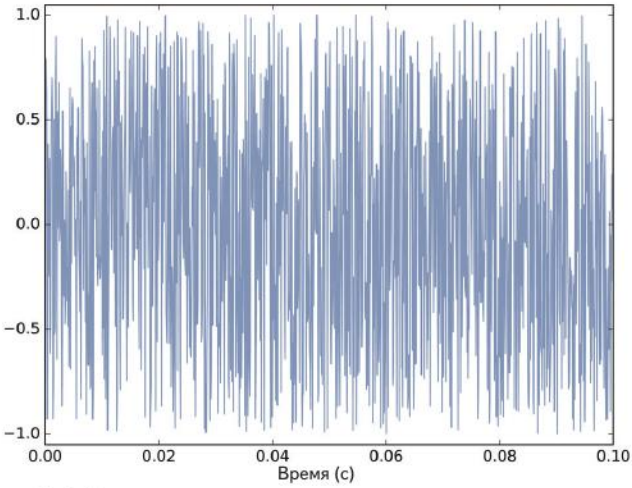

Рис. 4.1. Сигнал некоррелированного равномерного шума

Рассмотрим его спектр:

```
Spectrum = wave.make_Spectrum()
Spectrum.plot_power()
```

`Spectrum.plot_power` похож на `Spectrum.plot`, с тем отличием, что он выводит мощность, а не амплитуду. Мощность – это квадрат амплитуды. В этой главе вместо амплитуды используется мощность, поскольку в работе с шумами так правильнее.

На рис. 4.2 показаны результаты. Как и сам сигнал, спектр выглядит случайным. Он и в самом деле случайный, но, употребляя этот

термин, следует быть точным. О шумовом сигнале или о его спектре необходимо знать как минимум три вещи:

1. *Распределение.*
   Распределение случайного сигнала – это набор возможных значений и их вероятностей. Например, в равномерном шумовом сигнале набор значений лежит в диапазоне от –1 до 1, причем любые значения равновероятны. Альтернатива – *гауссов шум* с возможными значениями в диапазоне от минус до плюс бесконечности, но наиболее вероятны значения около 0, и вероятность спадает по гауссовой, или «колоколообразной», кривой.
2. *Корреляция.*
   Зависит ли одно значение сигнала от прочих или не зависит? В UU-шуме значения независимы. Альтернатива – *броуновский шум*, в нем каждое его новое значение есть сумма предыдущего и случайного шага. Так, если значение сигнала в некий момент времени велико, то следующее будет большим, а если мало, то будет малым.
3. *Связь между мощностью и частотой.*
   В спектре UU-шума мощность на всех частотах распределена одинаково; то есть средняя мощность одинакова на всех частотах. Альтернатива – *розовый шум*, его мощность обратно пропорциональна частоте; то есть мощность на частоте f получается из распределения, среднее которого пропорционально $1/f$.

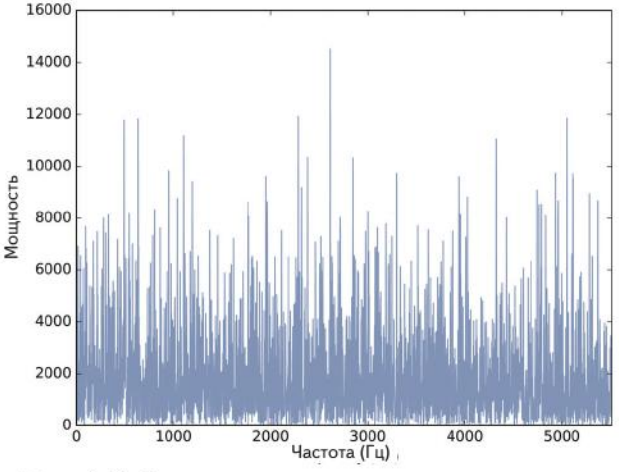

Рис. 4.2. Спектр мощности некоррелированного равномерного шума

Интегральный спектр

У UU-шума связь между мощностью и частотой видна четче, если смотреть на *интегральный спектр*. Это функция от частоты f, и она показывает полную мощность в спектре вплоть до f.

`Spectrum` предоставляет метод, вычисляющий `IntegratedSpectrum`:

```
def make_integrated_spectrum(self):
    cs = np.cumsum(self.power)
    cs /= cs[-1]
    return IntegratedSpectrum(cs, self.fs)
```

`self.power` – это NumPy-массив, содержащий мощность на каждой частоте. `np.cumsum` вычисляет полную сумму всех мощностей. Деление на последний элемент нормализует интегральный спектр так, что он лежит в пределах от 0 до 1.

Результатом будет `IntegratedSpectrum`. Вот определение класса:

```
class IntegratedSpectrum(object):
   def __init__ (self, cs, fs):
        self.cs = cs
        self.fs = fs
```

Как и `Spectrum`, `IntegratedSpectrum` дает `plot_power`, так что можно вычислить и напечатать интегральный спектр следующим образом:

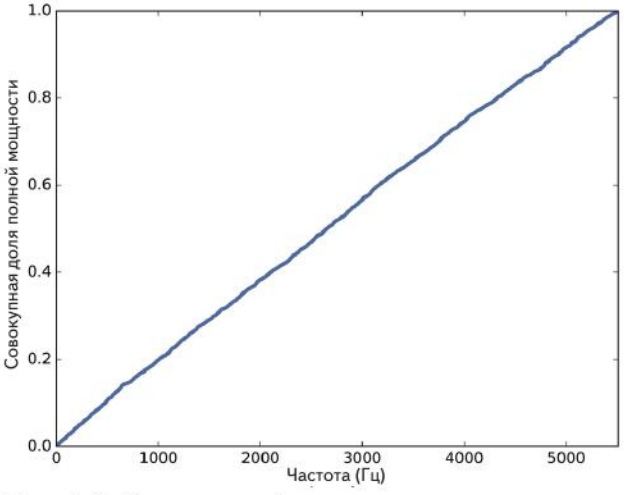

Рис. 4.3. Интегральный спектр некоррелированного равномерного шума

```
integ = spectrum.make_integrated_spectrum()
integ.plot_power()
```

Результат на рис. 4.3 – прямая линия, указывающая на то, что мощность на всех частотах в среднем неизменна. Шум с одинаковой мощностью на всех частотах называется *белым шумом*, по аналогии со светом, поскольку смесь света всех видимых (частот) цветов дает белый свет.

Броуновский шум

UU-шум не коррелирован, то есть каждое его значение не зависит от остальных. Альтернатива – *броуновский шум*, в котором каждое значение есть сумма предыдущего значения и некоего случайного шага.

Он называется броуновским по аналогии с броуновским движением, в котором движение частиц взвеси в жидкости кажется случайным из-за невидимого взаимодействия с жидкостью. Броуновское движение часто описывается с помощью *случайного блуждания* – математической модели пути, на котором расстояние между шагами описывается случайным распределением.

В одномерном случайном блуждании частица движется вверх или вниз на случайное расстояние на каждом этапе. Расположение частицы в любой момент времени определено суммой всех предыдущих шагов.

Очевиден способ создания броуновского шума: генерировать некоррелированные случайные шаги, а затем суммировать их. Вот определение класса, реализующего такой алгоритм:

```
class BrownianNoise(_Noise):

    def evaluate(self, ts):
        dys = np.random.uniform(-1, 1, len(ts))
        ys = np.cumsum(dys)
        ys = normalize(unbias(ys), self.amp)
        return ys
```

`evaluate` использует `np.random.uniform` для генерации некоррелированного сигнала и `np.cumsum` для вычисления полной суммы.

Поскольку сумма может выйти за пределы интервала от –1 до 1, надо использовать `unbias` для сдвига среднего значения в 0 и `normalize` для получения желаемой максимальной амплитуды.

Вот код, который создает объект `BrownianNoise` и печатает сигнал:

```
signal = thinkdsp.BrownianNoise()
```

```
wave = signal.make_wave(duration=0.5, framerate=11025)
wave.plot()
```

На рис. 4.4 показан результат. Сигнал блуждает вверх и вниз, но видна четкая взаимосвязь между соседними значениями. Большая амплитуда, как правило, остается большой, и наоборот.

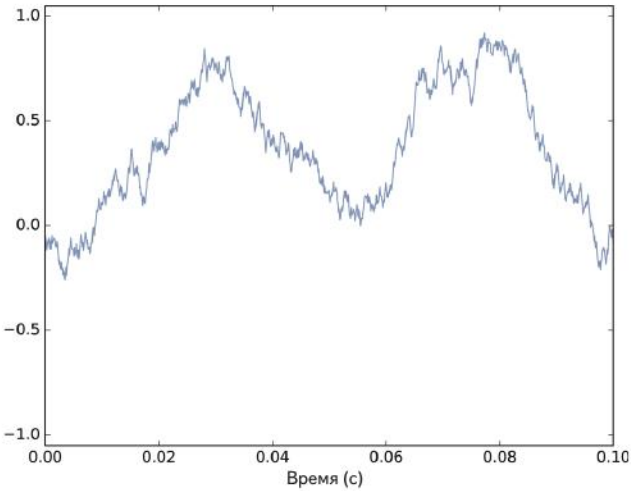

Рис. 4.4. Сигнал броуновского шума

Если спектр броуновского шума вывести в линейном масштабе, как показано на рис. 4.5 (слева), он выглядит необычно. Почти вся мощность сосредоточена на низких частотах; компоненты на высоких частотах почти не видны.

Вид спектра можно увидеть более четко, если вывести мощность и частоту в двойном логарифмическом масштабе. Вот код:

```
spectrum = wave.make_spectrum()
spectrum.plot_power(linewidth=1, alpha=0.5)
thinkplot.config(xscale='log', yscale='log')
```

Результат показан на рис. 4.5 (справа). Взаимосвязь между мощностью и частотой размытая, но строго линейная.

`Spectrum` поддерживает `estimate_slope`, использующую SciPy для вычисления среднего спектра мощности методом наименьших квадратов:

```
#class Spectrum

    def estimate_slope(self):
        x = np.log(self.fs[1:])
```

```
    y = np.log(self.power[1:])
    t = scipy.stats.linregress(x,y)
    return t
```

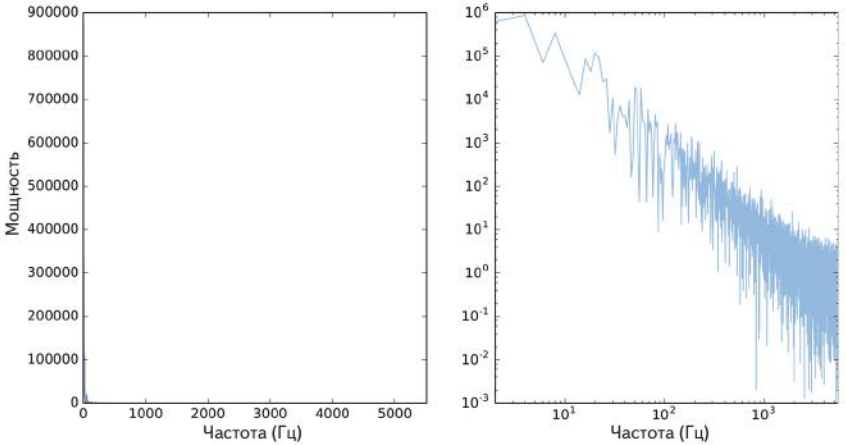

Рис. 4.5. Спектр броуновского шума в линейном (слева) и в двойном логарифмическом масштабе (справа)

Первый компонент спектра отбрасывается, поскольку он соответствует $f = 0$, а $\log 0$ не определен.

`estimate_slope` возвращает результат из `scipy.stats.linregress` – объекта, содержащего оценки уклона и перехода через ноль, коэффициент определенности (R^2), величину p и стандартное отклонение. Пока достаточно уклона.

Для броуновского шума уклон спектра мощности равен –2 (в главе 9 показано, почему), так что можно записать соотношение:

$$\log P = k - 2 \log f$$

где P – мощность, f – частота, а k – постоянная, которая сейчас не важна. Потенцируем обе части:

$$P = K / f^2$$

где K это e^k, но и это сейчас не важно. Важно то, что мощность пропорциональна $1/f^2$, – характерное свойство броуновского шума.

Броуновский шум называют также *красным шумом* по той же причине, по которой белый шум называют «белым». Если собрать видимый свет из составляющих с мощностью, пропорциональной $1/f^2$, большая часть мощности окажется в низкочастотной области спек-

тра, то есть в красной. Кроме того, броуновский шум именуют и коричневым шумом, но, видимо, в шутку, так что в дальнейшем мы не будем использовать это название.

Розовый шум

У красного шума взаимосвязь между частотой и мощностью выражается следующей формулой:

$$P = K / f^2$$

В степени 2 нет ничего особенного. В общем случае можно синтезировать шум с любой степенью, β:

$$P = K / f^{\beta}$$

При $\beta = 0$ мощность неизменна на всех частотах, и в результате будет белый шум. При $\beta = 2$ мы имеем красный шум.

Если β находится в пределах от 0 до 2, то результат будет между белым и красным шумом – он называется *розовым шумом*.

Есть несколько способов получения розового шума. Проще всего генерировать белый шум и пропускать его через фильтр НЧ со скатом, соответствующим нужной степени β. `thinkdsp` дает класс, предоставляющий сигнал вида «розовый шум»:

```
class PinkNoise(_Noise):

    def __init__ (self, amp=1.0, beta=1.0):
        self.amp = amp
        self.beta = beta
```

`amp` – нужная амплитуда сигнала. `beta` – нужный показатель степени. `PinkNoise` дает `make_wave`, генерирующий `wave`:

```
def make_wave(self, duration=1, start=0, framerate=11025):
    signal = UncorrelatedUniformNoise()
    wave = signal.make_wave(duration, start, framerate)
    spectrum = wave.make_spectrum()

    spectrum.pink_filter(beta=self.beta)

    wave2 = spectrum.make_wave()
    wave2.unbias()
    wave2.normalize(self.amp)
    return wave2
```

`duration` – длительность сигнала в секундах. `start` – время начала сигнала; оно нужно для того, чтобы у `make_wave` был единый

интерфейс для всех типов сигнала, хотя для случайного шума время начала не имеет значения. `framerate` – количество выборок в секунду.

`make_wave` создает сигнал белого шума, вычисляет его спектр, применяет фильтр с нужной степенью и преобразует отфильтрованный спектр обратно в сигнал. Затем убирается постоянная составляющая и нормализуется амплитуда.

`Spectrum` предоставляет `pink_filter`:

```
def pink_filter(self, beta=1.0):
    denom = self.fs ** (beta/2.0)
    denom[0] = 1
    self.hs /= denom
```

`pink_filter` делит каждый элемент спектра на $f^{\beta/2}$. Поскольку мощность – это квадрат амплитуды, данная операция делит мощность каждого компонента на f^{β}. Компонент с $f = 0$ обрабатывается особо – исключается деление на 0, но этот элемент также и постоянная составляющая сигнала и в любом случае устанавливается в 0.

На рис. 4.6 показан результат. Как и броуновский шум, сигнал блуждает вверх и вниз, причем заметна корреляция между соседними значениями, но выглядит он «более случайным». В следующей главе мы еще раз к этому вернемся для уточнения понятий «корреляция» и «более случайный».

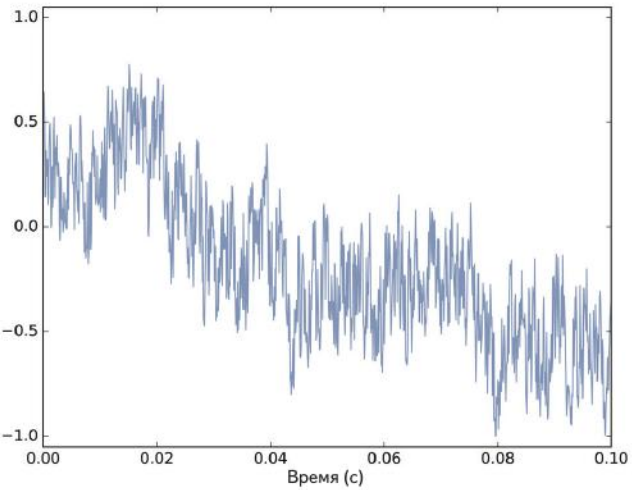

Рис. 4.6. Сигнал розового шума с $\beta = 1$

Наконец, на рис. 4.7 спектры белого, розового и красного шума показаны на общей двойной логарифмической шкале. На этом рисунке очевидна взаимосвязь между показателем степени β и наклоном спектра.

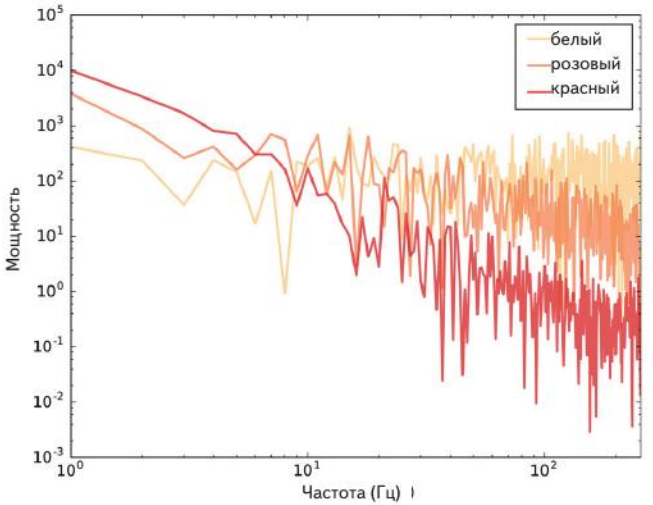

Рис. 4.7. Спектры белого, розового и красного шума на двойной логарифмической шкале

Гауссов шум

Выше уже был рассмотрен некоррелированный равномерный шум; мы отметили, что его спектр имеет одинаковую среднюю мощность на всех частотах, то есть это белый шум.

Но, когда речь идет о белом шуме, не всегда имеется в виду UU-шум. Часто в таких случаях подразумевают *некоррелированный гауссов шум* (UG).

`thinkdsp` поддерживает реализацию UG-шума:

```
class UncorrelatedGaussianNoise(_Noise):

    def evaluate(self, ts):
        ys = np.random.normal(0, self.amp, len(ts))
        return ys
```

`np.random.normal` возвращает NumPy-массив значений с гауссовым распределением, в данном случае с нулевым средним и стандартным отклонением `self.amp`. В теории диапазон значений лежит от

минус до плюс бесконечности, но здесь около 99% значений находятся в пределах –3 ... +3.

UG-шум многим похож на UU-шум. У спектра одинаковая средняя мощность на всех частотах, поэтому UG-шум также белый. Но есть интересное свойство: спектр UG-шума – также UG-шум. Точнее, действительные и мнимые части спектра – некоррелированные гауссовы значения.

Проверим это утверждение: сгенерируем спектр UG-шума, а затем «кривую нормального распределения вероятности» и проверим графически, будет ли распределение гауссовым:

```
signal = thinkdsp.UncorrelatedGaussianNoise()
wave = signal.make_wave(duration=0.5, framerate=11025)
spectrum = wave.make_spectrum()

thinkstats2.NormalProbabilityPlot(spectrum.real)
thinkstats2.NormalProbabilityPlot(spectrum.imag)
```

`NormalProbabilityPlot` содержится в `thinkstats2`, входящей в репозиторий этой книги. Подробнее о графиках нормального рапределения вероятностей можно прочитать в главе 5 книги *Think Stats* на сайте `http://thinkstats2.com`.

На рис. 4.8 показаны результаты. Серыми линиями обозначена усредненная линейная модель данных; а темными линиями – сами данные.

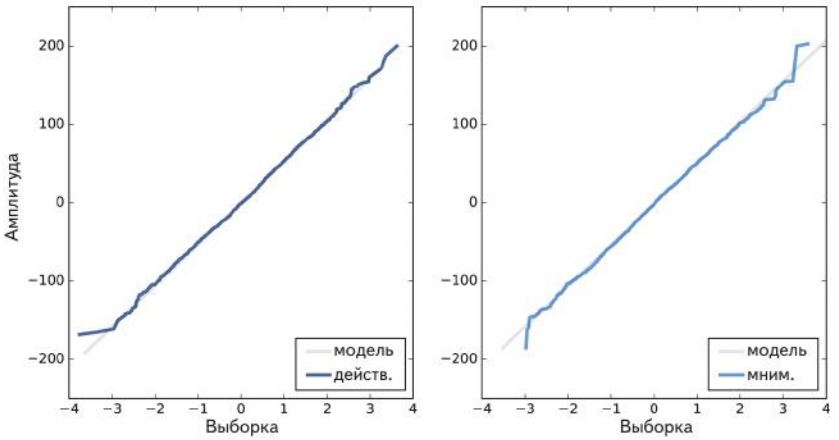

Рис. 4.8. График нормальной вероятности для действительной и мнимой части спектра гауссова шума

Прямая линия на графике нормальной вероятности показывает, что данные взяты из гауссова распределения. За исключением некоторых случайных вариаций на краях эти линии прямые, то есть спектр UG-шума – также UG-шум.

Спектр UU-шума в первом приближении – также UG-шум. В самом деле, согласно центральной предельной теореме, спектры большинства из некоррелированных шумов будут примерно гауссовыми, пока у распределения конечное среднее и стандартное отклонение, а количество выборок велико.

Упражнения

Решения этих упражнений находятся в блокноте `chap04soln.ipynb`.

Упражнение 4.1

На сайте Soft Murmur можно послушать множество природных источников шума, включая дождь, волны, ветер, и др. На веб-странице `http://asoftmurmur.com/about/` приведен перечень записей; большинство из них хранится на `http://freesound.org`.

Скачайте некоторые из этих файлов и вычислите спектры каждого сигнала. Похож ли их спектр мощности на белый, розовый или броуновский шум? Как спектр меняется во времени?

Упражнение 4.2

В шумовом сигнале частотный состав меняется во времени. На большом интервале мощность на всех частотах одинакова, а на коротком мощность на каждой частоте случайна.

Для оценки долговременной средней мощности на каждой частоте можно разорвать сигнал на сегменты, вычислить спектр мощности для каждого сегмента, а затем найти среднее по сегментам. Об этом алгоритме можно прочитать подробнее на странице `http://en.wikipedia.org/wiki/Bartlett's_method` (на англ. яз.).

Реализуйте метод Бартлетта и используйте его для оценки спектра мощности шумового сигнала. Подсказка: посмотрите на реализацию `make_spectrogram`.

Упражнение 4.3

На веб-странице `http://www.coindesk.com/price` можно скачать в виде CSV-файла исторические данные о ежедневной цене BitCoin. Откройте этот файл и вычислите спектр цен BitCoin как функцию времени. Похоже ли это на белый, розовый или броуновский шум?

Упражнение 4.4

Счетчик Гейгера – прибор для обнаружения радиации. Когда ионизирующие частицы попадают в детектор, на его выходе появляются импульсы тока. Общий выход в определенный момент времени можно смоделировать некоррелированным пуассоновым шумом (UP), где каждая выборка есть случайное число из распределения Пуассона, соответствующее количеству частиц, обнаруженных за интервал измерения.

Напишите класс, называемый `UncorrelatedPoissonNoise`, наследующий `thinkdsp._Noise` и предоставляющий `evaluate`. Следует использовать `Np.random.poisson` для генерации случайных величин из распределения Пуассона. Параметр этой функции `lam` – это среднее число частиц за время каждого интервала. Можно использовать атрибут `amp` для определения `lam`. Например, при частоте кадров 10 кГц и `amp` 0,001 получится около 10 «щелчков» в секунду.

Сгенерируйте пару секунд UP и прослушайте. Для малых значений `amp`, например 0,001, звук будет как у счетчика Гейгера. При больших значениях он будет похож на белый шум. Вычислите и напечатайте спектр мощности и посмотрите, так ли это.

Упражнение 4.5

В этой главе алгоритм для генерации розового шума концептуально простой, но затратный. Существуют более эффективные варианты, например алгоритм Voss–McCartney. Изучите этот способ, реализуйте его, вычислите спектр результата и убедитесь, что соотношение между мощностью и частотой соответствующее.

Глава 5. Автокорреляция

В предыдущей главе белый шум охарактеризован как «некоррелированный» в предположении, что каждое значение не зависит от других, а броуновский шум – как «коррелированный», поскольку каждое новое значение зависит от предыдущего. В этой главе мы определим эти термины точнее и рассмотрим автокорреляционную функцию – полезнейший инструмент для анализа сигналов.

Код для этой главы находится в репозитории книги, в блокноте `chap05.ipynb` (см. раздел «Работа с кодом» на стр. 9). Также код можно просмотреть на веб-странице `http://tinyurl.com/thinkdsp05`.

Корреляция

Обычно корреляция между переменными означает, что в каждой известной величине есть некоторая информация о другой. Из нескольких способов оценки корреляции наиболее известен коэффициент корреляции Пирсона – произведение моментов, обычно обозначаемое ρ. Для двух переменных x и y, содержащих каждая N значений:

$$\rho = \sum_i (x_i - \mu_x)(y_i - \mu_y) \,/\, N\sigma_x\sigma_y,$$

где μ_x и μ_y – средние от x и y, а σ_x и σ_y – их стандартные отклонения.

Корреляция Пирсона определена в интервале от –1 до +1 (включительно). Если ρ положительно, то корреляция положительная, то есть если одна переменная велика, то и другая тоже велика. Если ρ отрицательное, то корреляция отрицательная, – если одна переменная велика, то другая мала.

Значение ρ показывает силу корреляции. Если ρ равно +1 или –1, переменные идеально коррелируют, то есть если известна одна, то можно точно предсказать другую. Если ρ близко к нулю, корреляция, вероятно, слабая, так что при одном известном мало что можно сказать о другом.

Принято говорить «вероятно, слабая» из-за возможных (вероятных) нелинейных соотношений, не учитываемых коэффициентом корреляции. Нелинейные соотношения играют важную роль в статистике, а в обработке сигналов они редки и далее не рассматриваются.

В Python есть несколько способов вычисления корреляции. `np.corrcoef` принимает любое число переменных и вычисляет *матрицу корреляции*, включающую корреляции между каждой парой переменных.

Вот пример для двух переменных. Во-первых, определим функцию, строящую синусоидальные сигналы с разными фазами:

```
def make_sine(offset):
    signal = thinkdsp.SinSignal(freq=440, offset=offset)
    wave = signal.make_wave(duration=0.5, framerate=10000)
    return wave
```

Затем создадим два сигнала с разными начальными фазами:

```
wave1 = make_sine(offset=0)
wave2 = make_sine(offset=1)
```

На рис. 5.1 показаны несколько периодов этих сигналов. Когда один сигнал велик, другой тоже достаточно велик, поэтому ожидаемо, что они коррелированы:

```
>>> corr_matrix = np.corrcoef(wave1.ys, wave2.ys, ddof=0)
[[ 1.    0.54]
 [ 0.54  1.  ]]
```

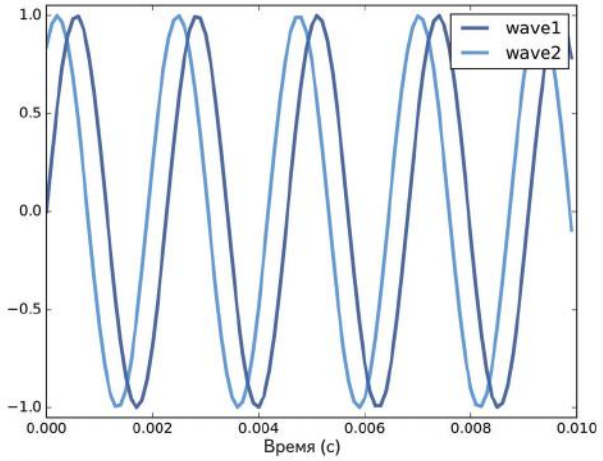

Рис. 5.1. Для двух синусоидальных сигналов, отличающихся по фазе на 1 радиан, коэффициент корреляции – 0,54

Параметр `ddof = 0` означает, что `corrcoef` следует делить на N, как в уравнении выше, а не на $N - 1$, как было бы по умолчанию.

Результат – матрица корреляции. Первый элемент – это корреляция `wave1` с самим собой (всегда 1). Аналогично последний элемент – это корреляция `wave2` с самим собой.

Элементы вне диагонали содержат искомые значения – корреляцию `wave1` и `wave2`. Значение 0,54 указывает, что сила корреляции умеренная.

С ростом разности фаз корреляция уменьшается вплоть до разности фаз 180 градусов, когда сигналы противофазны, а корреляция равна –1. Затем она вновь увеличивается до тех пор, пока разность фаз не достигнет 360 градусов. В этот момент замыкается полный круг и корреляция становится +1.

На рис. 5.2 показана связь между корреляцией и разностью фаз синусоидальных сигналов. Форма этой кривой давно известна – это косинус.

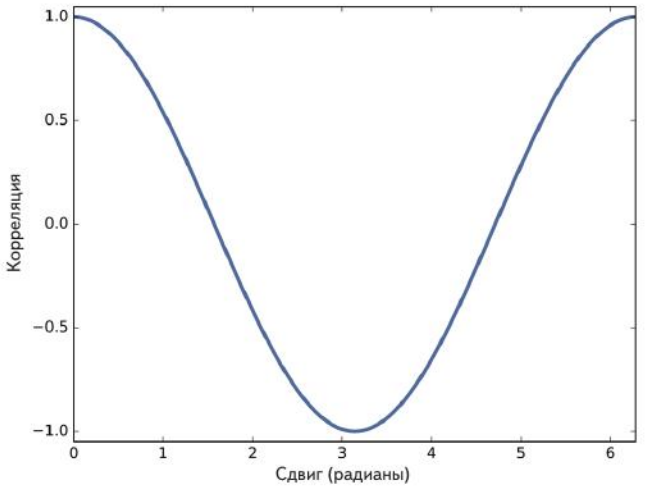

Рис. 5.2. Корреляция двух синусоидальных сигналов в зависимости от разности фаз между ними. Это косинус

`thinkdsp` предоставляет простой интерфейс для вычисления корреляции между сигналами:

```
>>> wave1.corr(wave2)
0,54
```

Последовательная корреляция

Сигналы – это чаще всего измерения изменяющихся во времени величин. Например, изученные ранее звуковые сигналы – это измерения напряжения (или тока), соответствующие изменениям давления воздуха, воспринимаемым как звук.

Такие измерения почти всегда имеют последовательную корреляцию, или корреляцию между любым элементом и следующим за ним (или предыдущим). Для расчета последовательной корреляции сигнал можно сдвинуть и вычислить корреляцию сдвинутой версии с оригиналом:

```
def serial_corr(wave, lag=1):
    n = len(wave)
    y1 = wave.ys[lag:]
    y2 = wave.ys[:n-lag]
    corr = np.corrcoef(y1, y2, ddof=0)[0, 1]
    return corr
```

`serial_corr` берет объект `wave` и целое число `lag`, задающее начальную фазу сигнала, а затем вычисляет корреляцию сигнала с его сдвинутой копией.

Проверим эту функцию на шумовых сигналах из предыдущей главы. Поскольку UU-шум некоррелированный просто по способу его создания (даже без учета названия), то

```
signal = thinkdsp.UncorrelatedGaussianNoise()
wave = signal.make_wave(duration=0.5, framerate=11025)
serial_corr(wave)
```

Результат этого примера – 0,006, указывающий на очень малую последовательную корреляцию. При выполнении теста могут получиться и другие значения, но они должны быть сравнимо малыми.

В сигнале броуновского шума каждое новое значение есть сумма предыдущего и случайного шага, поэтому ожидаема сильная последовательная корреляция:

```
signal = thinkdsp.BrownianNoise()
wave = signal.make_wave(duration=0.5, framerate=11025)
serial_corr(wave)
```

И точно, полученный автором результат больше 0,999.

Так как розовый шум есть нечто среднее между броуновским шумом и UU-шумом, можно ожидать и среднюю корреляцию:

```
signal = thinkdsp.PinkNoise(beta=1)
wave = signal.make_wave(duration=0.5, framerate=11025)
serial_corr(wave)
```

С параметром $\beta = 1$ последовательная корреляция у автора получилась 0,851. При изменении параметра от $\beta = 0$ для UU-шума до $\beta = 2$ для броуновского последовательная корреляция меняется от 0 почти до 1, что и показано на рис. 5.3.

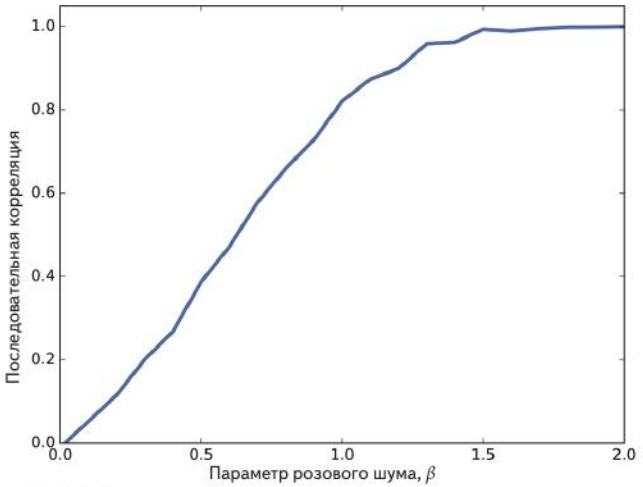

Рис. 5.3. Последовательная корреляция для розового шума в зависимости от параметра

Автокорреляция

В предыдущем разделе корреляция вычислялась между каждым значением и следующим за ним, поэтому элементы массива сдвигали на 1. Но последовательную корреляцию легко вычислить и для иных интервалов.

`serial_corr` можно рассматривать как функцию, сопоставляющую значение `lag` соответствующей корреляции, и ее можно оценить, перебирая значения `lag` в цикле:

```
def autocorr(wave):
    lags = range(len(wave.ys)//2)
    corrs = [serial_corr(wave, lag) for lag in lags]
    return lags, corrs
```

`autocorr` берет объект `wave` и возвращает автокорреляционную функцию как пару последовательностей: `lags` – это последовательность целых чисел от 0 до половины длины сигнала; `corrs` – это значения последовательной корреляции для всех `lag`.

На рис. 5.4 показаны автокорреляционные функции для розового шума с тремя значениями β. Для малых значений β сигнал менее коррелирован и автокорреляционная функция быстро спадает до нуля. Для больших значений последовательная корреляция сильнее и спадает медленнее. При $\beta = 1{,}7$ последовательная корреляция сильна даже для больших `lag`. Это явление называется *зависимостью дальнего действия*; оно указывает, что значение сигнала зависит от многих предыдущих значений.

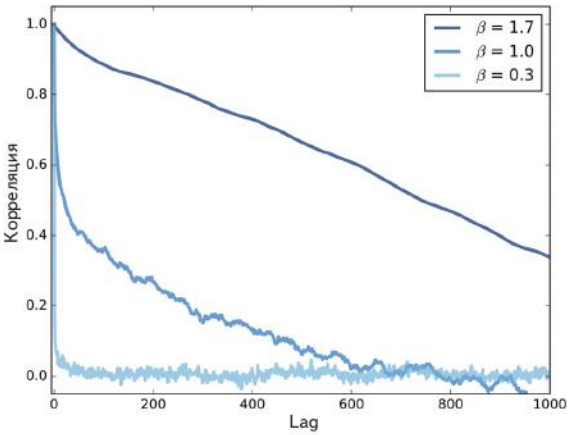

Рис. 5.4. Автокорреляционные функции для розового шума с набором параметров

Автокорреляция периодических сигналов

У автокорреляции розового шума интересные математические свойства, но пользы от них мало. Автокорреляция периодических сигналов полезнее.

Так, на сайте `https://freesound.org` есть запись вокального исполнения чирпов; файл включен в репозиторий этой книги (`https://github.com/AllenDowney/ThinkDSP/blob/master/code/28042__bcjordan__voicedownbew.wav`). Запись можно прослушать в блокноте Jupyter для этой главы (`chap05.ipynb`).

На рис. 5.5 показана спектрограмма этого сигнала. Основная частота и некоторые из гармоник видны четко. Чирп начинается в районе 500 Гц и спадает примерно до 300 Гц или, грубо, от C5 до E4.

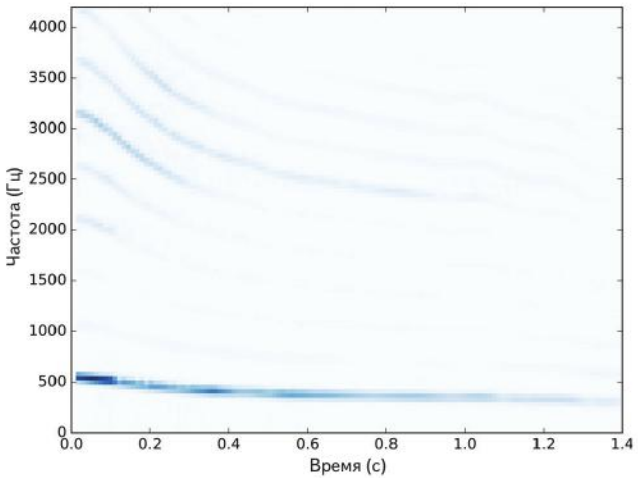

Рис. 5.5. Спектрограмма вокального чирпа

Оценить высоту тона в конкретный момент времени можно по спектру, но это не очень интересно. Разберемся, взяв короткий сегмент сигнала и напечатав его спектр:

```
duration = 0.01
segment = wave.segment(start=0.2, duration=duration)
spectrum = segment.make_spectrum()
spectrum.plot(high=1000)
```

Этот сегмент начинается с 0,2 секунды и длится 0,01 секунды. На рис. 5.6 показан его спектр. Есть четкий пик около 400 Гц, но точно определить высоту тона трудно. Длина сегмента – 441 выборка, а частота кадров – 44 100 Гц, поэтому разрешение по частоте – 100 Гц (см. раздел «Предел Габора», стр. 40). Значит, ошибка в высоте тона может быть до 50 Гц; в музыкальных терминах диапазон от 350 Гц до 450 Гц составит 5 полутонов, а это большая разница!

Большее разрешение по частоте получим, взяв сегмент подлиннее, но, поскольку высота тона меняется во времени, получится еще и «размытие»; то есть пик «размажется» между начальной и конечной высотой тона в сегменте, как уже было показано в разделе «Спектр чирпа» на стр. 38.

Оценим высоту тона точнее, применив автокорреляцию. Если сигнал периодический, то пик автокорреляции придется на момент, когда `lag` равен периоду.

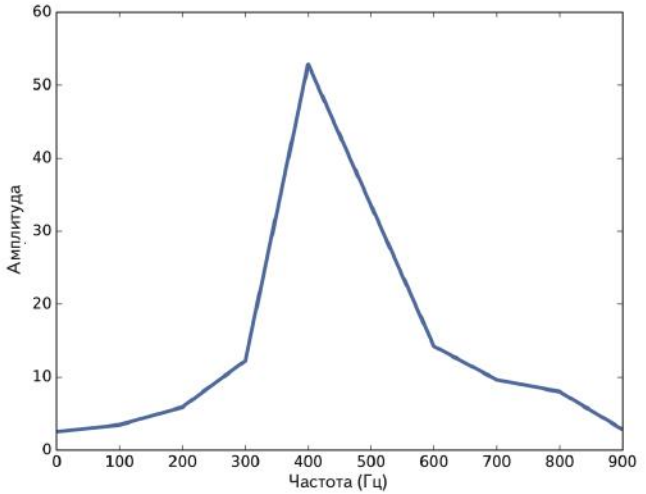

Рис. 5.6. Спектр сегмента вокального чирпа

Рассмотрим, как это работает, напечатав два сегмента одной и той же записи:

```
def plot_shifted(wave, offset=0.001, start=0.2):
    thinkplot.preplot(2)
    segment1 = wave.segment(start=start, duration=0.01)
    segment1.plot(linewidth=2, alpha=0.8)

    segment2 = wave.segment(start=start-offset, duration=0.01)
    segment2.shift(offset)
    segment2.plot(linewidth=2, alpha=0.4)

    corr = segment1.corr(segment2)
    text = r'$\rho =$ %.2g' % corr
    thinkplot.text(segment1.start+0.0005, -0.8, text)
    thinkplot.config(xlabel='Time (s)')
```

Один сегмент начинается с 0,2 секунды, другой – на 0,0023 секунды позже. На рис. 5.7 показан результат. Сегменты похожи, и их корреляция 0,99. Из этого результата следует, что период составляет около 0,0023 секунд, а частота, соответственно, – 435 Гц.

В этом примере период подобран методом проб и ошибок. Процесс подбора можно автоматизировать, используя автокорреляционную функцию:

```
lags, corrs = autocorr(segment)
thinkplot.plot(lags, corrs)
```

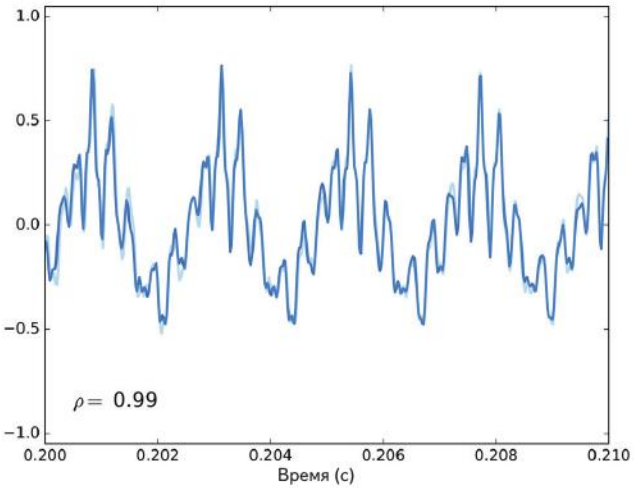

Рис. 5.7. Два сегмента чирпа: один начинается на 0,0023 секунды позже другого

На рис. 5.8 показана автокорреляционная функция сегмента начиная с $t = 0{,}2$ с. Первый пик появился при `lag` = 101. Вычислим соответствующую этому периоду частоту:

```
period = lag / segment.framerate
frequency = 1 / period
```

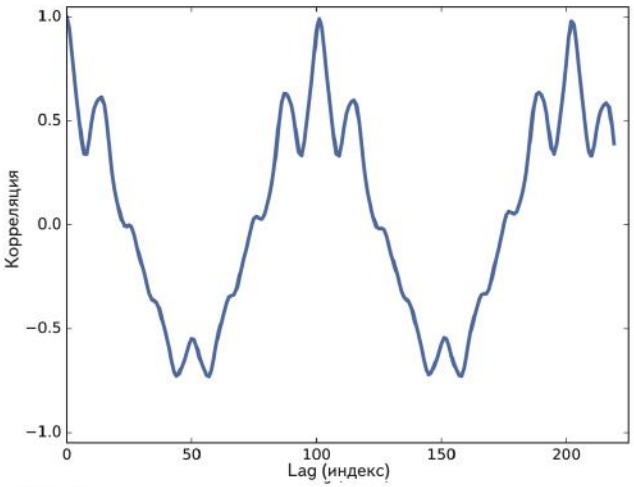

Рис. 5.8. Автокорреляционная функция для сегмента из чирпа

Искомая основная частота – 437 Гц. Для оценки точности подбора можно запустить тот же расчет с `lag` = 100 и 102, соответствующими частотам 432 Гц и 441 Гц. Разрешение по частоте у способа с автокорреляцией менее 10 Гц, а у спектрального – 100 Гц. В музыкальных терминах ожидаемая ошибка – около 30 центов (треть полутона).

Корреляция как скалярное произведение

В начале главы дано определение коэффициента корреляции Пирсона:

$$\rho = \sum_i (x_i - \mu_x)(y_i - \mu_y) / N\sigma_x\sigma_y$$

Это ρ уже использовалось для определения и последовательной, и автокорреляции. Так эти термины используются в статистике, а в смысле обработки сигналов определения немного иные.

В обработке сигналов обычно обрабатываются сигналы без смещения (постоянной составляющей), их среднее равно 0, и нормализованные, со стандартным отклонением, равным 1. В этом случае определение ρ упрощается:

$$\rho = 1/N \sum_i x_i y_i$$

Обычно упрощают еще больше:

$$r = \sum_i x_i y_i$$

Это определение корреляции не «стандартизировано», поэтому его интервал не всегда от –1 до +1. Но у него другие полезные свойства.

Если рассматривать x и y как вектора, то это формула скалярного произведения, $x \cdot y$. См. `http://en.wikipedia.org/wiki/Dot_product` (на англ. яз.)

Скалярное произведение указывает на степень похожести сигналов. Если они нормализованы, то их стандартные отклонения равны 1:

$$x \cdot y = \cos\theta$$

где θ – угол между векторами. И это объясняет, почему на рис. 5.2 – косинус.

Использование NumPy

NumPy дает функцию `correlate`, вычисляющую корреляцию двух функций или автокорреляцию одной функции. Ее можно использовать для вычисления автокорреляции сегмента из предыдущего раздела:

```
corrs2 = np.correlate(segment.ys, segment.ys, mode='same')
```

Опция `mode` указывает `correlate`, какой нужен интервал `lag`. Со значением `'same'` он будет от $-N/2$ до $N/2$, где N – длина массива сигнала.

На рис. 5.9 показан результат. Он симметричен, потому что два сигнала идентичны, и отрицательный `lag` у одного дает такой же эффект, как и положительный `lag` у другого. Для сравнения с результатами `autocorr` можно выбрать вторую половину:

```
N = len(corrs2)
half = corrs2[N//2:]
```

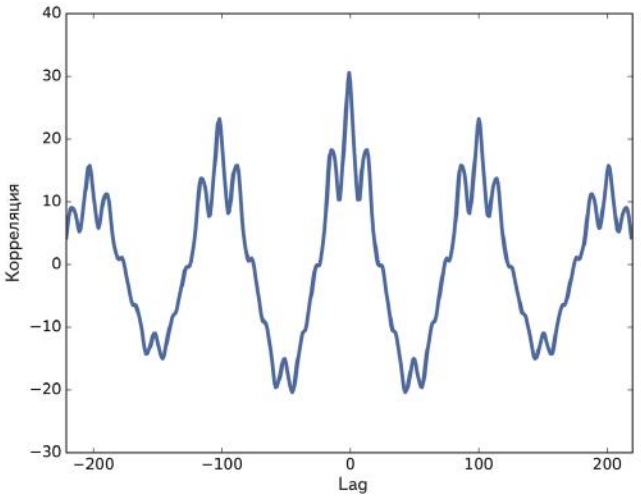

Рис. 5.9. Автокорреляционная функция, вычисленная с `np.correlate`

Если сравнить рис. 5.9 с рис. 5.8, то видно, что корреляции, вычисленные с помощью `np.correlate`, уменьшаются с ростом `lag`,

поскольку `np.correlate` использует нестандартное определение корреляции; с ростом `lag` количество точек в месте наложения двух сигналов становится все меньше, и величина корреляции уменьшается.

Это можно исправить, разделив результат на длины (`lengths`):

```
lengths = range(N, N//2, -1)
half /= lengths
```

И наконец, можно нормализовать результаты так, что корреляция при `lag` = 0 будет равна 1:

```
half /= half[0]
```

С этими поправками результаты, рассчитанные `autocorr` и `np.correlate`, станут почти одинаковыми. Они по-прежнему отличаются на 1–2%. Причина не важна, но для полноты изложения: `autocorr` стандартизирует корреляции независимо для каждого `lag`, а `np.correlate` стандартизирует их все только в конце.

Итак, мы выяснили, что такое автокорреляция и как ее использовать для оценки основного периода сигнала, а также рассмотрели два способа его вычислить.

Упражнения

Решения этих упражнений находятся в блокноте `chap05soln.ipynb`.

Упражнение 5.1

Блокнот Jupyter этой главы, `chap05.ipynb`, содержит приложение, в котором можно вычислить автокорреляции для различных `lag`. Оцените высоты тона вокального чирпа для нескольких времен начала сегмента.

Упражнение 5.2

Пример кода в `chap05.ipynb` показывает, как использовать автокорреляцию для оценки основной частоты периодического сигнала. Инкапсулируйте этот код в функцию, названную `estimate_fundamental`, и используйте ее для отслеживания высоты тона записанного звука.

Проверьте, насколько хорошо она работает, накладывая оценки высоты тона на спектрограмму записи.

Упражнение 5.3

Для упражнений в предыдущей главе были нужны исторические цены BitCoins, и надо было оценить спектр мощности изменения цен. Используя те же данные, вычислите автокорреляции цен в платежной системе Bitcoin. Быстро ли спадает автокорреляционная функция? Есть ли признаки периодичности процесса?

Упражнение 5.4

В репозитории этой книги есть блокнот Jupyter под названием `saxophone.ipynb`, в котором исследуются автокорреляция, восприятие высоты тона и явление, называемое *подавленная основная*. Прочтите этот блокнот и «погоняйте» примеры. Выберите другой сегмент записи и вновь поработайте с примерами.

У Ви Харт (Vi Hart) есть отличное видео под названием «Так что же там с шумами? (Наука и математика звука, частота и высота тона)». Она демонстрирует феномен подавленной основной и объясняет, как воспринимается высота тона (по крайней мере, насколько об этом известно). См. `https://www.youtube.com/watch?v=i_0DXxNeaQ0`.

Глава 6. Дискретное косинусное преобразование

Тема этой главы – дискретное косинусное преобразование (ДКП), используемое в MP3 и соответствующих форматах сжатия музыки, в JPEG и подобных форматах изображений, в семействе форматов MPEG для видео.

ДКП во многом похоже на дискретное преобразование Фурье (ДПФ), использованное выше в спектральном анализе. Изучив работу ДКП, легче разобраться в ДПФ.

Вот порядок изучения:

1. Задача синтеза: как построить сигнал из набора частотных компонент и их амплитуд?
2. Задача синтеза по-новому: вариант с массивами NumPy. На этом этапе растет скорость и дается представление о следующем шаге.
3. Задача анализа: расчет амплитуды каждой частотной компоненты, если заданы сигнал и набор частот. Первое решение концептуально простое, но медленное.
4. Задача с линейной алгеброй: построение более эффективного алгоритма с ее помощью. Все вопросы будут подробно разобраны по мере изучения темы, а знание линейной алгебры упростит и ускорит понимание предмета.

Код для этой главы находится в репозитории книги, в блокноте `chap06.ipynb` (см. раздел «Работа с кодом» на стр. 9). Также код можно просмотреть на веб-странице `http://tinyurl.com/thinkdsp06`.

Синтез

Пусть даны список амплитуд и список частот, и надо создать сигнал в виде суммы этих частотных компонент. С объектами из модуля

thinkdsp есть простой способ выполнения этой операции, называемой *синтезом*:

```
def synthesize1(amps, fs, ts):
    components = [thinkdsp.CosSignal(freq, amp)
                  for amp, freq in zip(amps, fs)]
    signal = thinkdsp.SumSignal(*components)

    ys = signal.evaluate(ts)
    return ys
```

Здесь amps – список амплитуд, fs – список частот, ts – последовательность моментов времени, в которых надо оценивать сигнал.

components – список объектов CosSignal, по одному для каждой пары амплитуда–частота. SumSignal – сумма этих частотных компонент.

И наконец, evaluate вычисляет значение сигнала в каждый момент времени из ts. Протестируем эту функцию следующим образом:

```
amps = np.array([0.6, 0.25, 0.1, 0.05])
fs = [100, 200, 300, 400]
framerate = 11025

ts = np.linspace(0, 1, framerate)
ys = synthesize1(amps, fs, ts)
wave = thinkdsp.wave(ys, framerate)
```

В данном примере создается сигнал, содержащий основную частоту 100 Гц и три гармоники (100 Гц – это G2 диез). Получается 1 секунда этого сигнала в виде 11 025 выборок, а результат помещается в объект wave.

Концептуально синтез – это просто. Но в таком виде он не поможет в *анализе*, то есть обратной задаче: как для данного сигнала определить частотные компоненты и их амплитуды?

Синтез с массивами

Вот еще один способ записи synthesize:

```
def synthesize2(amps, fs, ts):
    args = np.outer(ts, fs)
    M = np.cos(PI2 * args)
    ys = np.dot(M, amps)
    return ys
```

Эта функция очень необычна, но делает то же самое. Рассмотрим, как она работает:

1. `np.outer` вычисляет *тензорное произведение* `ts` и `fs`. Результатом будет массив строк для всех элементов `ts` и столбцов для всех элементов `fs`. Каждый элемент массива есть произведение частоты и времени, `ft`.
2. `args` умножается на 2π и берется `cos`, поэтому каждый элемент результата есть $\cos(2\pi ft)$. Поскольку `ts` работает вниз по столбцам, в каждом столбце будут выборки, взятые с некоторой скоростью из косинусоидального сигнала определенной частоты.
3. `np.dot` поэлементно умножает каждую строку `M` на `amps` и складывает произведения. В терминах линейной алгебры матрица `M` умножается на вектор `amps`. В терминах обработки сигналов вычисляется взвешенная сумма частотных компонент.

На рис. 6.1 показана структура этих расчетов. Каждая строка матрицы `M` соответствует времени от 0,0 до 1,0 секунды; t_n – время n-й строки. Каждый столбец соответствует частотам от 100 до 400 Гц; f_k – частота k-го столбца.

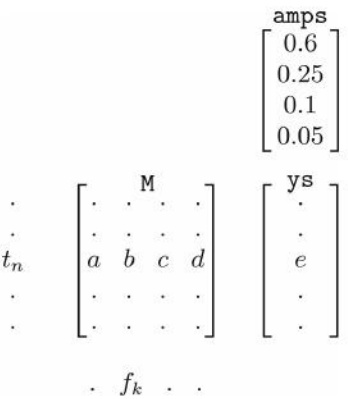

Рис. 6.1. Синтез с массивами

Разметим n-ю строку буквами от a до d; например, a – это $\cos[2\pi(100)t_n]$.

Результат скалярного произведения `ys` – это вектор с одним элементом на каждую строку `M`. n-й элемент, помеченный e, – это сумма произведений:

$$e = 0{,}6a + 0{,}25b + 0{,}1c + 0{,}05d$$

Аналогично и с другими элементами `ys`. Так, каждый элемент `ys` есть сумма четырех частотных компонент в некий момент времени,

умноженная на соответствующую амплитуду. А это именно то, что нам нужно.

Используем код из предыдущего раздела и убедимся, что результаты обеих версий `synthesize` одинаковы:

```
ys1 = synthesize1(amps, fs, ts)
ys2 = synthesize2(amps, fs, ts)
max(abs(ys1 - ys2))
```

Наибольшая разница между `ys1` и `ys2` будет `1e-13`, и то за счет ошибки округления.

Вычисления по правилам линейной алгебры делают код быстрее и компактнее. В линейной алгебре есть краткие обозначения для операций с матрицами и векторами. Например, записать `synthesize` можно так:

$$M = \cos(2\pi t \otimes f)$$
$$y = Ma$$

где a – вектор амплитуд, t – вектор времен, f – вектор частот, а $\otimes$ – символ тензорного произведения двух векторов.

Анализ

Теперь решим задачу анализа. Пусть сигнал задан в виде суммы косинусов с неким набором частот. Как найти амплитуду каждой частотной компоненты? Иными словами, как, имея `ys`, `ts` и `fs`, восстановить `amps`?

В терминах линейной алгебры первый шаг аналогичен синтезу: вычислим $M = \cos(2\pi t \otimes f)$. Затем найдем a, при котором $y = Ma$; иначе – решим линейную систему. В NumPy есть `linalg.solve`, делающая именно это.

Вот как выглядит код:

```
def analyze1(ys, fs, ts):
    args = np.outer(ts, fs)
    M = np.cos(PI2 * args)
    amps = np.linalg.solve(M, ys)
    return amps
```

Первые две строки строят матрицу `M`, используя `ts` и `fs`. Затем `np.linalg.solve` вычисляет `amps`.

Но здесь есть препона. Решить систему линейных уравнений можно, только если матрица квадратная – то есть если число уравнений (строк) совпадает с числом неизвестных (столбцов).

В этом примере только 4 частоты, а выборок из сигнала – 11 025. Поэтому уравнений намного больше, чем неизвестных.

В общем, если в `ys` более 4 элементов, то при наличии всего 4 частот маловероятно, что такой `ys` можно проанализировать.

Но в данном случае известно, что значения `ys` фактически получены как суммы всего 4 частотных компонент, поэтому для восстановления `amps` можно использовать любые 4 значения из массива сигнала.

Для простоты используем первые 4 выборки. Взяв значения `ys`, `fs` и `ts` из предыдущего раздела, запустим `analyze1`:

```
n = len(fs)
amps2 = analyze1(ys[:n], fs, ts[:n])
```

И, конечно, результат `amps2`:

```
[0.6  0.25  0.1  0.05]
```

Этот алгоритм работает, но медленно. Решение системы линейных уравнений требует времени пропорционально n^3, где n – число столбцов в M. Попробуем улучшить ситуацию.

Ортогональные матрицы

Один из способов решения линейных систем – инверсия (обращение) матрицы. Обратная квадратная матрица M записывается M^{-1}, и у нее есть свойство $M^{-1}M = I$. Здесь I – единичная матрица, у которой все диагональные элементы = 1, а остальные = 0.

Таким образом, чтобы решить уравнение $y = Ma$, можно умножить обе части на M^{-1}. Это даст:

$$M^{-1}y = M^{-1}Ma$$

Заменим в правой части $M^{-1}M$ на I:

$$M^{-1}y = Ia$$

Но умножение I на вектор a дает a, поэтому:

$$M^{-1}y = a$$

Значит, если эффективно вычислить M^{-1}, то a найдется простым умножением матриц (используя `np.dot`). Время работы будет пропорционально n^2, что лучше, чем n^3.

Инвертирование матриц вообще – операция медленная, но в некоторых особых случаях – быстрая. В частности, если M *ортогональная*, то матрица, обратная M, – просто транспонированная M, запи-

сываемая M^T. В NumPy транспонирование массива выполняется за неизменное время. Элементы массива не перемещаются, но создается «снимок», изменяющий порядок доступа к элементам.

Опять же, матрица ортогональна, если ее транспонирование будет вместе с тем и ее обращением, то есть $M^T = M^{-1}$. Значит, $M^TM = I$, и проверить, ортогональна ли матрица, можно вычислением M^TM.

Итак, посмотрим, как выглядит матрица в `synthesize2`. В предыдущем примере в M 11 025 строк, поэтому лучше работать с примером поменьше:

```
def test1():
    amps = np.array([0.6, 0.25, 0.1, 0.05])
    N = 4.0
    time_unit = 0.001
    ts = np.arange(N) / N * time_unit
    max_freq = N / time_unit / 2
    fs = np.arange(N) / N * max_freq
    ys = synthesize2(amps, fs, ts)
```

`amps` – это тот же вектор амплитуд, что и выше. Раз есть четыре частотных компоненты, то возьмем выборки сигнала в четырех точках во времени. Соответственно, M будет квадратной.

`ts` – вектор из равноотстоящих моментов времени в диапазоне от 0 до 1. Примем за единицу времени 1 миллисекунду. Это произвольный выбор, и скоро мы увидим, что он никак не влияет на вычисления.

Поскольку частота кадров – N выборок в единицу времени, частота Найквиста в этом примере – `N / time_unit / 2`, или 2000 Гц. Так что `fs` – вектор равноотстоящих частот от 0 до 2000 Гц.

С этими значениями `ts` и `fs` матрица M будет:

```
[[1.      1.      1.      1.    ]
 [1.      0.707   0.     -0.707]
 [1.      0.     -1.     -0.    ]
 [1.     -0.707  -0.      0.707]]
```

Заметим, что 0,707 – это приближенно $\sqrt{2}/2$, или $\cos \pi/4$. Отметим, что эта матрица *симметрична*, и элемент (j,k) всегда равен элементу (k,j). Значит, M транспонируется сама в себя, то есть $M^T = M$.

Но, к сожалению, M не ортогональна. Вычислим M^TM и получим:

```
[[ 4.    1.   -0.    1.]
 [ 1.    2.    1.   -0.]
 [-0.    1.    2.    1.]
 [ 1.   -0.    1.    2.]]
```

И она не единичная матрица.

ДКП-IV

Но M можно сделать ортогональной при тщательном выборе `ts` и `fs`. Для этого есть несколько способов, поэтому существует несколько версий дискретного косинусного преобразования (ДКП).

В простейшем случае можно сдвинуть `ts` и `fs` на пол-единицы. Эта версия называется *ДКП-IV*, где IV – римская цифра, указывающая, что это четвертая из восьми версий ДКП.

Вот обновленная версия `test1`:

```
def test2():
    amps = np.array([0.6, 0.25, 0.1, 0.05])
    N = 4.0
    ts = (0.5 + np.arange(N)) / N
    fs = (0.5 + np.arange(N)) / 2
    ys = synthesize2(amps, fs, ts)
```

По сравнению с предыдущей версией заметны два изменения. Во-первых, к `ts` и `fs` прибавили 0,5. Во-вторых, отключение `time_units` упростило выражение для `fs`.

С этими значениями M будет:

```
[[ 0.981  0.831  0.556  0.195]
 [ 0.831 -0.195 -0.981 -0.556]
 [ 0.556 -0.981  0.195  0.831]
 [ 0.195 -0.556  0.831 -0.981]]
```

А M^TM станет:

```
[[ 2.  0.  0.  0.]
 [ 0.  2. -0.  0.]
 [ 0. -0.  2. -0.]
 [ 0.  0. -0.  2.]]
```

Некоторые из недиагональных элементов отображаются как –0, то есть это малое отрицательное число в представлении с плавающей точкой. Эта матрица очень близка к $2I$; значит, M почти ортогональна, с точностью до 2. И этого достаточно.

Поскольку M симметричная и почти ортогональная, обращенная M будет просто $M/2$. Вот и запишем более эффективную версию `analyze`:

```
def analyze2(ys, fs, ts):
    args = np.outer(ts, fs)
    M = np.cos(PI2 * args)
    amps = np.dot(M, ys) / 2
    return amps
```

Вместо использования `np.linalg.solve` достаточно умножить на $M/2$.

Совместив `test2` и `analyze2`, запишем реализацию ДКП-IV:

```
def dct_iv(ys):
    N = len(ys)
    ts = (0.5 + np.arange(N)) / N
    fs = (0.5 + np.arange(N)) / 2
    args = np.outer(ts, fs)
    M = np.cos(PI2 * args)
    amps = np.dot(M, ys) / 2
    return amps
```

Вновь `ys` – массив сигнала. Передавать `ts` и `fs` как параметры не надо; `dct_iv` учтет их согласно N, длине `ys`.

Если все правильно, эта функция решит проблему анализа; она восстановит `amps` по `ys`. Протестируем ее следующим образом:

```
amps = np.array([0.6, 0.25, 0.1, 0.05])
N = 4.0
ts = (0.5 + np.arange(N)) / N
fs = (0.5 + np.arange(N)) / 2
ys = synthesize2(amps, fs, ts)
amps2 = dct_iv(ys)
max(abs(amps - amps2))
```

Начиная с `amps` синтезируется массив сигнала, а для вычисления `amps2` используется `dct_iv`. Наибольшая разница между `amps` и `amps2` будет `1e-16`, то есть на уровне ошибок округления.

Обратное ДКП

Наконец заметим, что `analyze2` и `synthesize2` почти идентичны. Вся разница в том, что в `analyze2` результат делится на 2. Используем это при вычислении обратного ДКП:

```
def inverse_dct_iv(amps):
    return dct_iv(amps) * 2
```

`inverse_dct_iv` решает проблему синтеза: берется вектор амплитуд и возвращается массив сигнала, `ys`. Протестируем это – начнем с `amps`, применим `inverse_dct_iv` и `dct_iv` и проверим, что получится именно то, с чего начали:

```
amps = [0.6, 0.25, 0.1, 0.05]
ys = inverse_dct_iv(amps)
amps2 = dct_iv(ys)
max(abs(amps - amps2))
```

И опять наибольшее отличие порядка `1e-16`.

Класс Dct

thinkdsp дает класс Dct, который инкапсулирует ДКП так же, как класс Spectrum инкапсулирует БПФ. Чтобы создать объект Dct, можно вызвать make_dct для wave:

```
signal = thinkdsp.TriangleSignal(freq=400)
wave = signal.make_wave(duration=1.0, framerate=10000)
dct = wave.make_dct()
dct.plot()
```

Результатом будет ДКП треугольного сигнала 400 Гц, как показано на рис. 6.2. Значения ДКП могут быть и положительными, и отрицательными; отрицательное значение в ДКП соответствует инверсному косинусу, то есть косинусу, сдвинутому на 180 градусов.

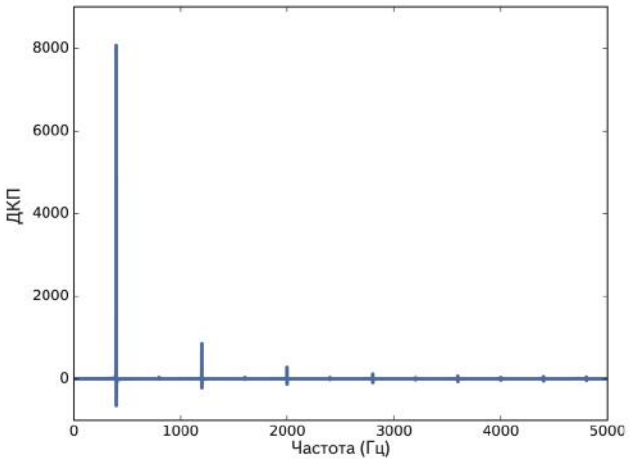

Рис. 6.2. DCT треугольного сигнала 400 Гц, скорость выборки 10 кГц

make_dct использует ДКП-II, самый известный тип ДКП, включенный в scipy.fftpack:

```
import scipy.fftpack

# class wave:
    def make_dct(self):
        N = len(self.ys)
        hs = scipy.fftpack.dct(self.ys, type=2)
        fs = (0.5 + np.arange(N)) / 2
        return Dct(hs, fs, self.framerate)
```

Результаты dct хранятся в hs. Соответствующие частоты, вычисленные по аналогии с примером в разделе «ДКП-IV» (см. стр. 80),

хранятся в fs. И оба массива используются для инициализации объекта Dct.

Dct дает make_wave, выполняющий обратное ДКП. Протестируем эту функцию следующим образом:

```
wave2 = dct.make_wave()
max(abs(wave.ys-wave2.ys))
```

Наибольшая разница между ys1 и ys2 будет 1e-16, на уровне ошибки округления.

make_wave использует scipy.fftpack.idct:

```
# class Dct
    def make_wave(self):
        n = len(self.hs)
        ys = scipy.fftpack.idct(self.hs, type=2) / 2 / n
        return wave(ys, framerate=self.framerate)
```

По умолчанию обратное ДКП не нормализует результат, поэтому надо делить на $2N$.

Упражнения

Небольшой стартовый код представлен в блокноте chap06starter.ipynb.

Решения находятся в chap06soln.ipynb.

Упражнение 6.1

В этой главе утверждается, что analyze1 требует времени пропорционально n^3, а analyze2 – пропорционально n^2. Убедитесь в этом, запуская их с несколькими разными массивами и засекая время работы. В блокнотах Jupyter можно использовать «волшебную команду» **%timeit**.

Если печатать зависимость времени работы от размера на логарифмической шкале, то получится прямая линия с уклоном 3 для analyze1 и с уклоном 2 для analyze2.

Также стоит поупражняться с dct_iv и scipy.fftpack.dct.

Упражнение 6.2

Одно из основных применений ДКП – это *сжатие* звука и изображений. В простейшей форме ДКП при сжатии работает следующим образом:

1. Разбивает длинный сигнал на сегменты.
2. Вычисляет ДКП каждого сегмента.

3. Определяет частотные компоненты с такой амплитудой, что их не слышно, и удаляет их, сохраняя только оставшиеся частоты и амплитуды.
4. При воспроизведении сигнала загружает частоты и амплитуды каждого сегмента и применяет обратное ДКП.

Реализуйте версию этого алгоритма и примените его для записи музыки или речи. Сколько компонент можно удалить до того, как разница станет заметной?

Для того, чтобы этот метод стал практичным, нужен способ хранения *прореженного массива*, то есть массива, где большинство элементов равно нулю. NumPy (SciPy) дает несколько способов работы с прореженными массивами; о них можно прочитать на веб-странице `http://docs.scipy.org/doc/scipy/reference/sparse.html`.

Упражнение 6.3

В репозитории этой книги есть блокнот Jupyter под названием `phase.ipynb`, в котором исследуется влияние фазы на восприятие звука. Прочтите этот блокнот и «погоняйте» примеры. Выберите иной сегмент звука и повторите эксперименты. Можно ли найти некие общие соотношения в фазовой структуре звука и его восприятии?

Глава 7. Дискретное преобразование Фурье

Дискретное преобразование Фурье (ДПФ) уже применялось в главе 1, но его работа не изучалась. Пора этим заняться.

Зная дискретное косинусное преобразование (ДКП), легко понять и ДПФ. Вся разница в том, что вместо косинусов используются комплексные экспоненциальные функции. Рассмотрим комплексные экспоненты, а затем пройдемся по пунктам главы 6:

1. Задача синтеза: как построить сигнал из набора частотных компонент и их амплитуд? Задача синтеза эквивалентна обратному ДПФ.
2. Задача синтеза по-новому: вариант с массивами NumPy и умножением матриц.
3. Задача анализа, эквивалентная ДПФ: расчет амплитуд и фаз частотных компонент в заданном сигнале.
4. Задача с линейной алгеброй: найти эффективный способ вычисления ДПФ.

Код для этой главы находится в репозитории книги, в блокноте `chap07.ipynb` (см. раздел «Работа с кодом» на стр. 9). Также код можно просмотреть на веб-странице `http://tinyurl.com/thinkdsp07`.

Комплексные экспоненты

Один из самых интересных ходов в математике – обобщение операций разных типов. Например, факториал – это функция, работающая с целыми числами; простое определение n-факториал – это произведение всех целых чисел от 1 до n.

Интересно попробовать вычислить факториал дробного числа, скажем 3,5. Так как простое определение неприменимо, поищем способы вычисления функции факториала, работающие с дробными числами.

В 1730 году Леонард Эйлер нашел обобщение функции факториала – это *гамма-функция* (см. http://ru.wikipedia.org/wiki/Гамма-функция).

Эйлер также нашел одно из самых полезных обобщений в прикладной математике – комплексную экспоненциальную функцию.

Простое определение потенцирования – это последовательное умножение, например $\phi^3 = \phi \cdot \phi \cdot \phi$. Но это определение не годится для дробных степеней.

Потенцирование можно выразить в виде степенного ряда:

$$e^{\phi} = 1 + \phi + \phi^2/2! + \phi^3/3! + ...$$

Это определение верно для вещественных чисел, для мнимых чисел и, простым расширением, для комплексных чисел. Применив это определение к чисто мнимому числу $i\phi$, получим:

$$e^{i\phi} = 1 + i\phi - \phi^2/2! - i\phi^3/3! + ...$$

Перестановкой членов можно показать, что это эквивалентно

$$e^{i\phi} = \cos\phi + i\sin\phi$$

Вывод можно посмотреть на веб-странице http://ru.wikipedia.org/wiki/Формула_Эйлера.

Предполагается, что $e^{i\phi}$ – это комплексное число с модулем 1; соответствующая ему точка на комплексной плоскости всегда расположена на единичной окружности. Если представить число как вектор, то угол ϕ между вектором и положительной осью x, выраженный в радианах, – это аргумент.

В случае, когда показатель степени – комплексное число, получим:

$$e^{a+i\phi} = e^a e^{i\phi} = Ae^{i\phi}$$

где A – действительное число, определяющее модуль (амплитуду), а $e^{i\phi}$ – единичное комплексное число, определяющее угол (фазу).

NumPy поддерживает версию `exp`, работающую с комплексными числами:

```
>>> phi = 1.5
>>> z = np.exp(1j * phi)
>>> z
(0.0707+0.997j)
```

Для представления мнимой единицы в Python используется не `i`, а `j`. Число, оканчивающееся на `j`, считается мнимым, так что `1j` – это просто i.

Если аргумент `np.exp` мнимый или комплексный, результат также будет комплексным числом; в частности, `np.complex128` представлено двумя 64-разрядными числами с плавающей запятой. В этом примере результат будет `0.0707 + 0.997j`.

У комплексных чисел есть атрибуты `real` и `imag`:

```
>>> z.real
0,0707
>>> z.imag
0,997
```

Для получения модуля используются встроенные функции `abs` или `np.absolute`:

```
>>> abs(z)
1,0
>>> np.absolute(z)
1,0
```

Для получения угла используется `np.angle`:

```
>>> np.angle(z)
1,5
```

Пример подтверждает, что $e^{i\phi}$ это комплексное число с модулем 1 и углом ϕ радиан.

Комплексные сигналы

Если $\phi(t)$ функция времени, $e^{i\phi(t)}$ – также функция времени. В частности:

$$e^{i\phi(t)} = \cos \phi(t) + i \sin \phi(t)$$

Эта функция описывает величину, изменяющуюся во времени, то есть сигнал. В частности, это *комплексный экспоненциальный сигнал*.

В особом случае, когда частота сигнала постоянна, $\phi(t)$ есть $2\pi ft$, а результат – *комплексная синусоида*:

$$e^{i2\pi ft} = \cos 2\pi ft + i \sin 2\pi ft$$

В общем случае у сигнала может быть ненулевая начальная фаза ϕ_0, что дает:

$$e^{i(2\pi ft + \phi_0)}$$

`thinkdsp` дает реализацию этого сигнала, `ComplexSinusoid`:

```
class ComplexSinusoid(Sinusoid):
    def evaluate(self, ts):
```

```
        phases = PI2 * self.freq * ts + self.offset
        ys = self.amp * np.exp(1j * phases)
        return ys
```

`ComplexSinusoid` наследует `__init__` из `Sinusoid`. Он дает версию `evaluate`, похожую на `Sinusoid.evaluate`; вся разница в использовании `np.exp` вместо `np.sin`.

Результат – NumPy-массив комплексных чисел:

```
>>> signal = thinkdsp.ComplexSinusoid(freq=1, amp=0.6, offset=1)
>>> wave = signal.make_wave(duration=1, framerate=4)
>>> wave.ys
[ 0.324+0.505j -0.505+0.324j -0.324-0.505j 0.505-0.324j]
```

Частота этого сигнала – 1 цикл в секунду, амплитуда – 0,6 (в абстрактных единицах), а фаза – 1 радиан.

В примере сигнал обрабатывается в четырех точках, в интервале от 0 до 1 с. Полученные выборки – комплексные числа.

Задача синтеза

Сложные сигналы можно создавать сложением не только действительных, но и комплексных синусоид с разными частотами. Это задача синтеза в комплексной форме: как оценить сигнал, имея частоты и амплитуды каждой комплексной компоненты?

Простейшее решение – создать объекты `ComplexSinusoid` и сложить их:

```
def synthesize1(amps, fs, ts):
    components = [thinkdsp.ComplexSinusoid(freq, amp)
                  for amp, freq in zip(amps, fs)]
    signal = thinkdsp.SumSignal(*components)
    ys = signal.evaluate(ts)
    return ys
```

Эта функция практически идентична `synthesize1` в разделе «Синтез» (см. стр. 74); вся разница в том, что `CosSignal` заменен на `ComplexSinusoid`.

Вот пример:

```
amps = np.array([0.6, 0.25, 0.1, 0.05])
fs = [100, 200, 300, 400]
framerate = 11025
ts = np.linspace(0, 1, framerate)
ys = synthesize1(amps, fs, ts)
```

Результатом будет:

```
[ 1.000+0.000e+00j  0.995+9.093e-02j  0.979+1.803e-01j ...,
  0.979-1.803e-01j  0.995-9.093e-02j  1.000-5.081e-15j]
```

Попросту говоря, комплексный сигнал – это последовательность комплексных чисел. Но как его интерпретировать? С действительными сигналами все понятно: величины, изменяющиеся во времени; например звуковой сигнал – это изменения давления воздуха. Но обычные измерения никогда не дают комплексных чисел[1].

Так что же это такое – комплексный сигнал? У автора нет исчерпывающего ответа на этот вопрос. Лучшее, что можно предложить, – два частных ответа:

1. Комплексный сигнал есть математическая абстракция, полезная при расчетах и анализе, но напрямую она не соответствует ничему в реальном мире.
2. Комплексный сигнал – это последовательность комплексных чисел, то есть два сигнала, составляющие действительную и мнимую части.

Приняв второй вариант, разделим предыдущий сигнал на действительную и мнимую части:

```
n = 500
thinkplot.plot(ts[:n], ys[:n].real, label='real')
thinkplot.plot(ts[:n], ys[:n].imag, label='imag')
```

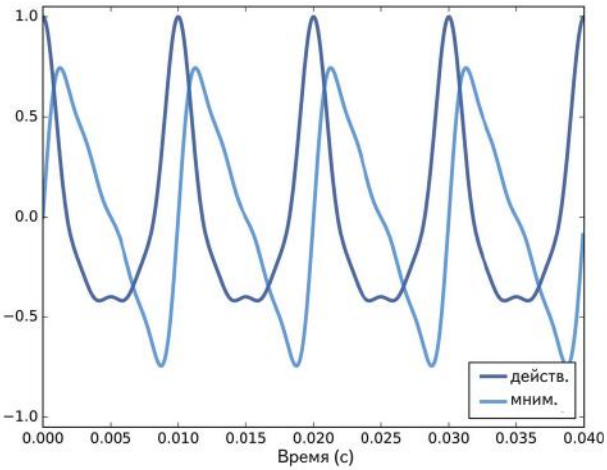

Рис. 7.1. Действительные и мнимые части смеси комплексных синусоид

[1] Автор категорически неправ, но рассмотрение понятия «аналитический сигнал» выходит за рамки этой книги. – *Прим. ред.*

На рис. 7.1 показан сегмент результата. Действительная часть есть сумма косинусов; мнимая часть есть сумма синусов. Сигналы выглядят по-разному, но у них одинаковые частотные компоненты и соотношения между ними. Звучат они одинаково (человек вообще не слышит разности фаз).

Синтез с матрицами

В разделе «Синтез с массивами» на стр. 75 показано, как выразить задачу синтеза в терминах перемножения матриц:

```
PI2 = np.pi * 2

def synthesize2(amps, fs, ts):
    args = np.outer(ts, fs)
    M = np.exp(1j * PI2 * args)
    ys = np.dot(M, amps)
    return ys
```

Вновь `amps` – это NumPy-массив, содержащий последовательность амплитуд.

`fs` – это последовательность, содержащая частотные компоненты. `ts` – это моменты времени, в которые оценивается сигнал.

`args` содержит тензорное произведение `ts` и `fs`, причем `ts` работает вниз по строкам, а `fs` – вдоль столбцов (см. рис. 6.1).

Каждый столбец матрицы `M` содержит комплексные синусоиды с определенной частотой, обрабатываемые в моменты времени из `ts`.

При умножении `M` на амплитуды получится вектор, элементы которого соотносятся с `ts`. Каждый элемент – это сумма нескольких комплексных синусоид, оцененных в соответствующие моменты времени.

Вот пример из предыдущего раздела:

```
>>> ys = synthesize2(amps, fs, ts)
>>> ys
[ 1.000 +0.000e+00j 0.995 +9.093e-02j 0.979 +1.803e-01j ...,
  0.979 -1.803e-01j 0.995 -9.093e-02j 1.000 -5.081e-15j]
```

Результат тот же самый.

В этом примере амплитуды действительные, но они могут быть и комплексными. Как влияет комплексная амплитуда на результат? Помните, что комплексное число можно выразить двумя способами: либо сумма действительной и мнимой частей, $x + iy$, либо произведение действительной амплитуды и комплексной экспоненты, $Ae^{i\phi}$.

Используя второе определение, можно видеть, что происходит при умножении комплексной амплитуды на комплексную синусоиду. Для каждой частоты f получим:

$$Ae^{i\phi_0} \cdot e^{i2\pi ft} = Ae^{i2\pi ft + \phi_0}.$$

Умножение на $Ae^{i\phi_0}$ умножает амплитуду на A и добавляет фазу ϕ_0.

Проверим это утверждение, выполнив предыдущий пример с комплексными амплитудами:

```
phi = 1.5
amps2 = amps * np.exp(1j * phi)
ys2 = synthesize2(amps2, fs, ts)

thinkplot.plot(ts[:n], ys.real[:n])
thinkplot.plot(ts[:n], ys2.real[:n])
```

Так как `amps` – массив действительных чисел, умножение на `np.exp(1j * phi)` возвращает массив комплексных чисел с фазами `phi` радиан и теми же амплитудами, что и `amps`.

На рис. 7.2 показаны сигналы с разными фазами. При $\phi_0 = 1{,}5$ каждая частотная компонента получает сдвиг по фазе примерно на четверть цикла. Однако у компонент с разными частотами разные циклы; в результате каждая компонента сдвигается на разный отрезок времени. После сложения компонент результирующий сигнал выглядит по-другому.

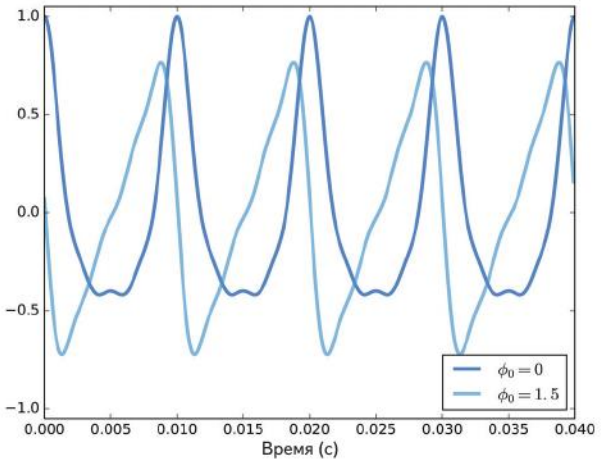

Рис. 7.2. Действительная часть двух комплексных сигналов, отличающихся фазами

Теперь, имея более общее решение задачи синтеза – работающее с комплексными амплитудами, – перейдем к задаче анализа.

Задача анализа

Задача анализа обратна задаче синтеза: имея последовательность образцов, *y*, и зная частоты, имеющиеся в сигнале, как вычислить комплексные амплитуды компонент?

В разделе «Анализ» на стр. 77 показано, как можно решить эту проблему созданием матрицы синтеза *M* и решением системы линейных уравнений, $Ma = y$, для a:

```
def analyze1(ys, fs, ts):
    args = np.outer(ts, fs)
    M = np.exp(1j * PI2 * args)
    amps = np.linalg.solve(M, ys)
    return amps
```

`analyze1` принимает массив (возможно, комплексный) сигнала `ys`, последовательность действительных частот `fs` и последовательность реального времени `ts`. Возвращает последовательность комплексных амплитуд `amps`.

Продолжая предыдущий пример, можно подтвердить, что `analyze1` восстанавливает именно те амплитуды, которые были заданы. Для запуска вычислителя линейной системы `M` должна быть квадратной, поэтому `ys`, `fs` и `ts` должны быть одинаковой длины. Усечем `ys` и `ts` до длины `fs`:

```
>>> n = len(fs)
>>> amps2 = analyze1(ys[:n], fs, ts[:n])
>>> amps2
[ 0.60+0.j 0.25-0.j 0.10+0.j 0.05-0.j]
```

Это очень похоже на то, что было в начале, но у каждой компоненты есть небольшая мнимая часть, появившаяся из-за ошибок округления.

Эффективный анализ

К сожалению, решение системы линейных уравнений идет медленно. ДКП получилось ускорить таким выбором `fs` и `ts`, чтобы `M` стала ортогональной. Тогда обратная `M` аналогична транспонированной `M`; значит, и ДКП, и обратное ДКП можно вычислить перемножением матриц.

Повторим все для ДПФ, но с небольшим изменением. Так как M комплексная, она должна быть *унитарной*, а не ортогональной, так что обратная M будет комплексно-сопряженной к транспонированной M, которую можно вычислить транспонированием матрицы и переменой знака у мнимой части каждого элемента. См. веб-страницу http://ru.wikipedia.org/wiki/Унитарная_матрица.

В NumPy это делают методы conj и transpose. Вот код, вычисляющий M для N = 4 компонент:

```
N = 4
ts = np.arange(N) / N
fs = np.arange(N)
args = np.outer(ts, fs)
M = np.exp(1j * PI2 * args)
```

Если M унитарная, то $M^* M = I$, где M^* сопряженно транспонированная M, а I – единичная матрица. Проверить, унитарна ли M, можно так:

```
MstarM = M.conj().transpose().dot(M)
```

Результат, с точностью до ошибок округления, будет $4I$, поэтому M унитарная с точностью до дополнительного фактора N, похожего на дополнительный фактор 2, который появился в ДКП.

Используем этот результат для записи быстрой версии analyze1:

```
def analyze2(ys, fs, ts):
    args = np.outer(ts, fs)
    M = np.exp(1j * PI2 * args)
    amps = M.conj().transpose().dot(ys) / N
    return amps
```

Протестируем его с соответствующими значениями fs и ts:

```
N = 4
amps = np.array([0.6, 0.25, 0.1, 0.05])
fs = np.arange(N)
ts = np.arange(N) / N
ys = synthesize2(amps, fs, ts)
amps3 = analyze2(ys, fs, ts)
```

И снова результат правильный, с точностью до ошибок округления:

```
[ 0.60+0.j 0.25+0.j 0.10-0.j 0.05-0.j]
```

ДПФ

Функцию analyze2 использовать неудобно, поскольку она работает только при правильно выбранных fs и ts. Перепишем ее так, чтобы брать только ys, а freq и собственно ts вычислять.

Во-первых, сделаем функцию для вычисления матрицы синтеза, `M`:

```
def synthesis_matrix(N):
    ts = np.arange(N) / N
    fs = np.arange(N)
    args = np.outer(ts, fs)
    M = np.exp(1j * PI2 * args)
    return M
```

Затем напишем функцию, которая берет `ys` и возвращает `amps`:

```
def analyze3(ys):
    N = len(ys)
    M = synthesis_matrix(N)
    amps = M.conj().transpose().dot(ys) / N
    return amps
```

Почти все готово; `analyze3` вычисляет что-то близкое к ДПФ с одним лишь отличием. Обычное определение ДПФ не требует деления на *N*:

```
def dft(ys):
    N = len(ys)
    M = synthesis_matrix(N)
    amps = M.conj().transpose().dot(ys)
    return amps
```

Теперь проверим, что эта версия дает такой же результат, что и `np.fft.fft`:

```
>>> dft(ys)
[ 2.4+0.j  1.0+0.j  0.4-0.j  0.2-0.j]
```

Результат близок к `amps * N`. И вот версия в `np.fft`:

```
>>> np.fft.fft(ys)
[ 2.4+0.j  1.0+0.j  0.4-0.j  0.2-0.j]
```

Они одинаковые, в пределах ошибки округления.

Обратное ДПФ будет почти тем же самым, за исключением того что транспонировать и сопрягать `M` не нужно, но *здесь* необходимо делить все на `N`:

```
def idft(ys):
    N = len(ys)
    M = synthesis_matrix(N)
    amps = M.dot(ys) / N
    return amps
```

И наконец, можно подтвердить, что `dft(idft(amps))` дает `amps`:

```
>>> ys = idft(amps)
```

```
>>> dft(ys)
[ 0.60+0.j  0.25+0.j  0.10-0.j  0.05-0.j]
```

Если вернуться назад, то можно изменить определение ДПФ так, что оно на `N` делит, а обратное – не делит. Это более соответствует данному представлению задач синтеза и анализа.

Или можно изменить определение так, чтобы обе операции делили на $\sqrt{N}$. Тогда ДПФ и обратное ДПФ станут более похожими.

Вернуться в прошлое нельзя (пока!), так что придется смириться с немного странным соглашением. Для практических целей это не важно.

Периодичность ДПФ

В этой главе ДПФ представлено в виде перемножения матриц. Вычисляется матрица синтеза M и матрица анализа M^*. При умножении M^* на массив сигнала y каждый элемент результата есть произведение строки из M^* и y, которые можно записать в виде суммы:

$$\mathrm{DFT}(y)[k] = \sum_n y[n] \exp(-2\pi i n k / N),$$

где k – индекс частоты от 0 до $N-1$, а n – индекс времени от 0 до $N-1$. Так что $\mathrm{DFT}(y)[k]$ – это k-й элемент ДПФ от y.

Обычно это сумма N значений k, в порядке от 0 до $N-1$. Можно оценить ее и для иных значений k, но в этом нет смысла, так как они начинают повторяться. Иными словами, значение в k то же, что и в $k+N$ или $k+2N$ или $k-N$, и т. д.

Подтвердим это математически, включив $k+N$ в суммирование:

$$\mathrm{DFT}(y)[k+N] = \sum_n y[n] \exp(-2\pi i n (k+N) / N).$$

Поскольку в показателе степени есть сумма, ее можно разбить на две части:

$$\mathrm{DFT}(y)[k+N] = \sum_n y[n] \exp(-2\pi i n k / N) \exp(-2\pi i n N / N).$$

Во втором члене показатель степени всегда целое кратное 2π, поэтому результат всегда равен 1 и его можно отбросить:

$$\mathrm{DFT}(y)[k+N] = \sum_n y[n] \exp(-2\pi i n k / N).$$

Видно, что это суммирование эквивалентно $\mathrm{DFT}(y)[k]$. Таким образом, ДПФ периодично, с периодом N. Этот результат понадобится

вам в одном из упражнений в конце главы: там надо будет реализовать быстрое преобразование Фурье (БПФ).

Заметим, что запись ДПФ в виде суммы дает понять, как оно работает. Если посмотреть на диаграмму в разделе «Синтез с массивами» (см. стр. 75), видно, что каждый столбец матрицы синтеза сигнала обрабатывается в определенной временной последовательности. Матрица анализа – это сопряженно транспонированная матрица синтеза, поэтому каждая строка – это сигнал, также вычисляемый в определенной временной последовательности.

Таким образом, каждая сумма – это корреляция y с одним из сигналов в массиве (см. раздел «Корреляция как скалярное произведение» на стр. 70). То есть каждый элемент ДПФ представляет собой корреляцию, которая дает количественную оценку похожести массива сигнала y и комплексной экспоненты на определенной частоте.

ДПФ реальных сигналов

Класс `Spectrum` в `thinkdsp` основан на `np.ftt.rfft`, вычисляющей «действительное ДПФ»; то есть работающей с реальными сигналами. ДПФ, представленное в настоящей главе, более общий случай, – оно работает с комплексными сигналами.

Что же произойдет, если «полное ДПФ» применить к реальному сигналу? Рассмотрим пример:

```
signal = thinkdsp.SawtoothSignal(freq=500)
wave = signal.make_wave(duration=0.1, framerate=10000)
hs = dft(wave.ys)
amps = np.absolute(hs)
```

Этот код создает пилообразный сигнал с частотой 500 Гц, выборки берутся с частотой кадров 10 кГц. `hs` содержит комплексное ДПФ сигнала, `amps` – амплитуды на каждой частоте. Но каким частотам соответствуют эти амплитуды? Посмотрим на тело `dft`:

```
fs = np.arange(N)
```

Кажется, что эти значения соответствуют правильным частотам. Но проблема в том, что `dft` не учитывает частоту выборки. В ДПФ предполагается, что длительность сигнала – это одна единица времени, и считается, что частота дискретизации – N в единицу времени. Получить правильные частоты можно, преобразовав абстрактные единицы времени в секунды следующим образом:

```
fs = np.arange(N) * framerate / N
```

С этим изменением диапазон частот станет от 0 до частоты кадров, 10 кГц. Теперь можно напечатать спектр:

```
thinkplot.plot(fs, amps)
thinkplot.config(xlabel='frequency (Hz)',
                 ylabel='amplitude')
```

На рис. 7.3 показаны амплитуды сигнала для каждой частотной компоненты, от 0 до 10 кГц. В левой части рисунка все ожидаемо: доминирующая частота на отметке 500 Гц и гармоники, спадающие пропорционально $1/f$.

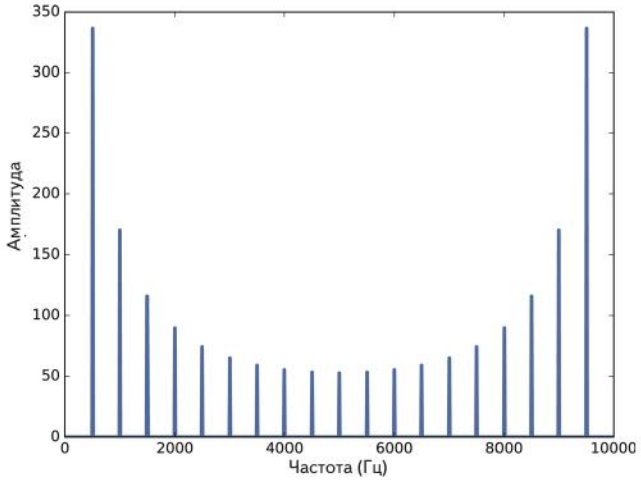

Рис. 7.3. ДПФ 500 Гц пилообразного сигнала с выборками на частоте 10 кГц

А в правой части рисунка – сюрприз! После 5000 Гц амплитуда гармоник начинает снова расти, с пиком на 9500 Гц. Что происходит?

Ответ: биения (алиасинг). Помните, что при частоте кадров 10 000 Гц частота заворота – 5000 Гц. Как показано в разделе «Сглаживание» на стр. 99, компонента на 5500 Гц ничем не отличается от компоненты на 4500 Гц. Значение ДПФ на 5500 Гц то же, что и на 4500 Гц. Аналогичным образом значение на 6000 Гц то же, что и на 4000 Гц, и т. д.

ДПФ реального сигнала симметрично вокруг частоты заворота. После нее нет никакой полезной информации, поэтому можно сэкономить время, оценивая только первую половину ДПФ; именно так и работает `np.fft.rfft`.

Упражнения

Решения этих упражнений находятся в блокноте `chap07soln.ipynb`.

Упражнение 7.1

В блокноте для этой главы, `chap07.ipynb`, представлены дополнительные примеры и пояснения. Прочитайте блокнот и запустите код.

Упражнение 7.2

В этой главе показано, как выразить ДПФ и обратное ДПФ произведениями матриц. Время выполнения этих операций пропорционально N^2, где N – длина массива сигнала. Во многих случаях этого достаточно, но есть алгоритм и побыстрее – быстрое преобразование Фурье (БПФ); время его работы пропорционально $N \log N$.

Ключ к БПФ – это лемма Дэниелсона–Ланцоша (Danielson–Lanczos):

$$\mathrm{DFT}(y)[n] = \mathrm{DFT}(e)[n] + \exp(-2\pi i n/N)\ \mathrm{DFT}(o)[n],$$

где $\mathrm{DFT}(y)[n]$ – это n-й элемент ДПФ от y, e и o – массивы сигнала, содержащие соответственно четные и нечетные элементы y.

Эта лемма предлагает рекурсивный алгоритм для ДПФ:

1. Дан массив сигнала y. Разделим его на четные элементы e и нечетные элементы o.
2. Вычислим DFT e и o, делая рекурсивные вызовы.
3. Вычислим $\mathrm{DFT}(y)$ для каждого значения n, используя лемму Дэниелсона–Ланцоша.

В простейшем случае эту рекурсию надо продолжать, пока длина y не дойдет до 1. Тогда $\mathrm{DFT}(y) = y$. А если длина y достаточно мала, можно вычислить его ДПФ перемножением матриц, используя заранее вычисленные матрицы.

Подсказка: реализовывать этот алгоритм стоит постепенно, начав с нерекурсивной версии. На шаге 2, вместо того чтобы делать рекурсивный вызов, используйте `dft`, как показано в разделе «ДПФ» на стр. 93, или `np.fft.fft`. Отладьте шаг 3 и проверьте, согласуются ли результаты с другими реализациями. Затем добавьте базовый случай и убедитесь, что он работает. И наконец, замените шаг 2 на рекурсивные вызовы.

Примечание: помните, что ДПФ периодично; попробуйте применить `np.tile`.

О БПФ можно прочитать подробнее в Википедии: `https://ru.wikipedia.org/wiki/Быстрое_преобразование_Фурье`.

Глава 8. Фильтрация и свертка

В этой главе будет рассмотрена одна из важных и полезных идей в обработке сигналов: теорема о свертке. Сначала изучим саму свертку, а потом перейдем к теореме. Начнем с простейшего случая – со сглаживания, затем приступим к сложным.

Код для этой главы находится в репозитории книги, в блокноте `chap08.ipynb` (см. раздел «Работа с кодом» на стр. 9). Также код можно просмотреть на веб-странице `http://tinyurl.com/thinkdsp08`.

Сглаживание

Сглаживание – это операция, удаляющая быстрые изменения сигнала для выявления общих особенностей. Например, график ежедневных изменений в цене акций будет выглядеть «зашумленным», а сглаживающий оператор сделает его удобным для понимания тенденции цен – растут они с течением времени или падают.

Обычный алгоритм сглаживания называют *скользящим средним* – оно дает среднее из предыдущих n значений для некоторого n.

Например, на рис. 8.1 показаны ежедневные цены закрытия акций Facebook с 17.05.2012 по 08.12.2015. Серая линия – необработанные данные, а темная – скользящее среднее за 30 дней. Сглаживание удаляет самые резкие изменения и упрощает оценку долгосрочных тенденций.

Сглаживание применяется и к звуковым сигналам. Например, начнем с прямоугольного сигнала 440 Гц. В разделе «Прямоугольный сигнал» на стр. 27 показано, что гармоники прямоугольного сигнала спадают медленно, в нем много высокочастотных компонент.

Создадим сигнал и два массива `wave`:

```
signal = thinkdsp.SquareSignal(freq=440)
wave = signal.make_wave(duration=1, framerate=44100)
segment = wave.segment(duration=0.01)
```

`wave` – это отрезок сигнала длительностью 1 с; `segment` – это короткая часть, для удобства вывода графиков.

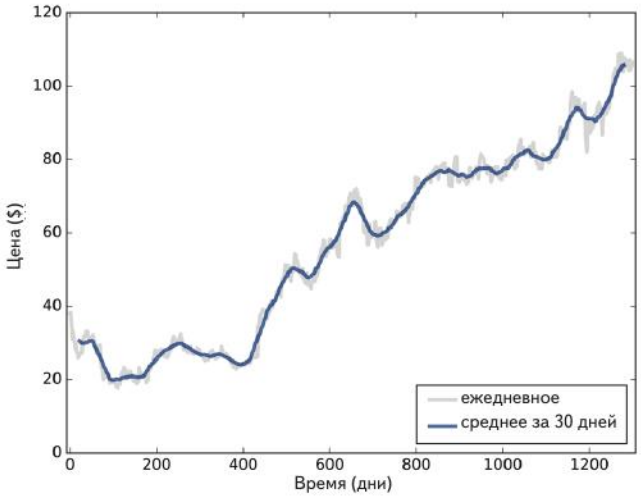

Рис. 8.1. Ежедневные цены закрытия акций Facebook и их скользящего среднего за 30 дней

Вычислим скользящее среднее этого сигнала, используя окно, похожее на те, что описывались в разделе «Окна» на стр. 42. Ранее использовалось окно Хэмминга – для исключения утечки спектра, вызванной разрывом между началом и концом сигнала. В общем, для расчета взвешенной суммы выборок из сигнала можно использовать окна.

Например, для вычисления скользящего среднего создадим окно в 11 элементов и нормализуем его так, чтобы сумма элементов росла до 1:

```
window = np.ones(11)
window /= sum(window)
```

Теперь вычислим среднее от первых 11 элементов путем умножения окна на массив сигнала:

```
ys = segment.ys
N = len(ys)
padded = thinkdsp.zero_pad(window, N)
prod = padded * ys
sum(prod)
```

`padded` – это версия окна с нулями, добавленными в конец, чтобы его длина совпала с длиной `segment.ys`. *Дополнение нулями* по-английски называется *padding*.

`prod` – это произведение окна на массив сигнала. Поэлементная сумма произведений – это среднее значение первых 11 элементов массива. Все эти элементы равны 1, и их среднее – также 1.

Для вычисления следующего элемента скользящего среднего «провернем» окно – сдвинем единицы вправо и переставим из конца в начало один из нулей.

Вновь умножим окно на массив сигнала и получим среднее следующих 11 элементов массива сигнала начиная со второго:

```
rolled = np.roll(rolled, 1)
prod = rolled * ys
sum(prod)
```

Результат – снова 1.

Остальные элементы вычислим аналогично. Ниже дана функция, организующая приведенный выше код в цикл и сохраняющая результаты в массив:

```
def smooth(ys, window):
    N = len(ys)
    smoothed = np.zeros(N)
    padded = thinkdsp.zero_pad(window, N)
    rolled = padded

    for i in range(N):
        smoothed[i] = sum(rolled * ys)
        rolled = np.roll(rolled, 1)
    return smoothed
```

`smoothed` – это массив, содержащий результат; `padded` – массив, содержащий окно и достаточное количество нулей, выравнивающее длину до `N`; `rolled` – копия `padded`, сдвигаемая вправо на один элемент за каждый проход цикла.

Внутри цикла `ys` умножается на `rolled` для выбора 11 элементов и их сложения.

На рис. 8.2 показан результат для прямоугольного сигнала. Серая линия – оригинальный сигнал; темная – сглаженный. Сглаженный сигнал начинает нарастать, когда начало окна достигает первой смены знака, и спадать, когда окно заканчивается. В результате переходы менее резкие, а углы тупые. При прослушивании сглаженный сигнал кажется глуше и звучит менее «гряз-з-зно».

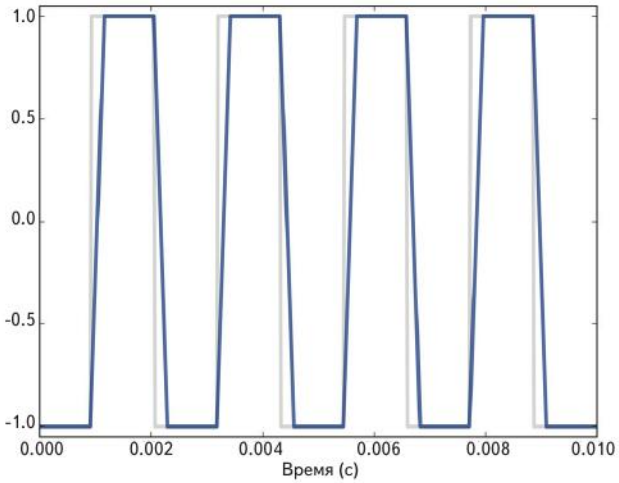

Рис. 8.2. Прямоугольный сигнал на 400 Гц (серый) и 11-элементное скользящее среднее

Свертка

Выполненная выше операция – применение оконной функции к перекрывающимся сегментам сигнала – и есть *свертка*.

Свертка – настолько распространенная операция, что NumPy дает реализацию, которая быстрее и проще, чем версия автора:

```
convolved = np.convolve(ys, window, mode='valid')
smooth2 = thinkdsp.wave(convolved, framerate=wave.framerate)
```

`np.convolve` вычисляет свертку массива сигнала с окном. Флаг режима `valid` указывает, что следует вычислять значения только при полном перекрытии массива сигнала и окна, поэтому он останавливается, когда правый край окна достигает конца массива сигнала. В остальном результат аналогичен показанному на рис. 8.2.

На самом деле есть еще одно отличие. В предыдущем разделе цикл вычисляет *кросс-корреляцию*:

$$(f \star g)[n] = \sum_{m=0}^{N-1} f[m]\, g[n+m],$$

где f – массив сигнала длиной N, g – это окно, $\star$ – символ кросс-корреляции. Для расчета n-го элемента результата g сдвигается вправо, поэтому индекс станет $n + m$.

Определение свертки немного иное:

$$(f * g)[n] = \sum_{m=0}^{N-1} f[m]\, g[n-m].$$

Символ * обозначает свертку. Разница состоит в индексе при g: m отрицательное, и суммирование выполняется в порядке, обратном элементам g (можно сказать, что отрицательные индексы «завернулись» на конце массива).

Окно, использованное в этом примере, симметричное, поэтому кросс-корреляция и свертка дают одинаковый результат. С другими окнами надо быть осторожнее.

Почему свертка определена так, что окно накладывается с конца? Тому две причины:

- данное определение естественно во многих областях, особенно в анализе систем обработки сигналов, и это тема главы 10;
- данное определение также является и основой теоремы о свертке, к которой мы подошли уже совсем близко.

А вот замечание для всезнаек: до сих пор не показаны различия между сверткой и круговой сверткой. Об этом – ниже.

Частотная область

Сглаживание делает фронты в прямоугольных сигналах менее крутыми, а звук – слегка приглушенным. Рассмотрим, как эта операция влияет на спектр. Сначала напечатаем участок спектра оригинального сигнала:

```
Spectrum = wave.make_Spectrum()
Spectrum.plot(color=GRAY)
```

Затем сглаженный сигнал:

```
convolved = np.convolve(wave.ys, window, mode='same')
smooth = thinkdsp.wave(convolved, framerate=wave.framerate)
spectrum2 = smooth.make_spectrum()
spectrum2.plot()
```

Флаг режима `same` указывает, что результат должен быть той же длины, что и входные данные. В этом примере у него будет несколько значений, которые «завернутся», но пока это не страшно.

На рис. 8.3 показан результат. Основная частота осталась без изменений; первые несколько гармоник ослаблены, а высшие гармоники практически ликвидированы. Так что сглаживание похоже на фильтр

НЧ, рассмотренный в разделах «Спектры» (стр. 20) и «Розовый шум» (стр. 55).

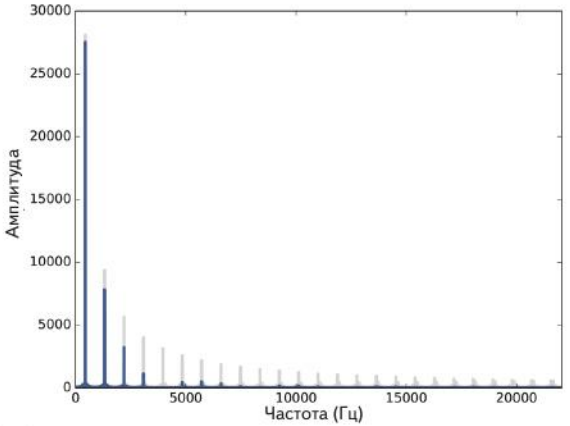

Рис. 8.3. Спектр прямоугольного сигнала до и после сглаживания

Посмотрим, насколько ослаблена каждая компонента – вычислим соотношение двух спектров:

```
amps = Spectrum.amps
amps2 = Spectrum2.amps
ratio = amps2 / amps
ratio[amps<560] = 0
thinkplot.plot(ratio)
```

`ratio` – это соотношение амплитуд до и после сглаживания. При малых `amps` оно может быть большим и шумным, поэтому для простоты стоит обнулить все кроме самих гармоник.

На рис. 8.4 показан результат. Как и ожидалось, соотношение велико для НЧ и мало за частотой среза, в районе 4000 Гц. Но есть одна неожиданная особенность – выше частоты среза соотношение «скачет» от 0 до 0,2. Что это?

Теорема о свертке

Ответ прост – это *теорема о свертке*. Математически говоря,

$$\mathrm{DFT}(f * g) = \mathrm{DFT}(f) \cdot \mathrm{DFT}(g),$$

где f – массив сигнала, а g – окно. Теорема о свертке гласит, что ДПФ от свертки f и g дает тот же результат, что и поэлементное перемножение ДПФ f и ДПФ g.

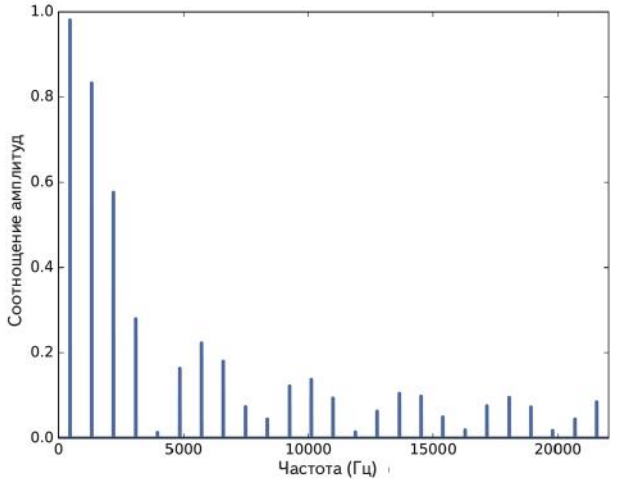

Рис. 8.4. Соотношение спектров прямоугольного сигнала до и после сглаживания

Применение операции типа свертки к сигналу подразумевает работу во временной области, ибо сигнал – функция времени. Применение операции типа умножения к ДПФ подразумевает работу в частотной области, ибо ДПФ – функция частоты.

Теперь сформулируем теорему о свертке лаконичнее:

Свертка во временной области соответствует умножению в частотной области.

И это объясняет рис. 8.4, поскольку при свертке сигнала с окном спектр сигнала умножается на спектр окна. Для изучения этого явления вычислим ДПФ окна:

```
padded = zero_pad(window, N)
dft_window = np.fft.rfft(padded)
thinkplot.plot(abs(dft_window))
```

`padded` – это сглаживающее окно, дополненное нулями до той же длины, что и `wave`; `dft_window` – это ДПФ от окна `padded`.

На рис. 8.5 показан результат, а также соотношения, вычисленные в предыдущем разделе. Они точно соответствуют амплитудам в `dft_window`. Математически:

$$abs(\mathrm{DFT}(f * g)) \,/\, abs(\mathrm{DFT}(f)) = abs(\mathrm{DFT}(g)).$$

В этом смысле ДПФ окна называется *фильтром*. Для любого сверточного окна во временной области существует соответствующий

фильтр в частотной области. И для любого фильтра, выражаемого через поэлементное умножение в частотной области, есть соответствующее окно.

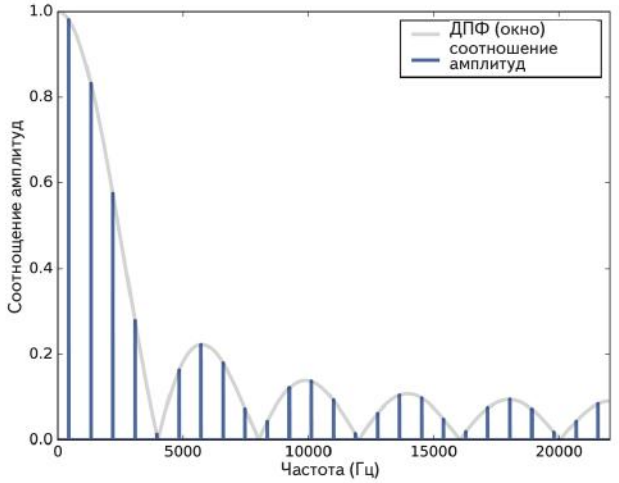

Рис. 8.5. Соотношение спектров прямоугольного сигнала до и после сглаживания и ДПФ окна сглаживания

Гауссов фильтр

Окно скользящего среднего, изученное в предыдущем разделе, есть фильтр НЧ, но он не очень хорош. ДПФ сначала резко падает, а потом заметно колеблется. Эти колебания называют *боковыми лепестками*, а появляются они оттого, что окно скользящего среднего похоже на прямоугольный сигнал и высокочастотные гармоники в его спектре спадают пропорционально $1/f$, то есть медленно.

Лучше будет гауссово окно. SciPy дает функции, вычисляющие много разных сверточных окон, в частности `gaussian`:

```
gaussian = scipy.signal.gaussian(M=11, std=2)
gaussian /= sum(gaussian)
```

`M` – это количество элементов в окне; `std` – стандартное отклонение гауссова распределения, используемое для его вычисления. На рис. 8.6 показана форма окна: это дискретное приближение гауссова «колокола». На том же рисунке показано окно скользящего среднего из предыдущего примера, иногда называемое *прямоугольным*[1].

[1] В амер. лит. – boxcar («товарный вагон»). – *Прим. ред.*

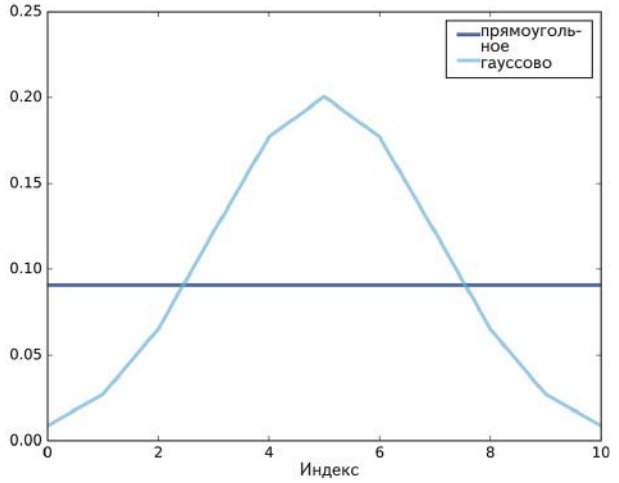

Рис. 8.6. Прямоугольное и гауссово окна

Повтор вычислений из предыдущих разделов уже с этим окном дает результаты, приведенные на рис. 8.7. Показаны ДПФ гауссова окна и соотношение спектров до и после сглаживания.

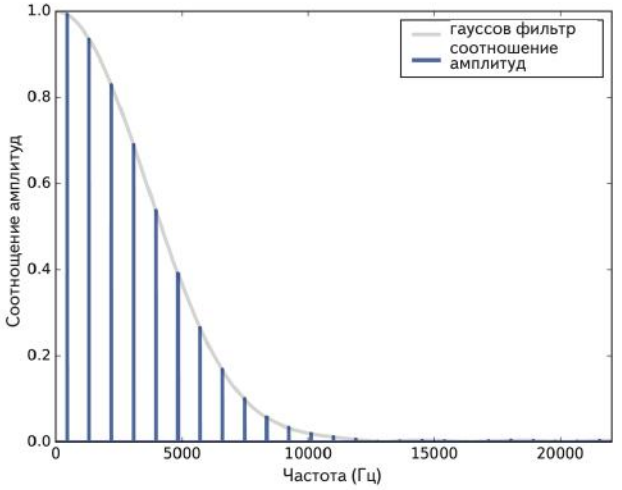

Рис. 8.7. Соотношение спектров до и после гауссова сглаживания и ДПФ окна

Гауссово сглаживание как фильтр НЧ лучше скользящего среднего. После спада соотношение остается малым, почти без боковых

лепестков, свойственных прямоугольному окну. Так что оно лучше подавляет высокие частоты.

Оно работает так хорошо потому, что ДПФ гауссовой кривой – также гауссова кривая. И соотношение спадает пропорционально $\exp(-f^2)$, а это намного быстрее, чем $1/f$.

Эффективная свертка

Одна из причин важности алгоритма БПФ в том, что в сочетании с теоремой о свертке он дает эффективный способ вычисления свертки, кросс-корреляции и автокорреляции.

Теорема о свертке гласит:

$$\mathrm{DFT}(f * g) = \mathrm{DFT}(f) \cdot \mathrm{DFT}(g).$$

Поэтому один из способов вычисления свертки:

$$f * g = \mathrm{IDFT}(\mathrm{DFT}(f) \cdot \mathrm{DFT}(g)),$$

где IDFT – обратное ДПФ. Простая реализация свертки требует времени пропорционально N^2; алгоритм с БПФ требует времени пропорционально $N \log N$.

Проверим это, вычисляя свертки обоими способами. Для примера возьмем биржевые сводки Facebook, показанные на рис. 8.1:

```
import pandas as pd

names = ['date', 'open', 'high', 'low', 'close', 'volume']
df = pd.read_csv('fb.csv', header=0, names=names)
ys = df.close.values[::-1]
```

В этом примере используется Pandas для чтения данных из CSV-файла (входит в репозиторий книги). Знать Pandas вам не обязательно: в этой книге оно не пригодится. Но если есть интерес к теме, вы найдете полезную информацию в Think Stats: `http://thinkstats2.com`.

Результат, `df`, – это `DataFrame`, одна из структур данных, поддерживаемых Pandas. `close` – это NumPy массив, содержащий ежедневные цены закрытия.

Далее создадим гауссово окно и свернем его с `close`:

```
window = scipy.signal.gaussian(M=30, std=6)
window /= window.sum()
smoothed = np.convolve(ys, window, mode='valid')
```

`fft_convolve` вычисляет то же самое, используя БПФ:

```
from np.fft import fft, ifft

def fft_convolve(signal, window):
    fft_signal = fft(signal)
    fft_window = fft(window)
    return ifft(fft_signal * fft_window)
```

Проверим это заполнением окна до той же длины, что и `ys`, а затем вычислим свертку:

```
padded = zero_pad(window, N)
smoothed2 = fft_convolve(ys, padded)
```

У результата в начале есть $M - 1$ фиктивных значений, где M – длина окна. Отсекаются они следующим образом:

```
M = len(window)
smoothed2 = smoothed2[M-1:]
```

Результат совпадает с `fft_convolve` с точностью порядка 12 разрядов.

Эффективная автокорреляция

В разделе «Свертка» на стр. 102 приводились определения кросс-корреляции и свертки; было показано, что они почти одинаковы, за исключением того что в свертке окно обращено.

Теперь, когда есть эффективный алгоритм свертки, используем его для вычисления кросс-корреляции и автокорреляции. Вычислим автокорреляцию цен на акции Facebook на основе данных из предыдущего раздела:

```
corrs = np.correlate(close, close, mode='same')
```

При `Mode ='same'` длина результата та же, что и у `close`, сообразно лагам от $-N/2$ до $N/2 - 1$. Серая линия на рис. 8.8 показывает результат. За исключением точки `lag` = 0 пиков не наблюдается, поэтому явной периодичности в поведении этого сигнала нет. Однако автокорреляционная функция спадает медленно, а значит, этот сигнал напоминает розовый шум, как показано в разделе «Автокорреляция» на стр. 65.

Для вычисления автокорреляции с применением свертки надо дополнить сигнал нулями до удвоения длины. Это необходимо, поскольку БПФ основано на предположении периодичности сигнала – как будто он свернут от конца к началу. Для данных реального времени это предположение негодное. Добавление нулей и последующее усечение результатов удаляет фиктивные значения.

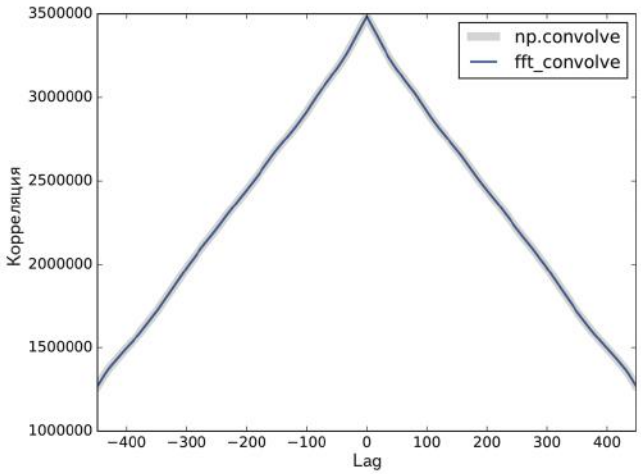

Рис. 8.8. Автокорреляционные функции, рассчитанные NumPy и `fft_autocorr`

Напомним, что свертка обращает направление окна. Учтем это и изменим направление окна перед вызовом `fft_convolve`, для чего используем `np.flipud`, разворачивающий массив NumPy. Результат – вид на массив, а не копия, так что это быстрая операция:

```
def fft_autocorr(signal):
    N = len(signal)
    signal = thinkdsp.zero_pad(signal, 2*N)
    window = np.flipud(signal)

    corrs = fft_convolve(signal, window)
    corrs = np.roll(corrs, N//2+1)[:N]
    return corrs
```

Длина результата `fft_convolve` будет $2N$. Из них первые и последние $N/2$ значимы; остальные – результат дополнения нулями. Значащие элементы получаем «прокручиванием» результата, выбирая первые N, соответствующие лагам от $-N/2$ до $N/2 - 1$.

Как показано на рис. 8.8, результаты `fft_autocorr` и `np.correlate` идентичны (с точностью до 9 разрядов).

Обратите внимание, что корреляции на рис. 8.8 – большие числа; их можно нормализовать (между -1 и 1) так, как показано в разделе «Использование NumPy» на стр. 70.

Стратегия, использованная здесь для автокорреляции, работает и для кросс-корреляции. И снова надо подготовить сигналы, развернув один и дополнив нулями оба, а затем отсечь незначащие части ре-

зультата. Дополнение и отсечение не очень удобны; именно поэтому библиотеки типа NumPy предоставляют функции, делающие это за вас.

Упражнения

Решения этих упражнений находятся в блокноте `chap08soln.ipynb`.

Упражнение 8.1

Блокнот для этой главы – `chap08.ipynb`. Прочитайте его и запустите код.

В нем есть интерактивный виджет, где можно экспериментировать с параметрами гауссова окна и изучить их влияние на частоту среза.

Что случится, если при увеличении ширины гауссова окна `std` не увеличивать число элементов в окне `M`?

Упражнение 8.2

В этой главе утверждается, что преобразование Фурье гауссовой кривой – также гауссова кривая. Для дискретного преобразования Фурье это сотношение приблизительно верно.

Попробуйте его на нескольких примерах. Что происходит с преобразованием Фурье, если меняется `std`?

Упражнение 8.3

В упражнениях к главе 3 изучалось влияние на утечки спектра окна Хэмминга и некоторых других, предоставляемых NumPy. Глубже понять эти окна можно, изучив их ДПФ.

В дополнение к Гауссову окну, использованному в этой главе, создайте окно Хемминга тех же размеров. Дополните окно нулями и напечатайте его ДПФ. Какое окно больше подходит для фильтра НЧ? Полезно напечатать ДПФ с логарифмическим масштабом по y.

Поэкспериментируйте с разными окнами и разными размерами этих окон.

Глава 9. Дифференцирование и интегрирование

В этой главе мы продолжим рассматривать соотношения между окнами во временной области и фильтрами – в частотной.

В частности, будет показана работа окна конечной разности, аппроксимирующего дифференцирование, и операции накапливающей суммы, аппроксимирующей интегрирование.

Код для этой главы находится в репозитории книги, в блокноте `chap09.ipynb` (см. раздел «Работа с кодом» на стр. 9). Также код можно просмотреть на веб-странице `http://tinyurl.com/thinkdsp09`.

Конечные разности

В разделе «Сглаживание» на стр. 99 сглаживающее окно было наложено на ежедневные цены акций Facebook и мы показали, что окно сглаживания во временной области соответствует фильтру НЧ в частотной области.

В этом разделе рассмотрим ежедневные изменения цен и покажем, что вычисление разницы между соседними элементами во временной области соответствует фильтру ВЧ.

Вот код для чтения данных – сохраним его как сигнал и вычислим его спектр:

```
import pandas as pd

names = ['date', 'open', 'high', 'low', 'close', 'volume']
df = pd.read_csv('fb.csv', header=0, names=names)
ys = df.close.values[::-1]
close = thinkdsp.wave(ys, framerate=1)
spectrum = wave.make_spectrum()
```

В этом примере используется Pandas для чтения файла CSV; результат – `DataFrame`, `df`, причем в столбцах – цены открытия, закры-

тия, максимумы и минимумы цен. Выберем цены закрытия и сохраним их в объекте `wave`. Частоту кадров установим равной 1 выборке в день.

На рис. 9.1 показаны временной ряд и его спектр. Временной ряд по виду похож на броуновский шум (см. раздел «Броуновский шум» на стр. 52). А спектр выглядит как прямая линия, хотя и размытая. Средний уклон –1,9, что согласуется с броуновским шумом.

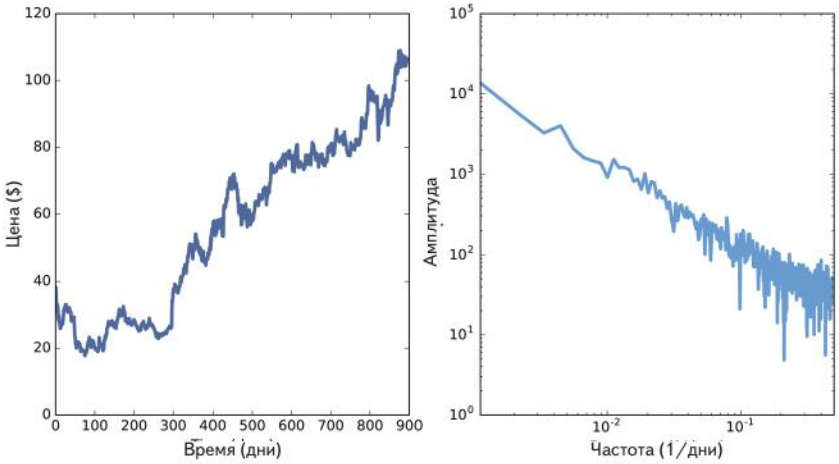

Рис. 9.1. Ежедневные цены закрытия акций Facebook и спектр этого временного ряда

Теперь вычислим ежедневные изменения цены, используя `np.diff`:

```
diff = np.diff(ys)
change = thinkdsp.wave(diff, framerate=1)
change_spectrum = change.make_spectrum()
```

На рис. 9.2 показаны результирующий сигнал и его спектр. Ежедневные изменения напоминают белый шум, а оценка уклона спектра, –0,06, близка к нулю, что ожидаемо для белого шума.

Частотная область

Вычисления разницы между соседними элементами – это то же, что и свертка с окном `[1, -1]`. Порядок элементов обращен, поскольку свертка обращает окно перед наложением его на сигнал.

Рассмотрим эффект этой операции в частотной области, вычислив ДПФ окна:

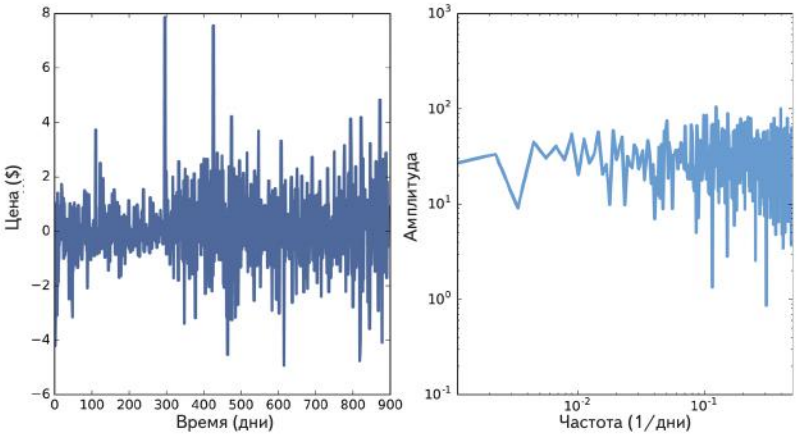

Рис. 9.2. Ежедневные цены закрытия акций Facebook и спектр этого временного ряда

```
diff_window = np.array([1.0, -1.0])
padded = thinkdsp.zero_pad(diff_window, len(close))
diff_wave = thinkdsp.wave(padded, framerate=close.framerate)
diff_filter = diff_wave.make_spectrum()
```

На рис. 9.3 показан результат. Окно конечной разности соответствует фильтру ВЧ: его амплитуда линейно растет в области низких частот, но скорость роста постепенно снижается. Причину мы выясним в следующем разделе.

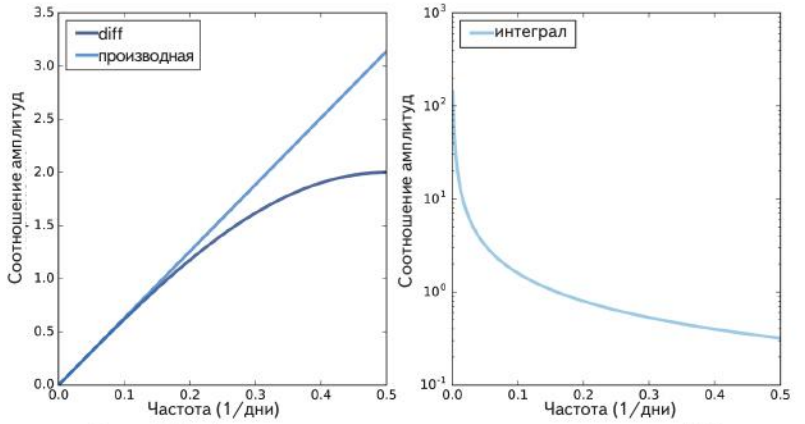

Рис. 9.3. Фильтры, соответствующие операторам diff и differentiate (слева) и оператору интегрирования (справа, логарифмический масштаб по *y*)

Дифференцирование

Окно, использованное в предыдущем разделе, есть численное приближение первой производной, поэтому фильтр аппроксимирует дифференцирование.

Дифференцирование во временной области соответствует простому фильтру в частотной области; математика поможет с этим разобраться.

Возьмем комплексную синусоиду с частотой f:

$$E_f(t) = e^{2\pi i f t}.$$

Первая производная от E_f будет:

$$d/dt\,(E_f(t)) = 2\pi i f\, e^{2\pi i f t},$$

что можно переписать следующим образом:

$$d/dt\,(E_f(t)) = 2\pi i f\, E_f(t).$$

Иначе говоря, взятие производной от E_f соответствует умножению на $2\pi if$, то есть на комплексное число с модулем $2\pi f$ и углом $\pi/2$.

Вычислим фильтр, соответствующий дифференцированию:

```
deriv_filter = close.make_spectrum()
deriv_filter.hs = PI2 * 1j * deriv_filter.fs
```

Начнем со спектра `close` с правильным размером и частотой кадров, а затем заменим `hs` на $2\pi if$. На рис. 9.3 (слева) показан этот фильтр; он представляет собой прямую линию.

В разделе «Синтез с матрицами» на стр. 90 мы говорили о двойственности умножения комплексной синусоиды на комплексное число: амплитуда в данном случае умножается на $2\pi f$ и сдвигается фаза – в нашем примере на $\pi/2$.

В понятиях операторов и собственных функций каждое E_f есть собственная функция оператора дифференцирования, с соответствующим собственным значением $2\pi if$. См. статью в Википедии `http://en.wikipedia.org/wiki/Eigenfunction`.

Для тех, кто не знаком с этими понятиями, поясним:

- Оператор – это функция, берущая одну функцию и возвращающая другую. Например, дифференцирование – это оператор.
- Функция g есть собственная функция оператора $\mathcal{A}$, если применение $\mathcal{A}$ к g соответствует умножению функции на скаляр. То есть $\mathcal{A}g = \lambda g$.

- В этом случае скаляр λ есть собственное значение, соответствующее собственной функции g.
- Данный оператор может иметь много собственных функций, каждая со своим собственным значением.

Поскольку комплексные синусоиды – собственные функции оператора дифференцирования, они легко дифференцируемы. Достаточно лишь умножения на комплексный скаляр.

Для сигналов со многими компонентами процесс ненамного сложнее:

1. Выразите сигнал совокупностью комплексных синусоид.
2. Вычислите производную каждой компоненты умножением.
3. Сложите продифференцированные компоненты.

Этот алгоритм очень похож на алгоритм свертки в разделе «Эффективная свертка» на стр. 108: вычислить ДПФ, умножить на фильтр и вычислить обратное ДПФ.

`Spectrum` дает метод, накладывающий дифференцирующий фильтр.

```
# class Spectrum:

    def differentiate(self):
        self.hs *= PI2 * 1j * self.fs
```

Его можно использовать для вычисления производной временного ряда Facebook:

```
deriv_spectrum = close.make_spectrum()
deriv_spectrum.differentiate()
deriv = deriv_spectrum.make_wave()
```

На рис. 9.4 сравниваются рассчитанные `np.diff` ежедневные изменения цен с вычисленной выше производной. Взяты первые 50 значений во временном ряду, чтобы различия были четко видны.

Производная шумит сильнее, потому что она усиливает высокочастотные компоненты, как показано на рис. 9.3 (слева). Кроме того, первые несколько элементов производной очень зашумлены. Проблема в том, что производная на базе ДПФ предполагает периодичность сигнала. При этом последний элемент из временного ряда соединяется с его первым элементом, а это дает артефакты на краях.

Таким образом:

- вычисление разности между соседними значениями в сигнале можно выразить сверткой с простым окном. Результат будет аппроксимацией первой производной;
- дифференцирование во временной области соответствует простому фильтру в частотной области. Для периодических

сигналов это даст точную первую производную. Для некоторых непериодических сигналов это будет аппроксимация производной.

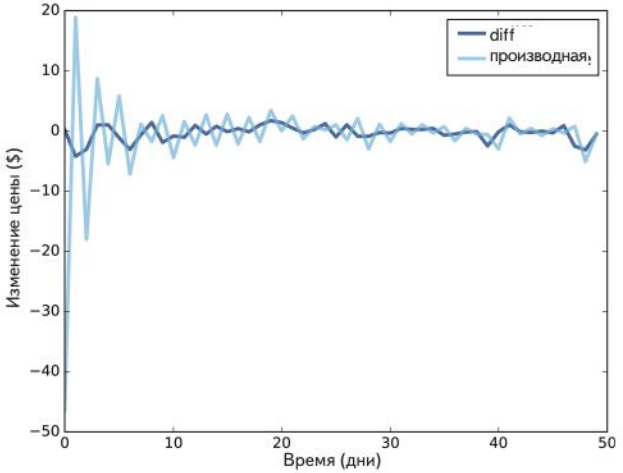

Рис. 9.4. Сравнение ежедневных изменений цен, рассчитанных с np.diff и с применением дифференцирующего фильтра

Использование ДПФ для вычисления производных – это основа *спектральных методов* решения дифференциальных уравнений (см. `http://en.wikipedia.org/wiki/Spectral_method`).

В частности, это полезно для анализа линейных стационарных (инвариантных во времени) систем, рассматриваемых в главе 10.

Интегрирование

В предыдущем разделе показано, что дифференцирование во временной области соответствует простому фильтру в частотной области: каждый компонент умножается на $2\pi if$. Поскольку интегрирование обратно дифференцированию, оно также соответствует простому фильтру: каждый компонент делится на $2\pi if$.

Вычислим этот фильтр следующим образом:

```
integ_filter = close.make_spectrum()
integ_filter.hs = 1 / (PI2 * 1j * integ_filter.fs)
```

На рис. 9.3 (справа) фильтр для удобства показан в логарифмическом масштабе по y.

`Spectrum` дает метод, накладывающий интегрирующий фильтр.

```
# class Spectrum:

    def integrate(self):
        self.hs /= PI2 * 1j * self.fs
```

Подтвердим, что интегрирующий фильтр точен, применив его к только что вычисленному спектру производной:

```
integ_Spectrum = deriv_Spectrum.copy()
integ_Spectrum.integrate()
```

Обратите внимание, что при $f = 0$ будет деление на 0. Результатом в NumPy будет NaN, особое значение с плавающей запятой, представляющее «не число». В частном случае проблему можно решить, установив эту величину в 0 перед преобразованием спектра обратно в сигнал:

```
integ_spectrum.hs[0] = 0
integ_wave = integ_spectrum.make_wave()
```

На рис. 9.5 показана проинтегрированная производная вместе с оригинальным временным рядом. Они почти идентичны, но проинтегрированная производная смещена вниз. Проблема в том, что при «насилии» над компонентом с $f = 0$ было задано нулевое смещение. И это неудивительно; обычно при дифференцировании теряется информация о смещении (постоянной составляющей), и интегрирование не может его восстановить. В некотором смысле NaN при $f = 0$ указывает, что этот элемент неизвестен.

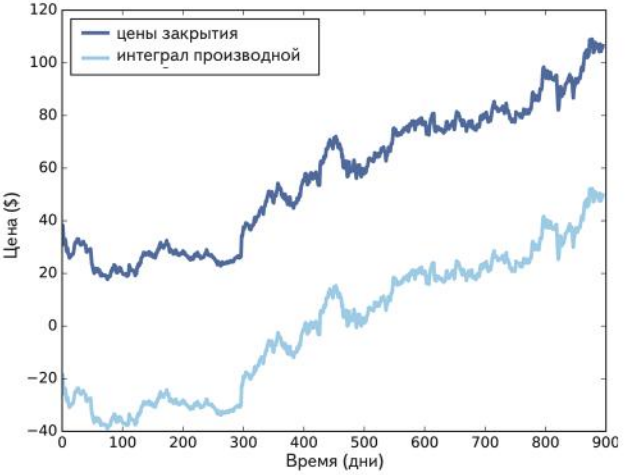

Рис. 9.5. Сравнение оригинального временного ряда и проинтегрированной производной

Если эту «постоянную интегрирования» добавить, то результаты будут идентичны, и именно поэтому интегрирующий фильтр обратен дифференцирующему.

Нарастающая сумма

Так же, как оператор `diff` аппроксимирует дифференцирование, нарастающая сумма аппроксимирует интегрирование. Покажем это на примере с пилообразным сигналом:

```
signal = thinkdsp.SawtoothSignal(freq=50)
in_wave = signal.make_wave(duration=0.1, framerate=44100)
```

На рис. 9.6 представлен этот сигнал и его спектр.

`wave` дает метод, вычисляющий нарастающую сумму массива сигнала и возвращающий новый объект `wave`:

```
# class wave:

    def cumsum (self):
        ys = np.cumsum (self.ys)
        ts = self.ts.copy()
        return wave(ys, ts, self.framerate)
```

Используем его для вычисления нарастающей суммы `in_wave`:

```
out_wave = in_wave.cumsum ()
out_wave.unbias()
```

На рис. 9.7 показан результирующий сигнал и его спектр. Сообразно упражнениям из главы 2, этот сигнал знаком: это параболический сигнал.

Сравним изображения на рис. 9.6 и 9.7 – очевидно, что амплитуды компонентов в спектре параболического сигнала спадают быстрее, чем в спектре пилообразного. В главе 2 показано, что компоненты «пилы» спадают пропорционально $1/f$. Поскольку нарастающая сумма аппроксимирует интегрирование, а интеграция фильтрует компоненты пропорционально $1/f$, то компоненты параболического сигнала спадают пропорционально $1/f^2$.

Покажем это на графике, вычислив фильтр, соответствующий нарастающей сумме:

```
cumsum_filter = diff_filter.copy()
cumsum_filter.hs = 1 / cumsum_filter.hs
```

Поскольку `cumsum` – это операция, обратная `diff`, начнем с копии `diff_filter`, с фильтра, соответствующего операции `diff`, и вычислим обратное `hs`.

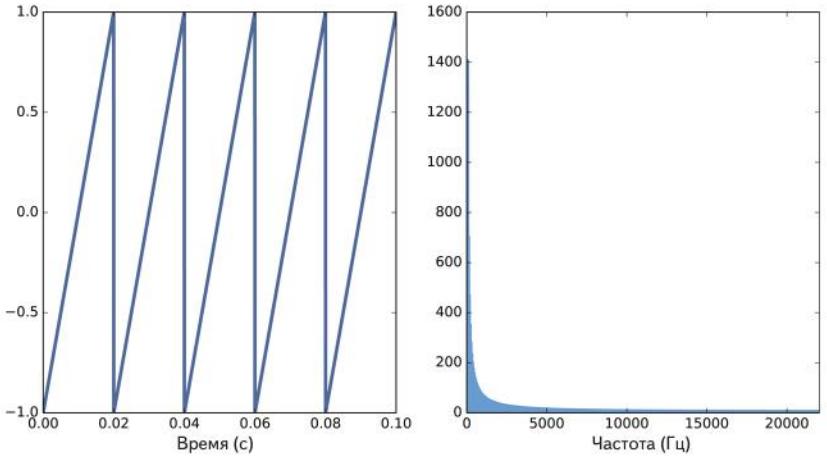

Рис. 9.6. Пилообразный сигнал и его спектр

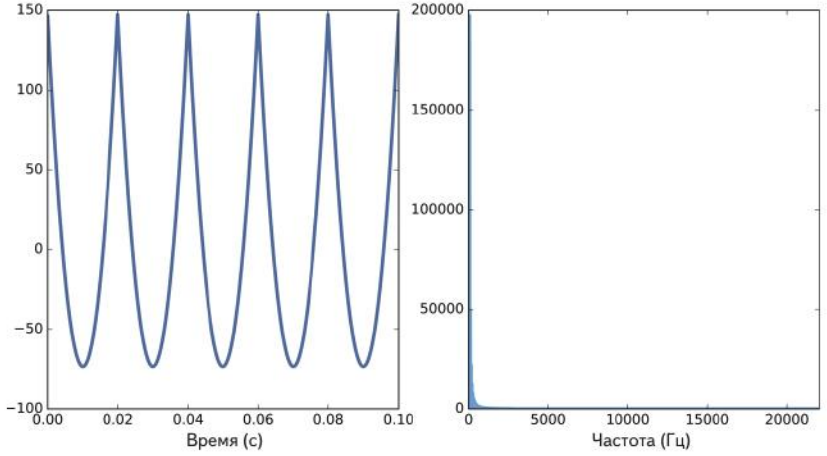

Рис. 9.7. Параболический сигнал и его спектр

На рис. 9.8 показаны фильтры, соответствующие нарастающей сумме и интегрированию. Нарастающая сумма – хорошая аппроксимация интегрирования, за исключением области высоких частот, где спад немного круче.

Подтвердить, что это правильный фильтр нарастающей суммы, можно сравнением его с соотношением спектров `out_wave` и `in_wave`:

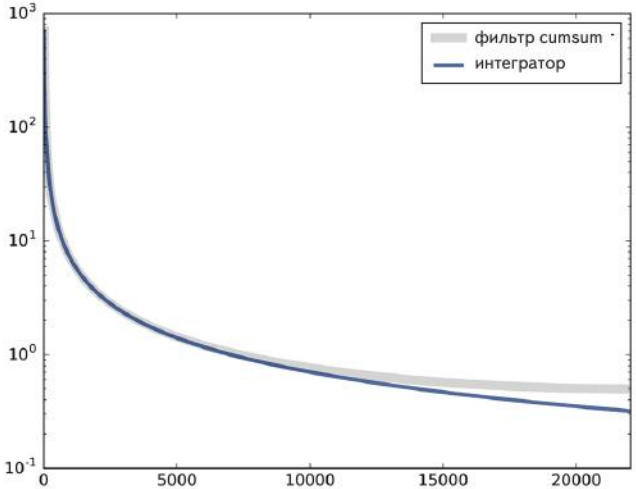

Рис. 9.8. Фильтры, соответствующие нарастающей сумме и интегрированию

```
in_spectrum = in_wave.make_spectrum()
out_spectrum = out_wave.make_spectrum()
ratio_spectrum = out_spectrum.ratio(in_spectrum, thresh=1)
```

А вот метод, вычисляющий соотношения:

```
def ratio(self, denom, thresh=1):
    ratio_spectrum = self.copy()
    ratio_spectrum.hs /= denom.hs
    ratio_spectrum.hs[denom.amps < thresh] = np.nan
    return ratio_spectrum
```

Когда `denom.amps` мал, полученное соотношение зашумлено и эти значения установлены в NaN.

На рис. 9.9 показаны соотношения и фильтр, соответствующий нарастающей сумме. Они похожи, и это подтверждает, что инвертирование фильтра `diff` дает фильтр `cumsum`.

И наконец, подтвердим теорему о свертке, применив фильтр `cumsum` в частотной области:

```
out_wave2 = (in_Spectrum * cumsum_filter).make_wave()
```

В пределах ошибки округления `out_wave2` идентичен `out_wave`, вычисленному с помощью `cumsum`, – итак, теорема о свертке работает! Но заметьте, что это верно только для периодических сигналов.

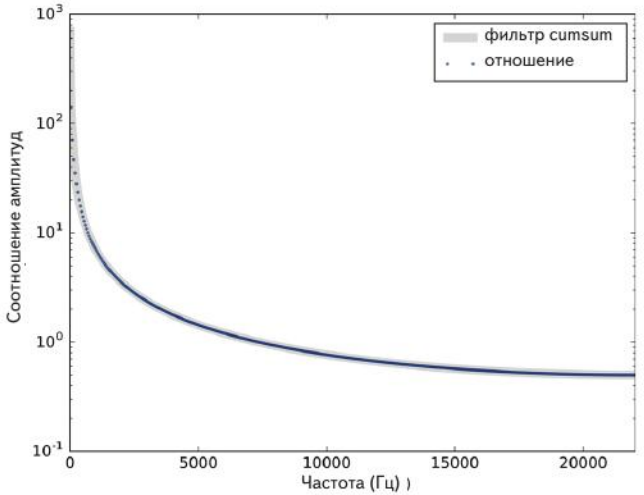

Рис. 9.9. Фильтр, соответствующий нарастающей сумме, и фактические отношения спектров до и после

Интегрирование шума

В примере из раздела «Броуновский шум» на стр. 52 броуновский шум был получен вычислением нарастающей суммы белого шума. Теперь, изучив влияние `cumsum` в частотной области, проще разобраться со спектром броуновского шума.

В среднем у белого шума одинаковая мощность на всех частотах. При вычислении нарастающей суммы амплитуда каждого компонента делится на f. Поскольку мощность – это квадрат амплитуды, мощность каждого компонента делится на f^2. Так что в среднем мощность на частоте f пропорциональна $1/f^2$:

$$P_f = K / f^2,$$

где K – несущественная константа. Логарифмируем обе части:

$$\log P_f = \log K - 2 \log f.$$

Значит, при выводе участка спектра броуновского шума в двойном логарифмическом масштабе получится прямая линия с уклоном, близким к –2.

В разделе «Конечные разности» на стр. 112 строился спектр цен закрытия на акции Facebook, и оценка наклона была близка к –1,9,

что согласуется с броуновским шумом. У цен на многие акции аналогичные спектры.

При использовании оператора `diff` для расчета ежедневных изменений *амплитуда* каждого компонента умножалась на фильтр, пропорциональный f; значит, *мощность* каждого компонента умножалась на f^2. На двойной логарифмической шкале эта операция прибавляет 2 к уклону спектра мощности, поэтому предполагаемый уклон результата близок к 0,1 (чуть меньше, ведь `diff` аппроксимирует дифференцирование).

Упражнения

Решения этих упражнений находятся в блокноте `chap09soln.ipynb`.

Упражнение 9.1

Блокнот для этой главы — `chap09.ipynb`. Прочитайте его и запустите код.

В разделе «Нарастающая сумма» на стр. 119 отмечено, что некоторые примеры не работают с апериодическими сигналами. Замените периодический пилообразный сигнал на непериодические данные Facebook и посмотрите, что пойдет не так.

Упражнение 9.2

В этом упражнении изучается влияние `diff` и `differentiate` на сигнал. Создайте треугольный сигнал и напечатайте его. Примените `diff` к сигналу и напечатайте результат. Вычислите спектр треугольного сигнала, примените `differentiate` и напечатайте результат. Преобразуйте спектр обратно в сигнал и напечатайте его. Есть ли различия в воздействии `diff` и `differentiate` на этот сигнал?

Упражнение 9.3

В данном упражнении изучается влияние `cumsum` и `integrate` на сигнал. Создайте прямоугольный сигнал и напечатайте его. Примените `cumsum` и напечатайте результат. Вычислите спектр прямоугольного сигнала, примените `integrate` и напечатайте результат. Преобразуйте спектр обратно в сигнал и напечатайте его. Есть ли различия в воздействии `cumsum` и `integrate` на этот сигнал?

Упражнение 9.4

В данном упражнении изучается влияние двойного интегрирования. Создайте пилообразный сигнал, вычислите его спектр, а затем дважды примените `integrate`. Напечатайте результирующий сигнал

и его спектр. Какова математическая форма сигнала? Почему он напоминает синусоиду?

Упражнение 9.5

В этом упражнении изучается влияние второй разности и второй производной. Создайте `CubicSignal`, определенный в `thinkdsp`. Вычислите вторую разность, дважды применив `diff`. Как выглядит результат? Вычислите вторую производную, дважды применив `differentiate` к спектру. Похожи ли результаты?

Распечатайте фильтры, соответствующие второй разнице и второй производной, и сравните их. Подсказка: для того чтобы получить фильтры в одном масштабе, используйте сигнал с частотой кадров 1.

Глава 10. Линейные стационарные системы

В настоящей главе представлены основы теории сигналов и систем, а для примеров взята музыкальная акустика. Раскрывается применение теоремы о свертке при описании линейных стационарных систем, определение которых дано ниже.

Код для этой главы находится в репозитории книги, в блокноте `chap10.ipynb` (см. раздел «Работа с кодом» на стр. 9). Также код можно просмотреть на веб-странице `http://tinyurl.com/thinkdsp10`.

Сигналы и системы

В контексте обработки сигналов *система* – это абстрактное представление чего-либо имеющего сигнал на входе и создающего сигнал на выходе.

Например, электронный усилитель принимает электрический сигнал на вход и производит усиленный сигнал на выходе.

Также и концертный зал можно считать системой, принимающей звуки музыки в месте ее проигрывания и производящей немного иной звук в месте его прослушивания.

Линейная стационарная система[1] (ЛС) – это система с двумя простыми свойствами:

1. Линейность.
   Если на вход системы подать два воздействия, то на выходе будет сумма результатов этих воздействий. Математически, если на входе x_1, то на выходе y_1, если x_2, то y_2, причем $ax_1 + bx_2$ на входе дадут $ay_1 + by_2$ на выходе, где a и b – скаляры.

[1] Это определение соответствует приведенному в статье Википедии: `http://en.wikipedia.org/wiki/LTI_system_theory` (на англ. яз.).

2. Неизменность во времени
   Свойства системы неизменны во времени и не зависят от состояния системы. Так что если воздействия x_1 и x_2 отличаются только временным сдвигом, то появляющиеся на выходе y_1 и y_2 отличаются только этим сдвигом, а в остальном идентичны.

Свойства многих физических систем таковы хотя бы в первом приближении:

- схемы, содержащие только сопротивления, емкости и индуктивности, – ЛС-системы, с точностью до соответствия компонентов своим идеализированным моделям;
- механизмы, содержащие пружины, массы и демпферы (амортизаторы), – также ЛС-системы, если пружины (сила пропорциональна смещению) и демпферы (сила пропорциональна скорости) – линейные;
- среды, передающие звук (включая воздух, воду и твердые тела), хорошо моделируются как ЛС-системы. И для нас это самое важное.

Описываются ЛС-системы линейными дифференциальными уравнениями, причем решениями этих уравнений будут комплексные синусоиды (см. статью в Википедии `http://en.wikipedia.org/wiki/Linear_differential_equation` на англ. яз.).

Теперь можно составить алгоритм вычисления воздействия ЛС-систем на входной сигнал:

1. Выразить входной сигнал суммой комплексных синусоид.
2. Вычислить соответствующий выходной компонент для каждого входного.
3. Сложить выходные компоненты.

Этот алгоритм уже хорошо знаком. Он использовался для свертки (см. раздел «Эффективная свертка», стр. 108) и для дифференцирования (см. раздел «Дифференцирование», стр. 115). Этот процесс называется *разложение в спектр*, поскольку входной сигнал «раскладывается» на спектральные компоненты.

Для применения этого процесса к ЛС-системе ее надо *охарактеризовать*, определив ее воздействие на каждый компонент входного сигнала. Так, для механических систем это легко и просто: надо «пнуть» систему и записать выходные данные.

Технически говоря, «пинок» – это *импульс*, а выход системы – *импульсная характеристика*. Удивительно, но одним импульсом можно

охарактеризовать систему целиком. Для начала вычислим ДПФ импульса. Вот массив сигнала с импульсом при $t = 0$:

```
impulse = np.zeros(8)
impulse[0] = 1
impulse_spectrum = np.fft.fft(impulse)
```

Массив сигнала выглядит следующим образом:

```
[ 1. 0. 0. 0. 0. 0. 0. 0.]
```

А вот его спектр:

```
[ 1.+0.j 1.+0.j 1.+0.j 1.+0.j 1.+0.j 1.+0.j 1.+0.j 1.+0.j]
```

Весь спектр единичный, то есть импульс есть сумма компонент с одинаковыми амплитудами на всех частотах. Этот спектр не следует путать с белым шумом, у которого одинаковая *средняя* мощность на всех частотах, но она колеблется вокруг этого среднего.

При тестировании системы подачей на вход импульса проверяется реакция системы на всех частотах. И проверять все можно одновременно – ведь система линейная, и разные составляющие не влияют друг на друга.

Окна и фильтры

Подтвердить, что вышеописанное изучение системы верно, можно на простом примере: двухэлементное скользящее среднее. Эта операция – по сути, система, принимающая сигнал в виде входных данных и выдающая на выходе слегка сглаженный сигнал.

В этом примере окно известно, и можно вычислить соответствующий фильтр. В реальности часто все не так; в следующем разделе мы рассмотрим пример, где ни об окне, ни о фильтре заранее ничего не известно.

Вот окно для вычисления двухэлементного скользящего среднего (см. раздел «Сглаживание» на стр. 99):

```
window_array = np.array([0.5, 0.5, 0, 0, 0, 0, 0, 0,])
window = thinkdsp.wave(window_array, framerate=8)
```

Найдем соответствующий фильтр вычислением ДПФ окна:

```
filtr = window.make_spectrum(full=True)
```

На рис. 10.1 показан результат. Фильтр, соответствующий окну скользящего среднего, есть фильтр НЧ, аппроксимирующий гауссову кривую.

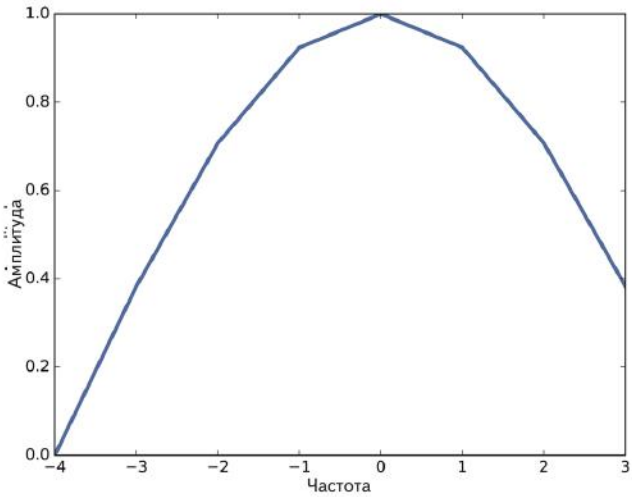

Рис. 10.1. ДПФ двухэлементного скользящего среднего

Теперь представим, что ни об окне, ни о соответствующем фильтре ничего не известно, а охарактеризовать систему надо. Сделать это можно – подать на вход импульс и измерить импульсную характеристику.

В этом примере импульсная характеристика вычисляется умножением спектра импульса на фильтр и преобразованием результата из спектра в сигнал:

```
product = impulse_spectrum * filtr
filtered = product.make_wave()
```

Так как в `impulse_Spectrum` одни единицы, произведение идентично фильтру, а отфильтрованный сигнал идентичен окну.

Из этого примера следуют два вывода:

- поскольку спектр импульса состоит из одних единиц, ДПФ импульсной характеристики идентично фильтру, характеризующему систему;
- с другой стороны, импульсная характеристика идентична окну свертки, характеризующему систему.

Акустическая характеристика

Охарактеризовать импульсом акустическую характеристику комнаты или открытого пространства очень просто – достаточно проколоть

воздушный шар или выстрелить. Получится входной сигнал, аппроксимирующий импульс, а слышимый звук саппроксимирует импульсную характеристику.

Для примера возьмем запись выстрела, охарактеризуем комнату, где был произведен выстрел, а затем применим импульсную характеристику для имитации воздействия этой комнаты на звук скрипки.

Этот пример есть в блокноте `chap10.ipynb`, размещенном в репозитории книги; просмотреть и прослушать примеры можно и на вебстранице `http://tinyurl.com/thinkdsp10`.

Вот выстрел:

```
response = thinkdsp.read_wave('180961__kleeb__gunshots.wav')
response = response.segment(start=0.26, duration=5.0)
response.normalize()
response.plot()
```

Тишина перед выстрелом удалена, сегмент начинается с 0,26 с.

На рис. 10.2 (слева) показан сигнал выстрела. Теперь вычислим ДПФ `responce`:

```
transfer = response.make_spectrum()
transfer.plot()
```

На рис. 10.2 (справа) показан результат. Этот спектр соответствует отклику комнаты; каждая частота в спектре представлена комплексным числом в виде амплитуды и фазы.

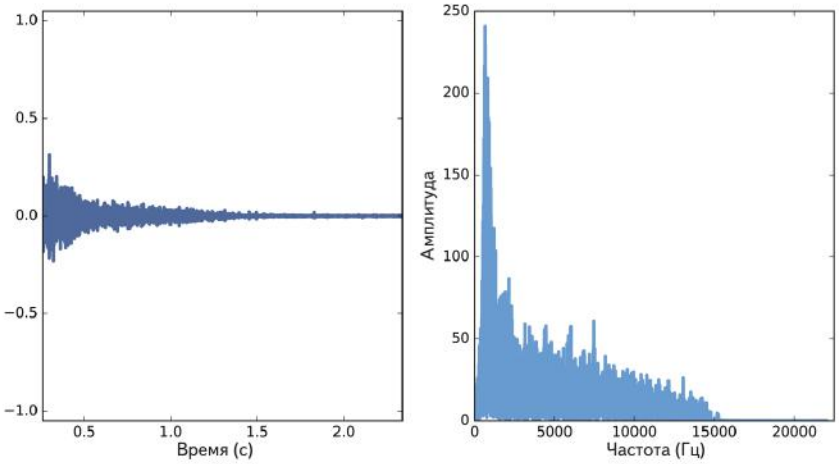

Рис. 10.2. Сигнал выстрела

Этот спектр называется *передаточной функцией*, поскольку он отражает то, как через систему передаются данные со входа на выход.

Теперь можно имитировать воздействие комнаты на звук скрипки. Возьмем запись скрипки, использованную в разделе «Периодические сигналы» на стр. 14:

```
violin = thinkdsp.read_wave('92002__jcveliz__violin-origional.wav')
violin.truncate(len(response))
violin.normalize()
```

Звуки скрипки и выстрела записаны с частотой кадров 44 100 Гц, и у них примерно одинаковая длительность. Длины сигналов скрипки и выстрела выровнены.

Вычислим ДПФ сигнала скрипки:

```
spectrum = violin.make_spectrum()
```

Теперь известны амплитуды и фазы всех частотных компонент на входе, а также передаточная функция системы. Умножение ее на `spectrum` даст ДПФ от выхода, и его можно использовать для вычисления выходного сигнала:

```
output = (spectrum * transfer).make_wave()
output.normalize()
output.plot()
```

На рис. 10.3 показаны вход (вверху) и выход (внизу) системы. Они заметно различаются, и разница хорошо слышна. Загрузите `chap10.ipynb` и прослушайте сами.

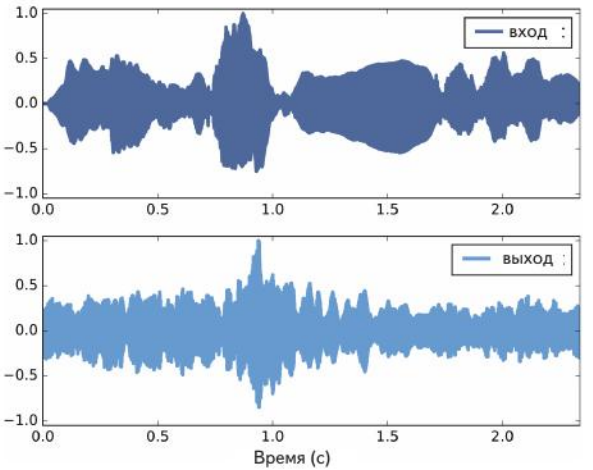

Рис. 10.3. Сигнал записи скрипки до и после свертки

В этом примере поражает одно – ощущение присутствия в комнате: звук такой, как в длинном узком помещении с твердым полом и потолком. Оно похоже на крытый стрелковый тир.

Мы до сих пор не упомянули об одной тонкости, которую следовало бы отметить. Звук скрипки при записи уже прошел через одну систему: это студия звукозаписи. Так что в примере на самом деле вычисляется звучание скрипки после двух преобразований. Для правильной имитации звучания скрипки в другой комнате следовало бы сначала охарактеризовать ту комнату, где звук скрипки был записан, и применить функцию, обратную передаточной.

Системы и свертка

Предыдущий пример – вовсе не черная магия, хотя автору до сих пор немного не по себе.

В предыдущем разделе предложено объяснение достигнутого эффекта:

- импульс состоит из компонент с амплитудой 1 на всех частотах;
- импульсная характеристика содержит сумму откликов системы на все эти компоненты;
- передаточная функция, или ДПФ импульсной характеристики, отражает воздействие системы на каждую частотную компоненту в виде амплитуды и фазы;
- для любого сигнала на входе отклик системы можно определить, разбив входные данные на компоненты: вычислить отклик на каждую компоненту и суммировать их.

Если этого мало, то есть еще один способ: свертка! Согласно теореме о свертке, умножение в частотной области соответствует свертке во временной области. В этом примере выход системы есть свертка входа и отклика системы.

Вот ключ к пониманию того, почему это сработает:

- выборки входного сигнала, по сути, – это последовательность импульсов различной амплитуды;
- каждый входной импульс «порождает» копию импульсной характеристики, соответствующую времени его появления (поскольку система стационарна во времени) и его амплитуде;

На выходе будет сумма сдвинутых во времени и масштабированных копий импульсной характеристики. Копии складываются, поскольку система линейная.

Предположим, что вместо одного выстрела произвели два: один громкий, с амплитудой 1 при $t = 0$, и один тихий, с амплитудой 0,5 при $t = 1$.

Отклик системы вычислим сложением оригинального импульсного отклика и его масштабированной и сдвинутой копии. Вот функция, генерирующая сдвинутую и масштабированную копии сигнала:

```
def shifted_scaled(wave, shift, factor):
    res = wave.copy()
    res.shift(shift)
    res.scale(factor)
    return res
```

Параметр `shift` – временной сдвиг в секундах; `factor` – коэффициент масштабирования. Вот как вычислить отклик на «салют» из двух выстрелов:

```
shift = 1
factor = 0.5
gun2 = response + shifted_scaled(response, shift, factor)
```

На рис. 10.4 показан результат. В блокноте `chap10.ipynb` можно послушать, как это звучит. Видно и слышно, что звучат два выстрела, притом первый громче второго.

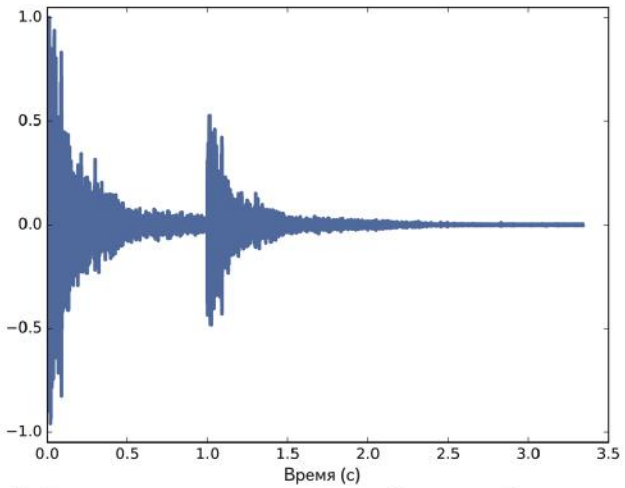

Рис. 10.4. Сумма сигнала и его сдвинутой, масштабированной копии

Теперь предположим, что вместо двух стволов взяли сто, и стреляют со скоростью 441 выстрел в секунду. Вот цикл, вычисляющий результат:

```
dt = 1 / 441
total = 0
for k in range(100):
    total += shifted_scaled(response, k*dt, 1.0)
```

При 441 выстреле в секунду расслышать отдельные выстрелы нельзя. Вместо этого слышится периодический сигнал 441 Гц. Если прослушать этот пример, то звук напомнит вам автомобильный гудок в гараже.

Это подводит к главному выводу: любой сигнал можно представить серией выборок, каждая из которых есть импульс со своей амплитудой.

В качестве примера начнем с пилообразного сигнала 441 Гц:

```
signal = thinkdsp.SawtoothSignal(freq=441)
wave = signal.make_wave(duration=0.1, framerate=response.framerate)
```

Зациклим пачку импульсов, составляющих пилу, и сложим импульсные характеристики:

```
total = 0
for t, y in zip(wave.ts, wave.ys):
    total += shifted_scaled(response, t, y)
```

В результате это будет звучать как пилообразный сигнал в крытом тире. Его можно прослушать в блокноте `chap10.ipynb`.

На рис. 10.5 показана схема вычисления, где f – пила, g – импульсная характеристика, h – сумма сдвинутых и масштабированных копий g.

$$\begin{array}{lllllllll}
f[0] & [& g[0] & g[1] & g[2] & \ldots & & &] \\
f[1] & [& & g[0] & g[1] & g[2] & \ldots & &] \\
f[2] & [& & & g[0] & g[1] & g[2] & \ldots &] \\
\hline
 & [& & & h[2] & & & &]
\end{array}$$

Рис. 10.5. Диаграмма суммирования масштабированных и сдвинутых копий g

Для показанного примера:

$$h[2] = f[0]\,g[2] + f[1]\,g[1] + f[2]\,g[0].$$

Или, в общем:

$$h[n] = \sum_{m=0}^{N-1} f[m]\,g[n-m].$$

Это уравнение из раздела «Свертка» на стр. 102. И это свертка f и g. Видно, что если вход – f, а импульсная характеристика системы – g, то выход будет сверткой f и g.

В целом есть два способа понимать влияние системы на сигнал:

1. Входной сигнал – это последовательность импульсов, поэтому на выходе – сумма масштабированных, смещенных копий импульсной характеристики. Эта сумма есть свертка входного сигнала и импульсной характеристики.
2. ДПФ импульсной характеристики есть передаточная функция, отражающая влияние системы на амплитуду и фазу каждой частотной компоненты. ДПФ входного сигнала дает амплитуду и фазу содержащихся в сигнале частотных компонент. Умножение ДПФ входного сигнала на передаточную функцию дает ДПФ выхода.

Эквивалентность этих утверждений не может быть сюрпризом. Это же теорема о свертке: свертка f и g во временной области соответствует умножению в частотной области.

При изучении окон сглаживания и разности определение свертки казалось странным, но теперь ясно, почему оно таково: определение свертки очевидно и понятно именно в отклике стационарной системы на сигнал.

Доказательство теоремы о свертке

Час настал. Докажем теорему о свертке, которая гласит:

$$\mathrm{DFT}(f * g) = \mathrm{DFT}(f)\ \mathrm{DFT}(g),$$

где f и g вектора одинаковой длины, N.

Действуем в два этапа:

1. Покажем, что в особом случае, когда f – комплексные экспоненты, свертка с g соответствует умножению f на скаляр.
2. В более общем случае, когда f не комплексные экспоненты, используем ДПФ, чтобы выразить его в виде суммы экспоненциальных компонент, вычислим свертки каждой компоненты (умножением), а затем сложим результаты.

Эти два шага, вместе взятые, доказывают теорему о свертке. Но сначала давайте соберем нужные нам части. Пусть G – это ДПФ от g, тогда:

$$\mathrm{DFT}(g)[k] = G[k] = \sum_n g[n]\exp(-2\pi i n k / N),$$

где k – индекс частоты от 0 до $N - 1$, а n – это индекс времени от 0 до $N - 1$. Результат – NumPy-массив из N комплексных чисел:

Пусть f – это обратное ДПФ от F, тогда:

$$\mathrm{IDFT}(F)[n] = f[n] = \sum_k F[k] \exp(2\pi i n k / N).$$

Вот определение свертки:

$$(f * g)[n] = \sum_m f[m]\, g[n-m],$$

где m – еще один индекс времени от 0 до $N-1$. Свертка коммутативна, так что можно записать:

$$(f * g)[n] = \sum_m f[n-m]\, g[m].$$

Теперь рассмотрим особый случай, когда f – комплексная экспонента с частотой k, назовем ее e_k:

$$f[n] = e_k[n] = \exp(2\pi i n k/N),$$

где k – индекс частоты, а n – это индекс времени.

Подставим e_k во второе определение свертки, получим:

$$(e_k * g)[n] = \sum_m \exp(2\pi i (n-m) k / N) g[m].$$

Разделим первую часть на произведение:

$$(e_k * g)[n] = \sum_m \exp(2\pi i n k / N) \exp(-2\pi i m k / N) g[m].$$

Первая половина не зависит от m, поэтому вынесем его из суммы:

$$(e_k * g)[n] = \exp(2\pi i n k / N) \sum_m \exp(-2\pi i m k / N) g[m].$$

Теперь видно, что первый член – это e_k, а сумма – это $G[k]$ (используя m как индекс времени). Теперь можно записать:

$$(e_k * g)\,[n] = e_k[n]\; G[k].$$

Видно, что для каждой комплексной экспоненты, e_k свертка с g дает умножение e_k на $G[k]$. В математических терминах каждый e_k есть собственный вектор этой операции, а $G[k]$ – соответствующее собственное значение (см. раздел «Дифференцирование» на стр. 115).

Переходим ко второй части доказательства. Если входной сигнал f не будет комплексной экспонентой, можно выразить его как совокупность комплексных экспонент, вычислив его ДПФ, F. Для каждого

значения k от 0 до $N-1$ выражение $F[k]$ есть комплексная амплитуда компоненты с частотой k.

Каждая входная компонента есть комплексная экспонента с амплитудой $F[k]$, поэтому каждая выходная компонента есть комплексная экспонента с величиной $F[k]\ G[k]$, согласно первой части доказательства.

Поскольку система линейная, на выходе просто сумма выходных компонент:

$$(f * g)[n] = \sum_k F[k]G[k]e_k[n].$$

Подстановка в определение e_k дает:

$$(f * g)[n] = \sum_k F[k]G[k]\exp(2\pi i n k / N).$$

Правая часть есть обратное ДПФ произведения FG. Таким образом:

$$(f * g) = \text{IDFT}(FG).$$

Подставим $F = \text{DFT}(f)$ и $G = \text{DFT}(g)$

$$(f * g) = \text{IDFT}(\text{DFT}(f)\ \text{DFT}(g)).$$

И наконец взятие ДПФ обеих частей дает теорему о свертке:

$$\text{DFT}(f * g) = \text{DFT}(f)\ \text{DFT}(g).$$

...Что и требовалось доказать.

Упражнения

Решения этих упражнений находятся в блокноте `chap10soln.ipynb`.

Упражнение 10.1

В разделе «Системы и свертка» на стр. 131 свертка описана как сумма сдвинутых и масштабированных копий сигнала.

А в разделе «Акустическая характеристика» на стр. 128 умножение ДПФ сигнала на передаточную функцию соответствует *круговой свертке*, но в предположении периодичности сигнала. В результате можно заметить, что на выходе, в начале фрагмента, слышна лишняя нота, «затекшая» из конца этого фрагмента.

К счастью, есть стандартное решение этой проблемы. Если перед вычислением ДПФ добавить достаточно нулей в конец сигнала, эффекта «заворота» можно избежать.

Измените пример в `chap10.ipynb` и убедитесь, что дополнение нулями устраняет лишнюю ноту в начале фрагмента.

Упражнение 10.2

Библиотека Open AIR (свободный эфир) – это «централизованный... онлайн-ресурс для тех, кто интересуется аурализацией и данными акустической импульсной характеристики» (`http://www.openairlib.net`). Просмотрите эту коллекцию импульсных характеристик и скачайте ту, звучание которой интереснее. Найдите короткие записи с той же частотой дискретизации, что и у скачанной импульсной характеристики.

Смоделируйте двумя способами звучание записи в том пространстве, где была измерена импульсная характеристика, как сверткой самой записи с импульсной характеристикой, так и умножением ДПФ записи на вычисленный фильтр, соответствующий импульсной характеристике.

Глава 11. Модуляция и выборка (квантование)

В разделе «Сглаживание» на стр. 99 показано, что при выборках сигнала на 10 000 Гц компонента 5500 Гц ничем не отличается от компоненты на 4500 Гц. Здесь частота заворота 5000 Гц, она равна половине частоты дискретизации. Почему это так, пока не важно.

В этой главе исследуются свойства выборок и представляется теорема отсчетов (о выборках), в ней объясняются биения и заворот частот.

Начнем с изучения эффекта свертки с импульсами; это удобно для изучения амплитудной модуляции (AM) и полезно для понимания теоремы отсчетов.

Код для этой главы находится в репозитории книги, в блокноте `chap11.ipynb` (см. раздел «Работа с кодом» на стр. 9). Его также можно просмотреть на `http://tinyurl.com/thinkdsp-ch11`.

Свертка с импульсами

В разделе «Системы и свертка» на стр. 131 показано, что свертка сигнала с пачкой импульсов состоит в сложении смещенных и масштабированных копий сигнала.

Для примера загрузим сигнал, звучащий как зуммер:

```
filename = '253887 themusicalnomad positive-beeps.wav'
wave = thinkdsp.read_wave(filename)
wave.normalize()
```

И построим сигнал с четырьмя импульсами:

```
imp_sig = thinkdsp.Impulses([0.005,  0.3,  0.6,  0.9],
                            amps=[1,      0.5,  0.25, 0.1])
impulses = imp_sig.make_wave(start=0, duration=1.0,
                             framerate=wave.framerate)
```

...а затем свернем их:

```
convolved = wave.convolve(impulses)
```

На рис. 11.1 показаны результаты: в верхнем левом углу – сигнал, в нижнем левом углу – импульсы, а справа – результат.

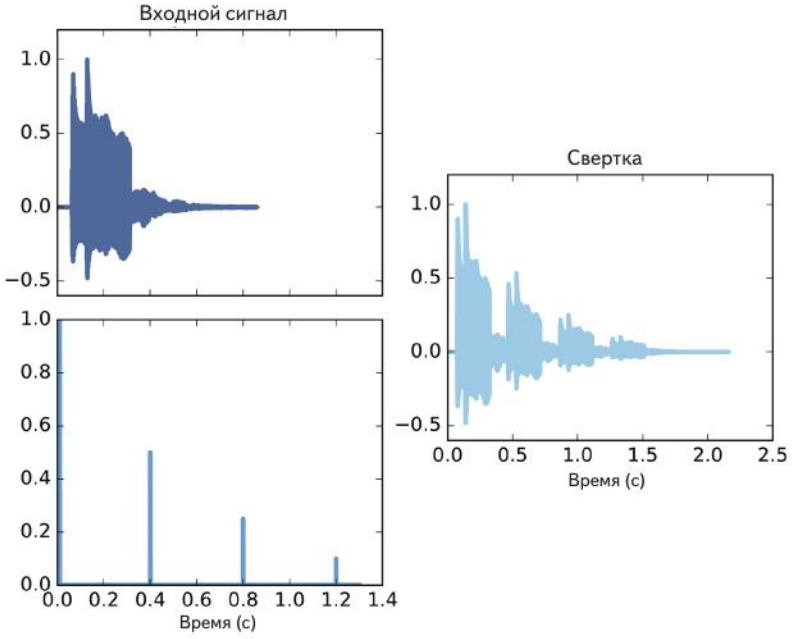

Рис. 11.1. Эффект свертки сигнала (вверху слева) с пачкой импульсов (внизу слева). Результат (справа) есть сумма смещенных, масштабированных копий сигнала

Результат можно прослушать в `chap11.ipynb`; это серия четырех коротких звуковых сигналов с уменьшающейся громкостью.

Данный пример просто показывает, что свертка сигнала с импульсами дает сдвинутые, масштабированные копии. И это понадобится нам для следующего раздела.

Амплитудная модуляция

Амплитудная модуляция (АМ) применяется в радиовещании (ДВ, СВ, КВ) и в других областях. В передатчике косинусоидальный сигнал, называемый «несущая», «модулируется» умножением на по-

лезный сигнал (речь, музыка и т. п.). Результатом будет высокочастотный сигнал, пригодный для радиовещания. Диапазон частот для АМ-радиостанций – 500–1600 кГц (см. статью в Википедии `https://en.wikipedia.org/wiki/AM_broadcasting` на англ. яз.).

На стороне приема вещательный сигнал «демодулируется» для восстановления оригинального сигнала. Удивительно, но демодуляция также основана на умножении вещательного сигнала на сигнал несущей.

Рассмотрим, как это работает, модулируя сигналом несущую 10 кГц. Вот сигнал:

```
filename = '105977 wcfl10 favorite-station.wav'
wave = thinkdsp.read_wave(filename)
wave.unbias()
wave.normalize()
```

А вот несущая:

```
carrier_sig = thinkdsp.CosSignal(freq=10000)
carrier_wave = carrier_sig.make_wave(duration=wave.duration,
                                     framerate=wave.framerate)
```

Перемножим массивы сигналов поэлементно с помощью оператора *:

```
modulated = wave * carrier_wave
```

Результат не очень впечатляет. Прослушать его можно в `chap11.ipynb`.

На рис. 11.2 показано, как выглядит частотная область. Верхняя часть – спектр оригинального сигнала. Ниже – спектр модулированного сигнала после перемножения с несущей. В нем две копии оригинального спектра, сдвинутые на ±10 кГц.

Выясним, почему. Вспомним, что свертка во временной области соответствует умножению в частотной области. И наоборот, умножение во временной области соответствует свертке в частотной области. Перемножение сигнала с несущей сворачивает его спектр с ДПФ несущей.

Так как несущая – просто косинусоидальный сигнал, ее ДПФ – два импульса, ±10 кГц. Свертка с этими импульсами дает смещенные, масштабированные копии спектра. Заметим, что амплитуды спектра после модуляции уменьшились. Энергия оригинального сигнала разделилась между копиями.

Демодулируем сигнал также умножением на сигнал несущей:

```
demodulated = modulated * carrier_wave
```

На рис. 11.2 (третье изображение сверху) показан результат. Вновь умножение во временной области соответствует свертке в частотной области, и получаются смещенные, масштабированные копии спектра.

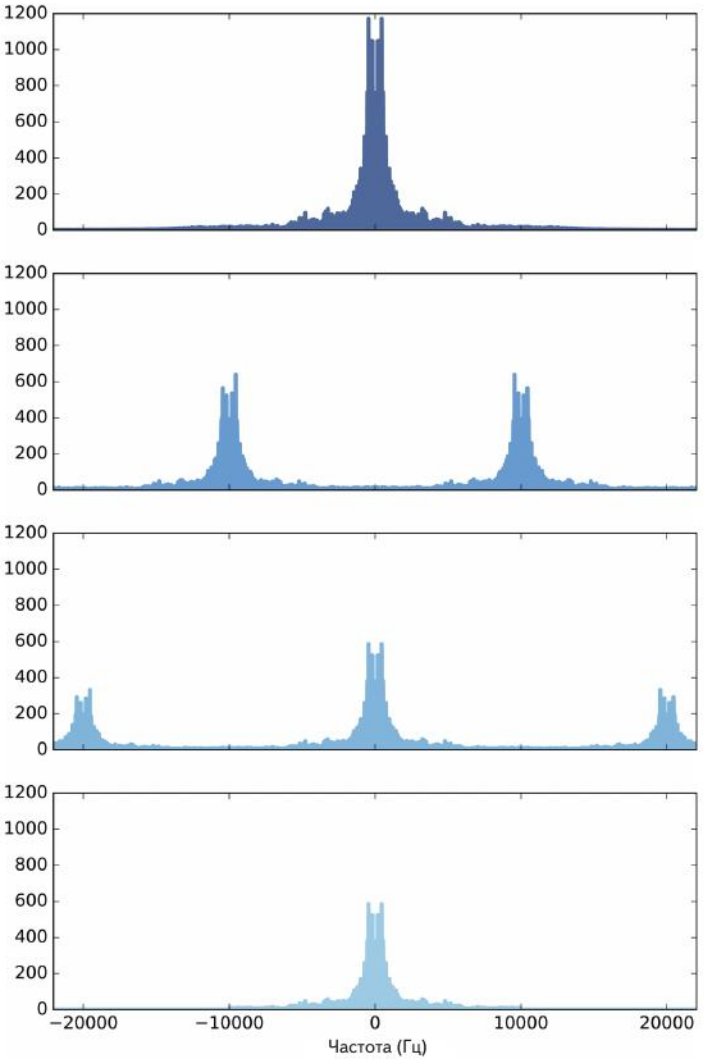

Рис. 11.2. Пример амплитудной модуляции. Верхний график – спектр сигнала; под ним – спектр после модуляции; далее – спектр после демодуляции; нижний – демодулированный сигнал после фильтра НЧ

В модулированном спектре два пика, и они оба раздваиваются и смещаются на ±20 кГц. При этом две из четырех копий складываются в 0 кГц, а другие две оказываются на ±20 кГц.

При прослушивании демодулированный сигнал звучит очень хорошо. Хоть копии спектра и добавили высокочастотные компоненты, отсутствовавшие в оригинальном сигнале, но они так высоки, что большинство динамиков их не воспроизводит, а большинство людей их просто не слышит. Если же у вас хорошие колонки и выдающийся слух, то эту «грязь» можно разобрать.

Избавиться от лишних компонентов можно, применив фильтр НЧ:

```
demodulated_spectrum = demodulated.make_spectrum(full=True)
demodulated_spectrum.low_pass(10000)
filtered = demodulated_spectrum.make_wave()
```

Результат достаточно близок к оригинальному сигналу, но примерно половина мощности теряется при демодуляции и фильтрации. На практике это не страшно, потому что гораздо больше энергии теряется при передаче и приеме. А результат в любом случае надо усиливать, так что проблем нет.

Выборка

Амплитудная модуляция пока показана узко – только самые основы, но этого достаточно, чтобы разобраться с выборками. *Выборка* – процесс измерения аналогового сигнала в серии моментов времени, обычно через равные промежутки.

Например, WAV-файлы, использованные в примерах, записаны выборкой из выходного сигнала микрофона с помощью аналого-цифрового преобразователя (АЦП). Частота выборок у большинства из них 44,1 кГц (стандартная частота для звука «CD-качества») или 48 кГц (стандартная для звука DVD).

При 48 кГц частота заворота будет 24 кГц, что выше, чем слышит большинство людей (см. статью в Википедии `https://en.wikipedia.org/wiki/Hearing_range` на англ. яз.).

В большинстве этих записей в каждой выборке 16 бит, то есть 2^{16} уровней. Такой «разрядности» достаточно, и увеличение числа битов(bit depth) почти не улучшает качество звука (см. статью в Википедии `https://en.wikipedia.org/wiki/Digital_audio` на англ. яз.).

Конечно, в приложениях, не связанных со звуком, может потребоваться большая частота дискретизации, чтобы охватить более высокие частоты, или большая разрядность для воспроизведения сигналов с большей достоверностью.

Рассматривать эффект выборки начнем с сигнала, квантованного с частотой 44,1 кГц, и выберем из него отсчеты, соответствующие частоте примерно 11 кГц. Это не совсем то же, что выборки из аналогового сигнала, но эффект идентичный.

Сначала загрузим запись соло на барабане:

```
filename = '263868 kevcio amen-break-a-160-bpm.wav'
wave = thinkdsp.read_wave(filename)
wave.normalize()
```

На рис. 11.3 (верхний) показан спектр этого сегмента. А вот функция для выборки из сигнала:

```
def sample(wave, factor=4):
    ys = np.zeros(len(wave))
    ys[::factor] = wave.ys[::factor]
    return thinkdsp.wave(ys, framerate=wave.framerate)
```

Используем ее для выбора каждого четвертого элемента:

```
sampled = sample(wave, 4)
```

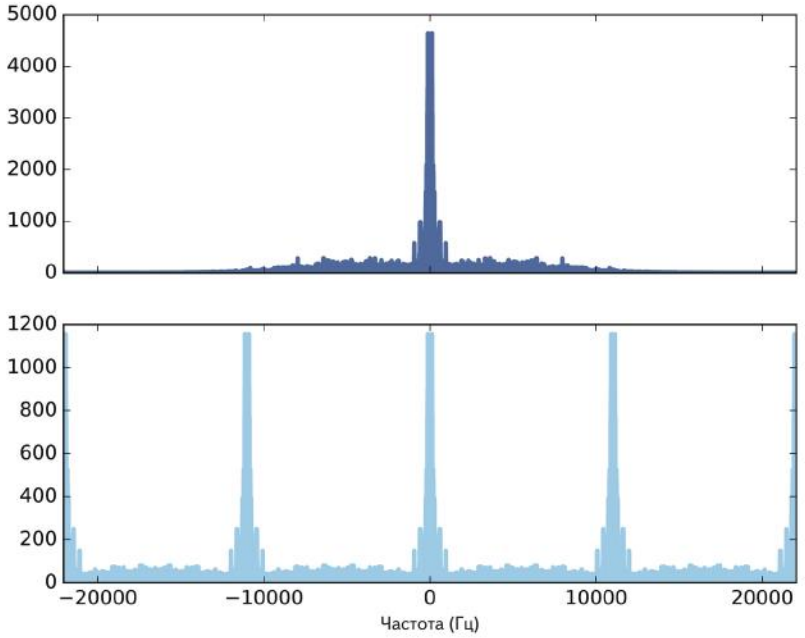

Рис. 11.3. Спектр сигнала до и после выборки (вверху и внизу соответственно)

У результата частота кадров оригинала, но большинство элементов равны нулю. При воспроизведении этот сигнал звучит не очень хорошо. Процесс выборки добавляет высокочастотные компоненты, которых не было в оригинале.

На рис. 11.3 (внизу) показан спектр. Он содержит четыре копии оригинального спектра (выглядят как пять копий, потому что один спектр «раздвоился» между высокими и низкими частотами).

Выясним, откуда берутся эти копии – представим процесс выборки как умножение на серию импульсов. Вместо использования `sample` для выбора каждого четвертого элемента по отдельности используем эту функцию для серии импульсов, которую иногда называют *пачка импульсов*:

```
def make_impulses(wave, factor):
    ys = np.zeros(len(wave))
    ys[::factor] = 1
    ts = np.arange(len(wave)) / wave.framerate
    return thinkdsp.wave(ys, ts, wave.framerate)
```

А затем умножим оригинальный сигнал на пачку импульсов:

```
impulses = make_impulses(wave, 4)
sampled = wave * impulses
```

Результат тот же самый; он по-прежнему звучит не слишком убедительно, но причина теперь ясна. Умножение во временной области соответствует свертке в частотной области. При умножении на пачку импульсов происходит свертка с ДПФ пачки импульсов. Получается, что ДПФ пачки импульсов – также пачка импульсов.

На рис. 11.4 показаны два примера. Верхняя строка – пачка импульсов из примера с частотой 11 025 Гц. ДПФ – пачка из четырех импульсов, поэтому и получается четыре копии спектра. В нижней части показана пачка импульсов с меньшей частотой, около 5512 Гц. Ее ДПФ – пачка из восьми импульсов. В целом большему числу импульсов во временной области соответствует меньшее число импульсов в частотной области.

Обобщим сказанное:

- взятие выборок можно представить умножением на пачку импульсов;
- умножение на пачку импульсов соответствует свертке с пачкой импульсов в частотной области;
- свертка с пачкой импульсов дает несколько копий спектра сигнала.

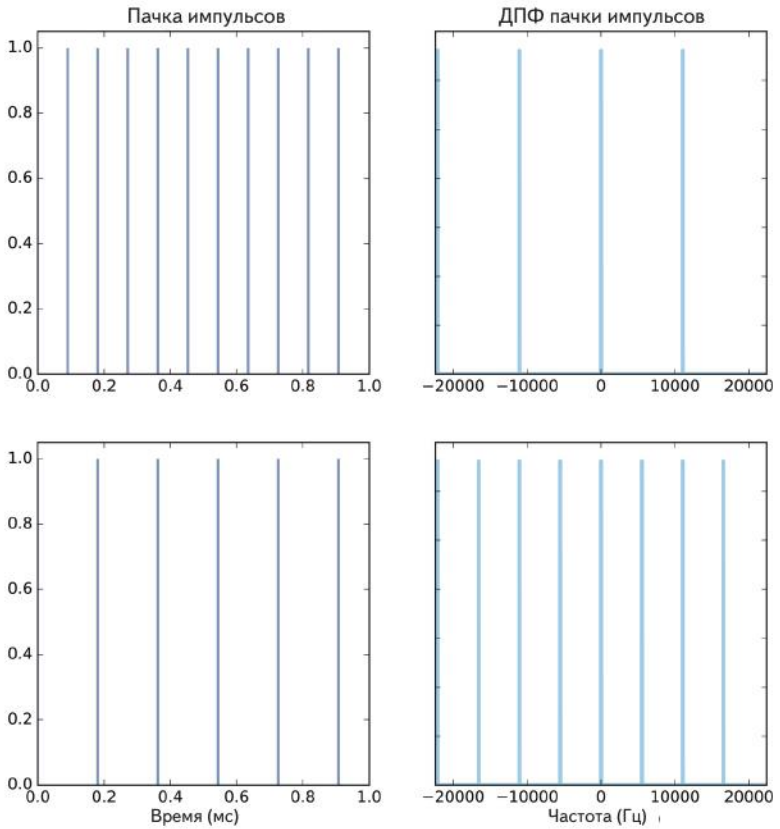

Рис. 11.4. ДПФ пачки импульсов – также пачка импульсов

Биения

В разделе «Амплитудная модуляция» на стр. 139 после демодуляции АМ сигнала дополнительные копии спектра убирались при помощи фильтра НЧ. То же самое можно сделать после взятия выборок, но это не лучшее решение.

На рис. 11.5 показано, почему. Верхняя часть – спектр сигнала соло на барабане. В нем есть высокочастотные компоненты, превышающие 10 кГц. При взятии выборок из этого сигнала его спектр сворачивается с пачкой импульсов (вторая часть) и появляются копии спектра (третья часть). В нижней части показан результат после применения фильтра НЧ со срезом на частоте заворота, 5512 Гц.

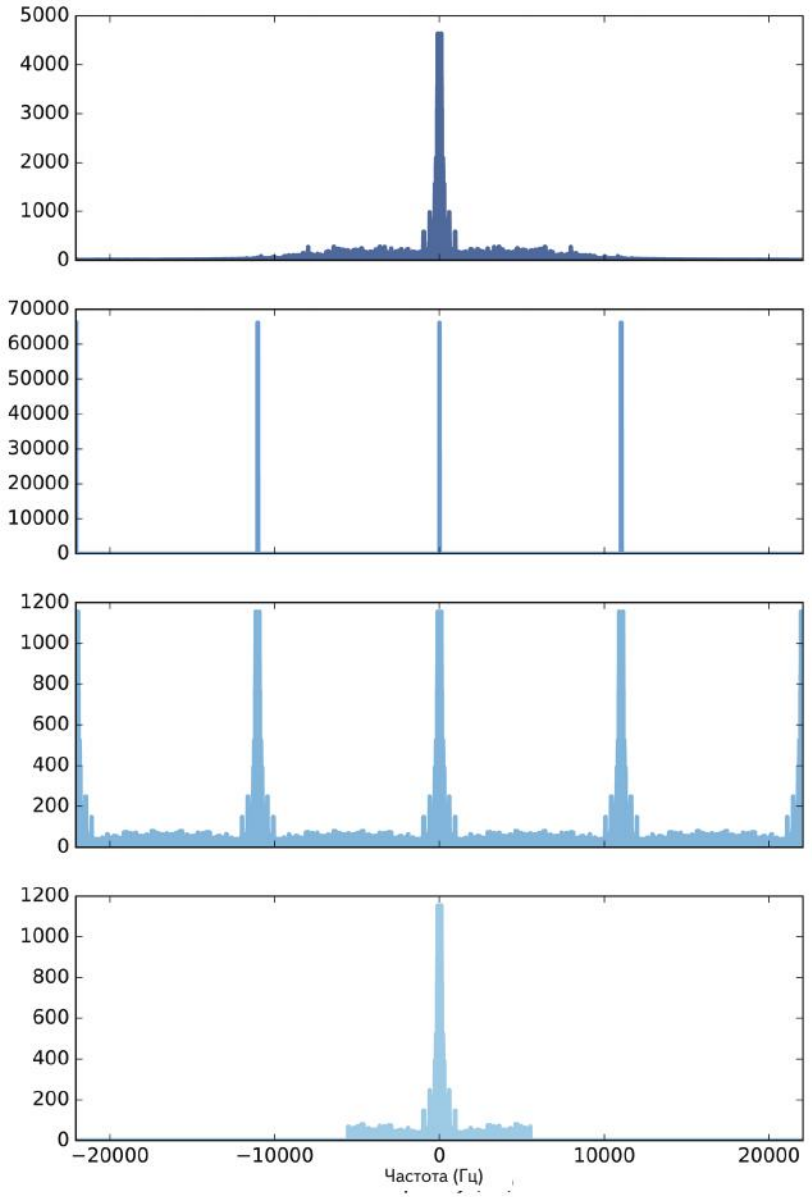

Рис. 11.5. Спектры соло на барабане (вверху), серия (пачка) импульсов (вторая строка), выборки из сигнала (третья строка) и результат после фильтра НЧ (внизу)

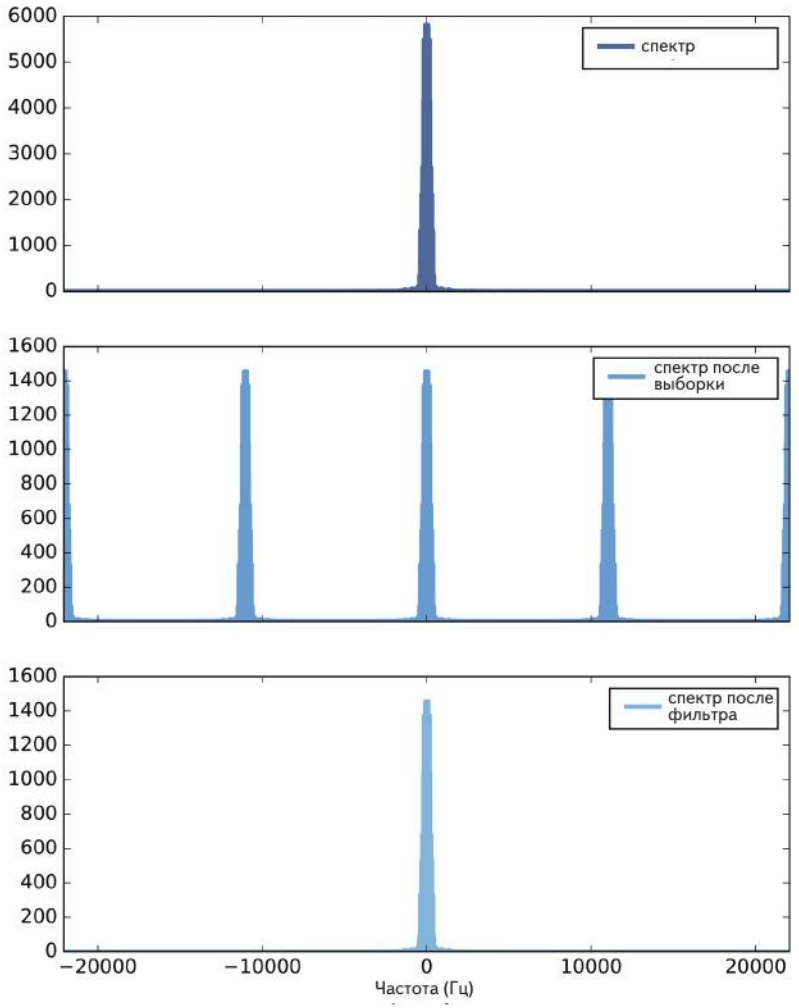

Рис. 11.6. Спектр соло бас-гитары (вверху), после выборок (в середине) и после фильтрации (внизу)

При преобразовании результата обратно в сигнал он будет похож на оригинальный, но появятся две проблемы:

- из-за фильтра НЧ потеряны компоненты выше 5500 Гц, и в результате звук приглушен;
- компоненты ниже 5500 Гц тоже не совсем верные, поскольку присутствуют остатки отфильтрованных копий спектра.

Если спектральные копии после выборки перекрываются, теряется часть информации о спектре, и его нельзя восстановить точно.

Если же копии не перекрываются, то все работает достаточно хорошо. Для второго примера загрузим запись соло бас-гитары.

На рис. 11.6 показан спектр этой записи (верхняя часть), и в нем нет видимой энергии выше 5512 Гц. Во второй части показан спектр выборок сигнала, а в третьей – спектр после фильтра НЧ. Амплитуда меньше, поскольку некоторая часть энергии отфильтровалась, но форма спектра почти точно та, что и в начале. И после преобразования обратно в сигнал звучание будет то же.

Интерполяция

Фильтр НЧ, использованный на последнем шаге – это прямоугольный фильтр (brick wall – «кирпичная стена»); все, что выше частоты среза, будет удалено полностью, как при разделе кирпичной стеной.

В правой части рис. 11.7 показано, что это за фильтр. Конечно, умножение на этот фильтр в частотной области соответствует свертке с окном во временной области. Выясним, что это за окно, вычислив обратное ДПФ фильтра – оно показано в левой части рис. 11.7.

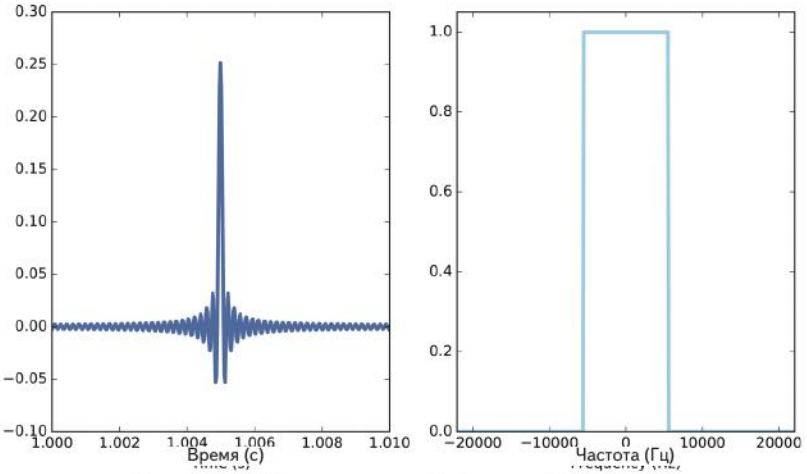

Рис. 11.7. Прямоугольный фильтр НЧ (справа) и соответствующее окно свертки (слева)

У этой функции есть имя – это нормализованная *sinc-функция*, или ее дискретная аппроксимация (см. статью в Википедии `https://en.wikipedia.org/wiki/Sinc_function` на англ. яз.):

$$\mathrm{sinc}(x) = \sin\pi x / \pi x$$

При применении фильтра НЧ сигнал сворачивается с sinc-функцией. Такую свертку можно понимать как сумму смещенных, масштабированных копий sinc-функции.

В нуле sinc равен 1, а при каждом целом x он равен 0. При сдвиге sinc-функции сдвигается точка нуля. При масштабировании sinc изменяется его размах в нуле. Поэтому при суммировании смещенных масштабированных копий они интерполируют между точками выборки.

На рис. 11.8 показано, как это работает на коротком сегменте соло на бас-гитаре. В верхней части – оригинальный сигнал. Вертикальные серые линии – точки выборок. Тонкие кривые – это смещенные, масштабированные копии sinc-функции. Сумма этих sinc-функций идентична оригинальному сигналу.

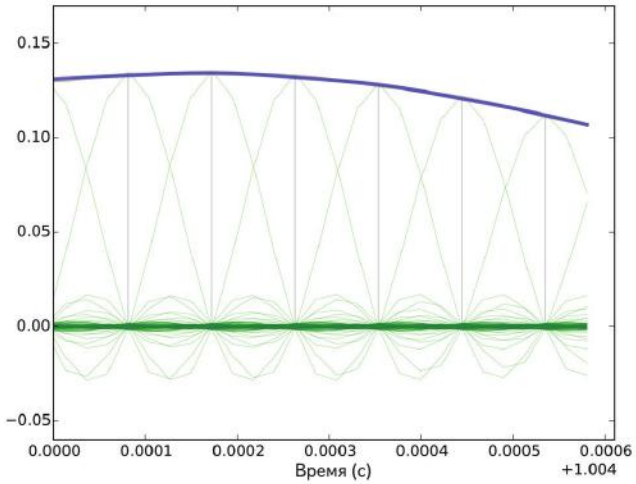

Рис. 11.8. Крупный план последовательности выборок (вертикальные серые линии), интерполирующие sinc-функции (тонкие кривые) и оригинальный сигнал (более толстые линии вверху)

Повторим еще раз, поскольку это удивительно и важно:

Сумма этих sinc-функций идентична оригинальному сигналу.

Был взят сигнал, не содержащий энергии выше 5512 Гц, из него взяты выборки с частотой 11 025 Гц, и удалось получить точный оригинальный спектр. Из точного оригинального спектра можно восстановить точный оригинальный сигнал.

В этом примере сигнал с выборками частотой 44100 Гц был «перевыбран» с частотой 11 025 Гц. После перевыборки разрыв между копиями спектра составляет 11 025 кГц.

Если в оригинальном сигнале нет энергии выше 5512 Гц, копии спектра не перекрываются, информация не теряется, и оригинальный сигнал можно точно восстановить.

Этот результат известен как *теорема Найквиста-Котельникова-Шеннона* (см. `https://ru.wikipedia.org/wiki/Теорема_Котельникова`), или *теорема отсчетов*, или *теорема о выборках*.

Этот пример теорему не доказывает, но он поможет вам понять, о чем она и как она работает.

Обратите внимание, что созданный аргумент не зависит от первоначальной частоты выборок, 44,1 кГц. Результат будет таким же, если выборки из оригинала будут на более высокой частоте, или даже если оригинал – непрерывный аналоговый сигнал: если выборки взяты с частотой f, точно восстановить оригинальный сигнал можно только тогда, когда он не содержит энергии на частотах выше $f/2$. О таком сигнале говорят, что у него *ограниченная полоса*.

Итог

Поздравляем! Вот и конец книги (не считая еще нескольких упражнений, приведенных ниже). Прежде чем закрыть книгу, посмотрим, что осталось позади:

- изучены периодические сигналы и их спектры, представлены ключевые объекты в библиотеке `thinkdsp`: `signal`, `wave` и `Spectrum`;
- рассмотрена гармоническая структура простых сигналов и записей музыкальных инструментов, исследован эффект биений;
- с помощью спектрограмм исследованы чирпы и другие звуки с изменяющимися во времени спектрами;
- сгенерированы и проанализированы шумовые сигналы, охарактеризованы природные источники шума;
- использована автокорреляционная функция для оценки высоты тона и дополнительной оценки шума;
- рассмотрено дискретное косинусное преобразование (ДКП), полезное для сжатия, а также для понимания дискретного преобразования Фурье (ДПФ);

- использованы комплексные экспоненты для синтеза комплексных сигналов, и обращен процесс разработки ДПФ. Если упражнения в конце главы 7 были выполнены, то реализовано и быстрое преобразование Фурье (БПФ), один из самых важных алгоритмов XX века;
- введено сглаживание, представлены определение свертки и теорема о свертке, связывающая операции сглаживания во временной области с фильтрацией в частотной области;
- исследованы дифференцирование и интегрирование в виде линейных фильтров, то есть основы спектральных методов решения дифференциальных уравнений. Объяснены и некоторые эффекты, встреченные в предыдущих главах, типа соотношения между белым и броуновским шумом;
- изучены начала теории ЛС-систем, а теорема о свертке применена для описания таких систем по их импульсной характеристике;
- представлена амплитудная модуляция (АМ), играющая важную роль как в радиосвязи, так и в понимании теоремы о выборках – неожиданный, но важнейший в цифровой обработке сигналов результат.

Итак, получен хороший баланс практических знаний (в работе с сигналами и спектрами при помощи вычислительных средств) и теории (понимание того, как и почему работают выборки и фильтрация).

Автор надеется, что книга была полезной. Спасибо за внимание!

Упражнения

Решения этих упражнений находятся в блокноте `chap11soln.ipynb`.

Упражнение 11.1

Код для этой главы содержится в блокноте `chap11.ipynb`. Изучите его и прослушайте примеры.

Упражнение 11.2

Крис «Монти» Монтгомери (Chris “Monty” Montgomery) сделал отличное видео под названием «D/A and A/D | Digital Show and Tell». Он демонстрирует теорему о выборках в действии, и представляет множество другой замечательной информации о выборках. См. видеоролик `https://www.youtube.com/watch?v=cIQ9IXSUzuM`.

Упражнение 11.3

Выше показано, что при взятии выборок из сигнала при слишком низкой частоте кадров составляющие, большие частоты заворота дадут биения. В таком случае эти компоненты не отфильтруешь, поскольку они неотличимы от более низких частот.

Полезно отфильтровать эти частоты до выборки; фильтр НЧ, используемый для этой цели, называется *фильтр сглаживания*.

Вернитесь к примеру «Соло на барабане», примените фильтр НЧ до выборки, а затем, опять же с помощью фильтра НЧ, удалите спектральные копии, вызванные выборкой. Результат должен быть идентичен отфильтрованному сигналу.

Предметный указатель

Н

О

П

Р

С

Т

У

Ф

Х

Ц

Ч

Ш

Э

A

B

C

D

F

G

I

J

L

M

N

Об авторе

Аллен Б. Дауни (Allen B. Downey) – профессор компьютерных наук в инженерном колледже Олин. Он преподавал в колледже Уэллсли, колледже Колби и в Калифорнийском университете Беркли. Получил степень доктора компьютерных наук в Калифорнийском университете Беркли, а также степени магистра и бакалавра в Массачусетском технологическом институте.

Об обложке

Животное на обложке *Think DSP* – гладкоклювый ани (*Crotophaga ani*), большая птица семейства кукушковых. Она распространена во Флориде, на Багамских и Карибских островах и в некоторых районах Центральной и Южной Америки.

У гладкоклювого ани черное оперение, длинный хвост и большой клюв с ребрами. Питаются эти птицы на земле, поедая термитов, насекомых и даже небольших ящериц или лягушек. Предпочитают полуоткрытые места, где поля соседствуют с зарослями. Рост городов и вырубка лесов повлияли на ареал распространения ани, но птицы приспособились к пастбищам при фермах и поедают насекомых и паразитов, обирая их с шерсти скота.

Этот вид очень социален и всегда живет шумными стаями. В брачный период несколько пар гнездятся совместно, поочередно строя чашеобразное гнездо высоко на дереве. Так же по очереди они высиживают яйца и кормят птенцов. Каждая самка откладывает 4–7 яиц; встречались гнезда, в которых насчитывалось до 29 яиц.

Многие из животных на обложках книг O'Reilly находятся под угрозой исчезновения; сохранение этих видов чрезвычайно важно для нашей планеты. Если вы хотите узнать, как можно им помочь, посетите сайт `animals.oreilly.com`.

Изображение на обложке предоставлено *Braukhaus Lexicon*.

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «Планета Альянс» наложенным платежом, выслав открытку или письмо по почтовому адресу: **115487, г. Москва, 2-й Нагатинский пр-д, д. 6А**.

При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя. Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине: **www.alians-kniga.ru**.

Оптовые закупки: **тел. +7 (499) 782-38-89**.

Электронный адрес: **books@alians-kniga.ru**.

Аллен Б. Дауни

Think DSP

Цифровая обработка сигналов на Python

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод с английского *Бряндинский А. Э.*
Корректор *Готлиб О. В.*
Верстка *Паранская Н. В.*
Дизайн обложки *Мовчан А. Г.*

Формат 60×90 1/16. Гарнитура «Петербург».
Печать офсетная. Усл. печ. л. 9,8.
Тираж 200 экз.

Веб-сайт издательства: www.дмк.рф